

Lecture Notes in Computer Science

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

1467

Chris Clack Kevin Hammond
Tony Davie (Eds.)

Implementation of Functional Languages

9th International Workshop, IFL'97
St. Andrews, Scotland, UK
September 10-12, 1997
Selected Papers



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Chris Clack
University College London, Department of Computer Science
Gower Street, London WC1E 6BT, UK
E-mail: clack@cs.ucl.ac.uk

Kevin Hammond
Tony Davie
University of St. Andrews, Division of Computer Science
North Haugh, St. Andrews, Scotland KY16 9SS, UK
E-mail: {kh,ad}@dcs.st-and.ac.uk

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Implementation of functional languages : 9th international workshop ; selected papers / IFL '97, St. Andrews, Scotland, UK, September 10 - 12, 1997. Chris Clack ... (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Budapest ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1998
(Lecture notes in computer science ; Vol. 1467)
ISBN 3-540-64849-6

CR Subject Classification (1991): D.3, D.1.1, F.3

ISSN 0302-9743

ISBN 3-540-64849-6 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1998
Printed in Germany

Typesetting: Camera-ready by author
SPIN 10638546 06/3142 - 5 4 3 2 1 0 Printed on acid-free paper

Preface and Overview of Papers

This volume contains a selection of papers presented at the 1997 International Workshop on the Implementation of Functional Languages (IFL '97), held at St. Andrews in Scotland, September 10–12, 1997. This is the ninth in a series of workshops that were previously held in The Netherlands, Germany, Sweden, and the UK, and the second to be published in the Springer-Verlag series of Lecture Notes in Computer Science (selected papers from IFL '96 are published in LNCS Volume 1268).

The workshop has been growing over the years, and the 1997 meeting was perhaps the largest to date, attracting over 50 researchers from the international functional language community, the majority of whom presented papers at the workshop. We are pleased to be able to publish the selection of refereed and revised papers that appear herein.

While the original focus of the workshop was on parallel implementation, it has broadened over time to include compilation, type systems, language issues, benchmarking and profiling, parallelism, language issues, memory management, applications, and some theoretical work. It thus represents a cross-section of the active research community. The papers presented in this volume have been grouped under seven topic headings as follows:

Compilation. In keeping with the theme of the workshop, several papers deal with abstract machines and compilation techniques. Peyton Jones *et al.* identify problems with using C as a compiler target language and suggest an alternative language which they call C--; Holyer and Spiliopoulou present the Brisk Machine, a simplified STG machine that is specifically designed to support multiple paradigms such as computational mobility, dynamic loading and logic programming; Wakeling investigates how compilation to the Java Virtual Machine (JVM) can support the use of Haskell to program embedded processors and provides an interesting comparison between the G-machine and the JVM; Chitil demonstrates that the application of the compile-time optimisation technique of common subexpression elimination to a lazy functional language compiler can lead to an unexpected conclusion; and finally, Scholz presents a technique for producing efficient code from high-level array operations.

Types. Exploitation of type information for compilation continues to be an attractive and fertile area of research. Agat presents a typed intermediate language and a type and effect system which can be used in the register-allocation phase of a compiler; and Mogensen describes a refined program analysis which extends previous work to detect zero uses of an argument, thereby supporting finer compile-time optimisation: for example to avoid updates during garbage collection, or for a limited form of compile-time garbage collection.

Benchmarking and Profiling. In order to ensure acceptable performance from functional programs, it is important for programmers to understand how different data-structuring techniques perform and it is vital for systems developers to provide good benchmarking and profiling techniques. Erwig analyses benchmarking results for two different implementations of purely functional graph data structures tested against three patterns of graph utilisation; Moss and Runci-

man present Auburn, a system tool which uses the concept of a “datatype usage graph” (DUG) to synthesise code for benchmarking different implementations of an Abstract Data Type; and Sparud and Runciman describe how the traditionally resource-hungry technique of recording execution traces (“redex trails”) can be managed parsimoniously — this is dramatically illustrated by their results from tracing a large computation.

Parallelism. Parallel implementation has always been a strong theme of the IFL workshops and this year is no exception. Loidl and Trinder analyse the use of “evaluation strategies” together with system tools such as the GranSim simulator, the instrumented GUM implementation and high-level profiling tools in order to parallelise three large programs; Loidl *et al.* provide an in-depth discussion of the parallelisation of LOLITA, which is claimed to be the largest existing non-strict functional program (at 47 000 lines); Junaidu *et al.* discuss their experience of using GranSim and GUM to parallelise Naira (a compiler for a parallel dialect of Haskell which has itself been parallelised); Chakravarty investigates the benefits of distinguishing remote *tasks* from local *threads*, and of generating tasks and threads lazily, in order to alleviate the problem of idle processors in a message-passing parallel system with high message latencies; Breitinger *et al.* present a distributed-memory parallel system that comprises Eden (which extends Haskell with a coordination language), PEARL (which extends the STG language with message-passing and processes) and DREAM (which extends the STG machine to provide a distributed run-time environment for Eden programs); and Serrarens illustrates how support for multicasting can improve the efficiency of data-parallel programs in Concurrent Clean.

Interaction. Managing events and other forms of user-interaction has long been a *bête-noir* of functional programming systems and at last support for such interaction is maturing. Karlsen and Westmeier provide a uniform framework for event handling that is independent of the source of the event; and Achten and Plasmeijer explain how the new Clean IO model (using new Clean features such as existential types and constructor classes) can support event-driven user interfaces with call-back functions, local state, and message-passing communication.

Language Design. Large-scale and real-world programming is also an important issue for sequential systems. Didrich *et al.* consider two issues related to programming in the large — modularisation and name spaces — and describe the design of the corresponding concepts to support large scale functional programming in the language Opal 2 α ; and Mohnen proposes the use of “extended context patterns” (together with additional typing rules and inference rules) in order to support the highly expressive language feature of context patterns (see LNCS 1268) whilst overcoming the efficiency problems associated with implementing the technique.

Garbage Collection. The final paper presented in this volume addresses the issue of memory management in the programming language Erlang. Given the unusual characteristic of Erlang that all heap pointers point from newer to older objects, Boortz and Sahlin present a compacting garbage collection algorithm

which has zero memory overheads and takes linear time with respect to the size of the data area.

The papers published in this volume were selected using a rigorous *a posteriori* refereeing process from the 34 papers that were presented at the workshop. The reviewing was shared among the program committee, which comprised:

Warren Burton	Simon Fraser University	Canada
Chris Clack	University College London	UK
Tony Davie	University of St. Andrews	UK
Martin Erwig	Fern Universität Hagen	Germany
Pascal Fradet	IRISA/INRIA	France
Kevin Hammond	University of St. Andrews	UK
Pieter Hartel	University of Southampton	UK
Fritz Henglein	University of Copenhagen	Denmark
John Hughes	Chalmers University of Technology	Sweden
Werner Kluge	University of Kiel	Germany
Herbert Kuchen	Westfälische Wilhelms-Universität Münster	Germany
Bruce McKenzie	University of Canterbury	New Zealand
Erik Meijer	University of Utrecht	The Netherlands
John Peterson	Yale University	USA
Rinus Plasmeijer	University of Nijmegen	The Netherlands
Colin Runciman	University of York	UK

The overall balance of the papers is representative, both in scope and technical substance, of the contributions made to the St. Andrews workshop as well as to those that preceded it. Publication in the LNCS series is not only intended to make these contributions more widely known in the computer science community but also to encourage researchers in the field to participate in future workshops, of which the next one will be held in London, UK, September 9–11, 1998 (for more information see <http://www.cs.ucl.ac.uk/staff/if198/>).

April 1998

Chris Clack, Tony Davie, and Kevin Hammond

Table of Contents

Compilation

C--: A Portable Assembly Language	1
<i>Simon Peyton Jones, Thomas Nordin, and Dino Oliva</i>	
The Brisk Machine: A Simplified STG Machine	20
<i>Ian Holyer and Eleni Spiliopoulou</i>	
A Haskell to Java Virtual Machine Code Compiler	39
<i>David Wakeling</i>	
Common Subexpressions Are Uncommon in Lazy Functional Languages	53
<i>Olaf Chitil</i>	
WITH-Loop Folding in SAC - Condensing Consecutive Array Operations	72
<i>Sven-Bodo Scholz</i>	

Types

Types for Register Allocation	92
<i>Johan Agat</i>	
Types for 0, 1 or Many Uses	112
<i>Torben Æ. Mogensen</i>	

Benchmarking and Profiling

Fully Persistent Graphs - Which One To Choose?	123
<i>Martin Erwig</i>	
Auburn: A Kit for Benchmarking Functional Data Structures	141
<i>Graeme E. Moss, and Colin Runciman</i>	
Complete and Partial Redex Trails of Functional Computations	160
<i>Jan Sparud and Colin Runciman</i>	

Parallelism

Engineering Large Parallel Functional Programs	178
<i>Hans-Wolfgang Loidl and Phil Trinder</i>	

Parallelising a Large Functional Program or: Keeping LOLITA Busy	198
<i>Hans-Wolfgang Loidl, Richard Morgan, Phil Trinder, Sanjay Poria, Chris Cooper, Simon Peyton Jones, and Roberto Garigiano</i>	
Naira: A Parallel ² Haskell Compiler	214
<i>Sahalu Junaidu, Antony Davie, and Kevin Hammond</i>	
Lazy Thread and Task Creation in Parallel Graph Reduction	231
<i>Manuel M. T. Chakravarty</i>	
DREAM: The DistRibuted Eden Abstract Machine	250
<i>Silvia Breitter, Ulrike Klusik, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña</i>	
Using Multicasting for Optimising Data-Parallelism	270
<i>Pascal R. Serrarens</i>	
Interaction	
Using Concurrent Haskell to Develop Views over an Active Repository	285
<i>Einar W. Karlsen and Stefan Westmeier</i>	
Interactive Functional Objects in Clean	304
<i>Peter Achten and Rinus Plasmeijer</i>	
Language Design	
Programming in the Large: The Algebraic-Functional Language Opal 2α	322
<i>Klaus Didrich, Wolfgang Grieskamp, Christian Maeder, and Peter Pepper</i>	
Context Patterns, Part II	338
<i>Markus Mohnen</i>	
Garbage Collection	
A Compacting Garbage Collector for Unidirectional Heaps	358
<i>Kent Boortz and Dan Sahlin</i>	
Author Index	375

C--: A Portable Assembly Language

Simon Peyton Jones¹, Thomas Nordin², and Dino Oliva²

¹ Department of Computing Science, University of Glasgow

² Pacific Software Research Centre, Oregon Graduate Institute

Abstract. Of late it has become very common for research compilers to emit C as their target code, relying on a C compiler to generate machine code. In effect, C is being used as a portable compiler target language. It offers a simple and effective way of avoiding the need to re-implement effective register allocation, instruction selection, and instruction scheduling, and so on, all for a variety of target architectures. The trouble is that C was designed as a *programming* language not as a compiler target language, and is not very suitable for the latter purpose. The obvious thing to do is to define a language that *is* designed as a portable target language.

This paper describes C--, a portable compiler target language, or assembler. C-- has to strike a balance between being high-level enough to allow the back end a fair crack of the whip, while being low level enough to give the front end the control it needs. It is not clear that a path exists between these two rocks; the ghost of UNCOL lurks ominously in the shadows [6]. Yet the increasing popularity of C as a compiler target language (despite its unsuitability) suggests strong demand, and provides an existence proof that something useful can be done.

1 Introduction

The author of a new compiler often wants to generate good code, but does not want to duplicate the effort involved in writing a good code generator. One approach is to use C as a portable assembler, relying on a collection of tricks and non-standard extensions to persuade C to generate the sort of code they want. This approach has become quite common among compilers for functional and logic languages [8, 19, 11, 21, 3], including our own compiler for Haskell, the Glasgow Haskell Compiler.

In the light of our experience we have become more and more dissatisfied with using C as a compiler target language. (That is not a criticism of C — it was not designed for that purpose.) For example, a particular difficulty with using C for a garbage-collected language is that it is difficult for the garbage collector to find pointers that are manipulated by C-compiled code. Another is the lack of unrestricted jumps. We elaborate on these difficulties in Section 3.

A possible way to resolve these problems is to design a language that is specifically intended as a compiler target language for garbage-collected languages.

This paper sketches the design for just such a language, **C--**. Despite its name **C--** is by no means a strict subset of C. The name arose from the above-noted popularity of C as a compiler target language, suggesting that a good design approach would be to delete as many features as possible from C, and add as few new features as possible. Whether **C--** is sufficiently better than C to justify the switching costs is an open question — but it is a question that we can only answer by debating a particular concrete design.

The paper gives only an informal overview of **C--**, concentrating on design choices, rather than giving a complete definition.

2 Goals and Non-goals

The goals of **C--** are these:

C-- is a portable assembler, intended particularly for garbage-collected source languages, such as Prolog, Lisp, Smalltalk, ML, Erlang, and Haskell. However, **C--** does not *embody* a particular garbage collector. The garbage-collection field is too active, and the storage management loads imposed by different languages too diverse, to make the collector part of the **C--** implementation. Nor does **C--** force the use of a conservative garbage collector (Section 3.2). Instead, it makes enough information available to support an accurate garbage collector.

C-- generates high-performance code. A conventional assembler does not guarantee good code, but by definition it does not stand in the way of the highest possible performance — it simply exposes the bare machine. That is not true of programming languages in general: the more a language hides the machine behind tractable and portable abstractions, the harder it is to generate really fast code. **C--** sits near the bare-machine end of the spectrum. It is designed to gain a reasonable degree of architecture independence for an extremely modest cost in performance.

C-- exploits existing code generators. The *raison d'être* of **C--** is the desire to exploit the tremendous amount of research and implementation that has been done in code generation technology. The tasks that should be largely or completely done by the code generator are:

- Register allocation, both local and inter-procedural.
- Instruction selection.
- Instruction scheduling.
- Jump elimination and other local optimisations.

In particular, **C--** should support the abstraction of an infinite number of named “registers” — usually called “local variables” in a programming-language context — rather than requiring the front end to map its values onto a finite set (no matter how large). For example, the front end should not have to worry about re-using a register for two different purposes; it can just use two different local variables.

C-- is independent of any particular code generator. Despite all the work on code generation, actual implementations of this technology are surprisingly inaccessible to compiler writers. Usually the back-end implementation is presented in the form of some data types and procedures for building them, rather than as a *language*. (*gcc*'s RTL, XIL [18], and ML-Risc [7] are examples.) This approach forces the back-end user to adopt the language of back-end provider. It also forces commitment to a particular back end.

C-- is intended to be independent of any particular back end, by the simple expedient of being defined as a language with a concrete ASCII syntax, and a semantics independent of any particular implementation. The hope is that a relatively simple parser should suffice to impedance-match C-- to a variety of different back ends.

C-- is inter-operable. It is possible to call C from C-- (and hence C++, COM, and so on), and for a C-- procedure to be called by C. (Perhaps Pascal calling conventions should also be supported; we're not sure.)

On the other hand, when a C-- procedure is calling, or jumping to, another C-- procedure, there is no requirement that standard C parameter passing conventions should be used — it's up to the C-- implementation.

C-- is largely independent of the target architecture. That is, a C-- program contains very little architecture-specific code. It would be ideal if all C-- programs would run unchanged on every architecture. As we will see, that is a very hard goal to meet without sacrificing efficiency, so we accept that the front-end compiler may need to know a few facts about the target architecture (such as its word size).

C-- is human writable, and readable. While most C-- will be written by compilers, some will be written by human beings (such as fragments of run-time system code), and some will be read by compiler writers when debugging their compilers, so C-- should not be impossibly inconvenient for human programmers.

There are some important non-goals too.

C-- is not a distribution format. There are now many machine-independent distribution formats on the market, including *mcode* [14], *ANDF* [4], Java byte codes [9], and *Omnware* [15]. Their war-cry is complete machine-independence, safety, and compactness. They can still be executed rather quickly using just-in-time compilation. Some of these properties come at the price of having to adopt a higher-level execution model than one would want for a compiler target language.

C-- is not a competitor in this market place. Complete machine independence, checkable safety, and compactness are not goals.

C-- is an assembler, not a full programming language. While we expect programmers to write a little C-- code, C-- lacks many features you would expect to find in a language intended for large-scale programming. For example, it has virtually no type system (Section 7).

Apart from C, which we discuss next, the Java Virtual Machine byte codes are C++’s most obvious competitor. Why not compile into JVM, and rely on JVM compilers for execution speed? The difficulty is that the JVM is much too high level. To encode Haskell or ML or Prolog into the JVM can be done, but it requires heroic optimism to believe that the resulting performance will ever be good. For a start, these languages allocate like crazy, and Java implementations are not built for that load; and even if they were, every object carries too much baggage. The JVM is not a portable assembler; it is a compact representation for Java.

3 Why C Is Not Suitable as a Portable Assembler

An obvious question is this: why not (continue to) use C as a portable assembler? Quite a few papers have appeared in the last few years describing ways of achieving this effect [8, 19, 11, 21, 3], but all are unsatisfactory in one way or another. More precisely, we can identify the following difficulties:

- Some source languages require the so-called *tail call optimisation*. This means that a procedure call whose result is returned to the caller of the current procedure is executed in the stack frame of the current procedure. In turn, this allows iteration to be implemented efficiently using recursion. C’s generality makes it very difficult to implement the tail call optimisation, and no C compiler known to us does so across separately compiled modules. This makes it difficult to map source-language procedures onto C procedures (Section 3.1).
- A C compiler is at liberty to lay out its stack frames as it pleases. This makes it difficult for a garbage collector to find the live pointers. In fact, the use of C procedures more or less forces the use of a conservative garbage collector (Section 3.2).
- A C compiler has to be very conservative about the possibility of memory aliasing. This seriously limits the ability of the instruction scheduler to move loads earlier in the instruction stream, perhaps past preceding stores, where there is less chance of the load causing a stall. The front-end compiler often knows that aliasing cannot occur, but there is no way to convey this information to the compiler.
- C lacks the ability to control a number of important low-level features, including returning multiple values in registers from a procedure, mis-aligned memory accesses, arithmetic, data layout, and omitting range checks on multi-way jumps (Section 5).
- Many language implementations require frequent global access to certain variables, such as the stack pointer(s), allocation pointer, environment pointer, and so on. To accommodate this, most implementations rely on non-standard gcc extensions to C that nail specified global variables into particular registers. The wide availability of gcc means that this is not a pressing problem.

3.1 Tail Calls

The ability to make tail calls requires the ability to jump to (rather than call) an arbitrary address. In higher-order languages this address might be fetched from a data structure rather than statically known. In an assembler we take for granted the ability to branch arbitrarily, but not so in C.

Several ways of mapping tail calls onto C have become popular:

1. One can embed all of the program inside a single C procedure, using local labels and `gotos` for control transfer. This does not work with separate compilation, and some C compilers choke on multi-thousand-line procedures.
2. One can treat parameter-less C procedures as extended basic blocks, using a so-called “trampoline” to transfer control between them [20, 21, 11]. The idea is simple: to “jump” to the next basic block the C procedure *returns* the address of the next C procedure. A tiny top-level loop simply calls the returned address:

```
while TRUE { addr = (*addr)(); }
```

This portable trick avoids growing stack, at the cost of a call and return instead of a jump.

3. One can trampoline more sparingly, by calling the next extended basic block (rather than jumping to it), allowing the C stack to grow, and periodically resetting it when it gets too big [2].
4. A gruesome but effective alternative to a trampoline is to post-process the assembly code to remove procedure prologues and epilogues, and use `asm` statements to generate real jumps.
5. `gcc` provides “first-class labels” as a non-standard extension. At first sight these might seem to solve the problem, but there are significant difficulties in practice. Notably, `gcc` says that “totally unpredictable things will happen” if control is transferred to a computed label in a different function body. Not only that, but some separate mechanism must be used to transfer parameters from the jump site to the destination. With considerable care, the Mercury compiler does, nevertheless, use this technique [8].

The bottom line seems to be this. The lack of tail calls is not an insuperable obstacle to using C, especially if `gcc` is used, but to generate efficient code usually leads to a complex and fragile implementation that relies on un-specified aspects of the C compiler.

3.2 Garbage Collection

Our particular interest is in source languages that require a garbage-collected heap. That goal places quite complex requirements on the code generation technology.

One way of classifying garbage collectors is as follows:

- A *conservative* collector treats any word that *looks* as if it is a pointer (e.g. it points into the heap) as if it *were* a pointer. It might actually be an integer that just happened to address the heap, but if so all that happens is that some heap structure is retained that is not actually needed. A conservative collector cannot, therefore, move live objects, because altering the apparent “pointer” to it might instead alter the value of an integer.
- An *accurate* collector accurately follows live pointers. It never treats an integer (say) as a pointer. An accurate collector therefore needs a lot more information than a conservative collector. Using C as a code generator is effectively incompatible with accurate garbage collection, because the C compiler may save heap pointers on the stack, using a layout known only to itself. The only way out is to avoid letting C ever save a live pointer *by never calling a C procedure*. Instead, one uses tail calls exclusively, saving all live pointers explicitly on a separate, explicitly-managed stack.

The main reason that accurate garbage collection is desirable is because it allows compaction. Compaction improves locality, eliminates fragmentation, and allows allocation from a single contiguous area rather than from a free list or lists. This in turn makes allocation much cheaper, amortises the cost of the heap-exhaustion check over multiple allocations (where these occur together), and reduces register pressure (because multiple allocations can all be addressed as offsets from the allocation pointer).

A second reason that accurate garbage collection is desirable is because it does not follow pointers that are dead, even though they are valid pointers. A conservative collector would follow a dead pointer and retain all the structure thereby accessible. Plugging such “space leaks” is sometimes crucially important [10, 1]. Without an accurate garbage collector, the mutator must therefore “zap” dead pointers by overwriting them with zero (or something). Frustratingly, these extra memory stores are usually redundant, since garbage collection probably will not strike before the zapped pointer dies.

However, accurate garbage collection also imposes mutator costs, to maintain the information required by the garbage collector to find all the pointers. Whether accurate garbage collection is cheaper than conservative collection depends on a host of factors, including the details of the implementation and the rate of allocation.

Our goal for C-- is to *permit* but not *require* accurate collection.

4 Possible Back Ends

The whole point of C-- is to take advantage of existing code-generation technology. What, then, are suitable candidates? The most plausible-looking back ends we have found so far are these:

- ML-Risc [7].
- The Very Portable Optimiser (VPO) [5].
- The `gcc` back end, from RTL onwards.

Some of these (e.g. VPO) have input languages whose *form* is architecture independent, but whose details vary from one architecture to another. The impedance-matcher, that reads C-- and stuffs it into the code generator, would therefore need to be somewhat architecture-dependent too.

5 The Main Features of C--

The easiest way to get a flavour of C-- is to look at an example program. Figure 1 gives three different ways of writing a procedure to compute the sum and product of the numbers 1.. N . From it we can see the following design features.

Procedures. C-- provides ordinary procedures. The only unusual feature is that C-- procedures may return multiple results. For example, all the `sp` procedures return two results, the sum and the product. The `return` statement takes zero or more parameters, just like a procedure call; and a call can assign to multiple results as can be seen in the recursive call to `sp1`. This ability to return multiple results is rather useful, and is easily implemented. The number and type of the parameters in a procedure call must match precisely the number and type of parameters in the procedure definition; and similarly for returned results. Unlike C, procedures with a variable number of arguments are not supported.

Types. C-- has a very weak type system whose sole purpose is to tell the code generator how big each value is, and what class of operations can be performed on it. (In particular, floating point values may well be kept in a different bank of registers.) Furthermore, the size of every type is explicit — for example, `word4` is a 4-byte quantity. We discuss the reasons for these choices in Section 7.

Tail calls. C-- guarantees the tail call optimisation, even between separately compiled modules. The procedure `sp2_help`, for example, tail-calls itself to implement a simple loop with no stack growth. The procedure `sp2` tail-calls `sp2_help`, which returns directly to `sp2`'s caller. A tail call can be thought of as “a jump carrying parameters”.

Local variables. C-- allows an arbitrary number of local variables to be declared. The expectation is that local variables are mapped to registers unless there are too many alive at one time to keep them all in registers at once. In `sp3`, for example, the local variables `s` and `p` are almost certainly register-allocated, and never even have stack slots reserved for them.

Labels. C-- provides local labels and `gotos` (see `sp3`, for example). We think of labels and `gotos` simply as a textual description for the control-flow graph of a procedure body. A label is not in scope anywhere except in its own procedure body, nor is it a first class value. A label can be used only as the

target of a `goto`, and only a label can be the target of a `goto`. For all other jumps, tail calls are used.

Conditionals. C-- provides conditional statements, but unlike C there is no boolean type. (In C, `int` doubles as a boolean type.) Instead, conditionals syntactically include a comparison operation, such as “==”, “>”, “>=”, and so on. Like C, C-- also supports a `switch` statement. The only difference is that C-- allows the programmer to specify that the scrutinised value is sure to take one of the specified values, and thus omit range checks.

```
/* sp1, sp2, sp3 all compute the sum 1+2+...+n
                        and the product 1*2*...*n */

/* Ordinary recursion */
sp1( word4 n ) {
    word4 s, p;
    if n == 1 {
        return( 1, 1 );
    } else {
        s, p = sp1( n-1 );
        return( s+n, p*n );
    } }

/* Tail recursion */
sp2( word4 n ) {
    jump sp2_help( n, 1, 1 );
}

sp2_help( word4 n, word4 s, word4 p ) {
    if n==1 {
        return( s, p );
    } else {
        jump sp2_help( n-1, s+n, p*n )
    } }

/* Loops */
sp3( word4 n ) {
    word4 s, p;
    s = 1; p = 1;

    loop:
        if n==1 {
            return( s, p );
        } else {
            s = s+n;
            p = p*n;
            n = n-1;
            goto loop;
        }
}
```

Fig. 1. Three C-- sum-and-product functions

6 Procedures

Most machine architectures (and hence assembler) support a jump instruction, and usually a call instruction. However, these instructions deal simply with control flow; it is up to the programmer to pass parameters in registers, or on the stack, or whatever. In contrast, C-- supports parameterised procedures like most high-level languages. Why?

If C-- had instead provided only an un-parameterised call/return mechanism, the programmer would have to pass parameters to the procedure through explicit registers, and return results the same way. So a call to `sp1` might look something like this, where `R1` and `R2` register names:

```
R1 = 12;      /* Load parameter into R1 */
call sp1;
/* Results returned in R1, R2 */
```

This approach has some serious disadvantages:

- Different code must be generated for machines with many registers than for machines with few registers. (Presumably, some parameters must be passed on the stack in the latter case.) This means that the front end must know how many registers there are, and generate a calling sequence based on this number. Matters are made more complicated by the fact that there are often two sorts of registers (integer and floating point) – this too must be exposed to the front end.
- If the front end is to pass parameters on the stack, responsibility for stack management must be split between the front end and the back end. That is quite difficult to arrange. Is the front end or the back end responsible for saving live variables across a call? Is the front end or the back end responsible for managing the callee-saves registers? Does the call “instruction” push a return address on the stack or into a register? What if the target hardware’s call instruction differs from the convention chosen for C--? And so on. We have found this question to be a real swamp. It is much cleaner for *either* the front end *or* the back end to have complete responsibility for the stack.
- If the mapping between parameters and registers is not explicit, then it may be possible for the code generator to do some inter-procedural register allocation and use a non-standard calling convention. It is premature for the front end to fix the exact calling convention.
- Finally, it seems inconsistent to have an infinite number of named “virtual registers” available as local variables, but have to pass parameters through a fixed set of real registers.

For these reasons, C-- gives complete control of the stack to the back end, and provides parameterised procedures as primitive¹.

¹ Of course, a C-- implementation is free not to use a control stack at all, provided it implements the language semantics; but we will find it convenient to speak as if there were a stack.

However, C--'s procedures are carefully restricted so that they can be called very efficiently:

- The types and order of parameters to a call completely determine the calling convention for a vanilla call. (After inter-procedural register allocation C-- might decide to use non-vanilla calling conventions for some procedures, but that is its business.)
- The actual parameters to a call must match the formal parameters both in number and type. No checks are made. All the information needed to compile a vanilla call is apparent at the call site; the C-- compiler need know nothing about the procedure that is called. There is no provision for passing a variable number of arguments.
- Procedure calls can only be written as separate statements. They cannot be embedded in expressions, like this:

```
x = f(y) + 1;
```

The reason for this restriction is partly that the expression notation makes no sense when `f` returns zero or more than one result, and partly that C-- may not be able to work out the type of the procedure's result. Consider:

```
g( f( x ) );
```

This is illegal, and must instead be written:

```
float8 r;
...
r = f( x );
g( r );
```

Now the type of the intermediate is clear.

6.1 Parameterised Returns

The **return** statement takes as arguments the values to return to the caller. In fact a **return** statement looks just like a call to a special procedure called “**return**”.

At a call site, the number and type of the returned values must match precisely the actual values returned by **return**. For example, if `f` returns an `word4` and a `float8` then every call to `f` must look like:

```
r1, r2 = f( ... );
```

where `r2` is an lvalue of type `word4` and `r2` of type `float8`. It is *not* OK to say

```
f( ... );
```

and hope that the returned parameters are discarded.

6.2 Tail Calls

A tail call is written

```
jump f( ...parameters... )
```

Here, we do not list the return arguments; they will be returned to the current procedure's caller.

No special properties are required of `f`, the destination of the tail call². It does not need to take the same number or type of parameters as the current procedure (the tail call to `sp2_help` in `sp2` in Figure 1 illustrates this); it does not need to be defined in the same compilation unit; indeed, `f` might be dynamically computed, for example by being extracted from a heap data structure.

Tail calls are expected to be cheap. Apart from getting the parameters in the right registers, the cost should be about one jump instruction.

Every control path must end in a `jump` or a `return` statement. There is no implicit `return()` at the end of a procedure. For example, this is illegal:

```
f( word4 x ) {  
    word4 y;  
    y = x+x;  
}
```

7 Data Types

C-- has a very weak type system, recognising only the following types:

- `word1`, `word2`, `word4`, `word8`.
- `float4`, `float8`.

The suffix indicates the size of the type in bytes. This list embodies two major design choices that we discuss next: the paucity of types (Section 7.1), and the explicit size information (Section 7.2).

7.1 Minimal Typing

The main reason that most programming languages have a type system is to detect programming errors, and this remains a valid objective even if the program is being generated by a front-end compiler. However, languages that use dynamic

² This situation contrasts rather with C, where `gcc`'s manual says of the `-mtail-call` flag: "Do (or do not) make additional attempts (beyond those of the machine-independent portions of the compiler) to optimise tail-recursive calls into branches. You may not want to do this because the detection of cases where this is not valid is not totally complete."

allocation are frequently going to say “I know that this pointer points to a structure of this shape”, and there is no way the C-- implementation is going to be able to verify such a claim. Its truth depends on the abstractions of the original source language, which lie nakedly exposed in its compiled form. Paradoxically, the lower level the language the more sophisticated the type system required to gain true type security, so adding a secure type system to an assembler is very much a research questions [17, 16].

So, like a conventional assembler, C-- goes to the other extreme, abandoning any attempt at providing type security. Instead, types are used only to tell the code generator the information it absolutely has to know, namely:

- The kind of hardware resource needed to hold the value. In particular, floating point values are often held in different registers than other values.
- The size of the value.

For example, signed and unsigned numbers are not distinguished. Instead, like any other assembler, it is the *operations* that are typed. Thus, “+” adds signed integers, while “+u” adds unsigned integers, but the operands are simply of type **word1**, **word2** etc. (The size of the operation is, however, implicit in the operand type.)

Similarly, C-- does not have a type corresponding to C’s “*” pointer types. In C, **int*** is the type of a pointer to **int**. In C-- this type is just **word4** (or **word8**). When doing memory loads or stores, the size of the value to be loaded or stored must therefore be given explicitly. For example, to increment the integer in memory location **foo** we would write:

```
word4[foo] = word4[foo] + 1;
```

The addressing mode **word4[foo]** is interpreted as a memory load or store instruction, depending on whether it appears on the left or right of an assignment.

7.2 Type Sizes

Since different machines have different natural word sizes, it is tempting to suggest that C-- should abstract away from word-size issues. That way, an unchanged C-- program could run on a variety of machines. C does this: an **int** is 32 bits wide on some machines and 64 bits on others.

While this is somewhat attractive in a programming language, it seems less appropriate for an assembler. The front-end compiler may have to generate huge offset-calculation expressions. Suppose the front-end compiler is computing the offset of a field in a dynamically-allocated data structure containing two floats, a code pointer, and two integers. Since it does not know the actual size of any of these, it has to emit an offset expression looking something like this:

```
2*sizeof(float) + sizeof(codepointer) + 2*sizeof(int)
```

Apart from the inconvenience of generating such expressions (and implementing arithmetic over them in the front-end compiler) they produce a substantial increase in the size of the C-- program.

Another difficulty is that the front-end compiler cannot calculate the alignment of fields within a structure based on the alignment of the start of the structure.

One way of mitigating these problems would be to introduce **struct** types, as in C; offsets within them would then be generated by field selectors. However, this complicates the language, and in a Haskell or ML implementation there might have to be a separate **struct** declaration for each heap-allocated object in the program. Offset calculations using **sizeof** would still be required when several heap objects were allocated contiguously. And so on.

C-- instead takes a very simple, concrete approach: *each data type has a fixed language-specified size*. So, for example, **word4** is a 4-byte word, while **word8** is an 8-byte word.

This decision makes life easy for the C-- implementation, at the cost of having to tell the front-end compiler what C-- data types to use for the front end's various purposes. In practice this seems unlikely to cause difficulties, and simplifies some aspects (such as arithmetic on offsets).

8 Memory

8.1 Static Allocation

C-- supports rather detailed control of static memory layout, in very much the way that ordinary assemblers do. A data block consists of a sequence of: labels, initialised data values, uninitialised arrays, and alignment directives. For example:

```
data {
  foo:  word4{10};          /* One word4 initialised to 10 */
        word4{1,2,3,4};    /* Four initialised word4's */
        word4[80];         /* Uninitialised array of 80 word4's */
  baz1:
  baz2: word1               /* An uninitialised byte */
  end:
}
```

Here **foo** is the address of the first **word4**, **baz1** and **baz2** are both the address of the **word1**, and **end** is the address of the byte after the **word1**.

*All the labels have type **word4** or **word8**, depending on the particular architecture.* On a Sparc, **foo**, **baz1**, **baz2** and **end** all have type **word4**. You should think of **foo** as a pointer, not as a memory location.

Unlike C, there is no implicit alignment nor padding. Furthermore, the address relationships between the data items in a data block is guaranteed; for example, `baz1 = foo + 340`.

8.2 Dynamic Allocation

The C-- implementation has complete control over the system stack; for example, the system stack pointer is not visible to the C-- programmer. However, sometimes the C-- programmer may want to allocate chunks of memory (not virtual registers) on the system stack. For this, the `stack` declaration is provided:

```
f( word4 x ) {
    word4 y;
    stack {
        p : word4;
        q : float8;
        r : word4[30];
    }
    ...
}
```

Just as with static allocation, `p`, `q`, and `r` all have type `word4` (or `word8` on 64-bit architectures). Their address relationship is guaranteed, regardless of the direction in which the system stack grows; for example, `q = p+4`.

`stack` declarations can only occur inside procedure bodies. The block is preserved across C-- calls, and released at a `return` or a *tail call*. (It is possible that one might want to allocate a block that survives tail calls, but we have not yet found a reasonable design that accommodates this.)

8.3 Alignment

Getting alignment right is very important. C-- provides the following support.

- Simple alignment operations, `aligni`, are provided for static memory allocation. For example, the top-level statements

```
foo:  word4;
      align8;
baz:  float8;
```

ensures that `baz` is the address of a 8-byte aligned 8-byte memory location. There is no implicit alignment at all.

- Straightforward memory loads and stores are assumed aligned to the size of the type to be loaded. For example, given the addressing mode `float8[ptr]`, C-- will assume that `ptr` is 8-byte aligned.

Sometimes, however, memory accesses may be mis-aligned, or over-aligned:

- When storing an 8-byte float in a dynamically allocated object, it may be convenient to store it on a 4-byte boundary, the natural unit of allocation.
- Byte-coded interpreters most conveniently store immediate constants on byte boundaries.
- Sometimes one might make a byte load from pointer that is known to be 4-byte aligned, for example when testing the tag of a heap-allocated object. Word-oriented processors, such as the Alpha, can perform byte accesses much more efficiently if they are on a word boundary, so this information is really worth conveying to the code generator.

Coding up mis-aligned accesses in a language that does not support them directly is terribly inefficient (lots of byte loads and shifts). This is frustrating, because many processors support mis-aligned access reasonably well. For example, loading a 8-byte float with two 4-byte loads is easy on a Sparc and MIPS R3000; the Intel x86 and PowerPC support pretty much any unaligned access; and the Alpha has canned sequences using the `LDQ_U` and `EXTxx` instructions that do unaligned accesses reasonably efficiently.

For these reasons, C-- also supports explicitly-flagged mis-aligned or over-aligned accesses.

- The load/store addressing mode is generalised to support an alignment assumption: `type{align}[ptr]`. For example:
 - `float8{align4}[ptr]` does a 8-byte load, but only assumes that `ptr` is aligned to a 4 byte boundary.
 - `word4{align1}[ptr]` does a 4-byte load from a byte pointer.
 - `word1{align4}[ptr]` does a 1-byte load from a pointer that is 4-byte aligned.

8.4 Endian-Ness

There is no support varying endian-ness. (No language provides such support. At best, the type system prevents one reading a 4-byte integer one byte at a time, and hence discovering the endian-ness of the machine, but that would be inappropriate for an assembler.)

8.5 Aliasing

It is often the case that the programmer knows that two pointers cannot address the same location or, even stronger, no indexed load from one pointer will access the same location as an indexed load from the other.

The `noalias` directive supports this:

```
noalias x y;
```

is a directive that specifies that no memory load or store of the form *type* [*x op e*] can conflict with a load or store of similar form involving *y*. Here *op* is + or -.

9 Garbage Collection

How should C-- support garbage collection?

One possibility would be to offer a storage manager, including a garbage collector, as part of the C-- language. The trouble is that every source language needs a different set of heap-allocated data types, and places a different load on the storage manager. Few language implementors would be happy with being committed to one particular storage manager. We certainly would not.

Another possibility would be to provide no support at all. The C-- user would then have the choice of using a conservative collector, or of eschewing C-- procedures altogether and instead using tail calls exclusively (thereby avoiding having C-- store any pointers on the stack). This alternative has the attraction of clarity, and of technical feasibility, but then C-- would be very little better than C.

The third possibility is to require C-- to keep track of where the heap pointers are that it has squirreled away in its stack, and tell the garbage collector about them when asked. A possible concrete design is this. The garbage collector calls a C procedure `FindRoots(mark)`, passing a C procedure `mark`. `FindRoots` finds all the heap pointers in the C-- stack, and calls `mark` on each of them. Which of the words stored in the stack are reported as heap pointers by `FindRoots`? The obvious suggestion is to have a new C-- type, `gcptr`, which identifies such pointers.

This is easily said, but not quite so easily done. How might C-- keep track of where the `gcptr` values are in the stack? The standard technique is to associate a stack descriptor (a bit pattern, for example) with each return address pushed on the stack. After all, the code at the return address "knows" the layout of the stack frame, and it is only a question of making this information explicitly available. One can associate the descriptor with the code either by placing it just before the code, or by having a hash table that maps the code address to the descriptor information.

The trouble with this approach is that it is somewhat specific to a particular form of garbage collection. For example, what if there is more than one kind of heap pointer that should be treated separately by the collector? Or, what if the pointer-hood of one procedure parameter can depend on the value of another, as is the case for the TIL compiler [22]. We are instead developing other ideas that provide much more general support for garbage collection, and also provide support for debuggers and exception handlers, using a single unified mechanism [13].

10 Other Features of C--

There are several features of C-- that we have not touched on so far:

- Global registers.
- Arithmetic operations and conversions between data types.
- Interfacing to C.
- Separate compilation.

They are described in the language manual [12].

11 Status and Future Directions

C-- is at a very early stage. We have two implementations of a core of the language, one using VPO and one using ML-Risc. Each was built in a matter of weeks, rather than months, much of which was spent understanding the code generator rather than building the compiler. These implementations do not, however, cover the whole of C-- and are far from robust.

The support for tail calls raises an interesting question: does it make sense for the implementation to use callee-saves registers, as do most implementations of conventional languages? The trouble is that the callee-saves registers must be restored just before a tail call, and then perhaps immediately saved again by the destination procedure. Since each procedure decides independently which callee-saves registers it needs to save it is not at all obvious how to avoid these multiple saves of the same register. Perhaps callee-saves registers do not make sense if tail calls are common. Or perhaps some inter procedural analysis might help.

As modern architectures become increasingly limited by memory latency, one of C--'s biggest advantages should be its ability to provide detailed guidance about possible aliasing to the code generator, and thereby allow much more flexibility in instruction scheduling. We have not yet implemented or tested such facilities.

Many garbage-collected languages are also concurrent, so our next goal is to work out some minimal extensions to C-- to support concurrency. We have in mind that there may be thousands of very light-weight threads, each with a heap-allocated stack. This means that C-- cannot assume that the stack on which it is currently executing is of unbounded size; instead it must generate stack-overflow checks and take some appropriate action when the stack does overflow, supported by the language-specific runtime system.

Acknowledgements

This document has benefited from insightful suggestions from Chris Fraser, Dave Hanson, Fergus Henderson, Xavier Leroy, Norman Ramsay, John Reppy, and the IFL referees. We are very grateful to them.

References

- [1] AW Appel. *Compiling with continuations*. Cambridge University Press, 1992.
- [2] H Baker. Cons should not cons its arguments, Part II: Cheney on the MTA. *SIGPLAN Notices*, 30(9):17–20, Sept 1995.
- [3] JF Bartlett. SCHEME to C: a portable Scheme-to-C compiler. Technical Report RR 89/1, DEC WRL, 1989.
- [4] ME Benitez, P Chan, JW Davidson, AM Holler, S Meloy, and V Santhanam. ANDF: Finally an UNCOL after 30 years. Technical Report TR-91-05, University of Virginia, Department of Computer Science, Charlottesville, VA, March 1989.
- [5] ME Benitez and JW Davidson. The advantages of machine-Dependent global optimization. In *International Conference on Programming Languages and Architectures (PLSA'94)*, pages 105–123, 1994.
- [6] ME Conway. Proposal for an UNCOL. *Communications of the ACM*, 1(10):5–8, October 1958.
- [7] L George. MLRISC: Customizable and Reusable Code Generators. Technical report, Bell Laboratories, Murray Hill, 1997.
- [8] Fergus Henderson, Thomas Conway, and Zoltan Somogyi. Compiling logic programs to C using GNU C as a portable assembler. In *Postconference Workshop on Sequential Implementation Technologies for Logic Programming (ILPS'95)*, pages 1–15, Portland, Or, 1995.
- [9] H McGilton J Gosling. The Java Language Environment: a White Paper. Technical report, Sun Microsystems, 1996.
- [10] R Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–80, Jan 1992.
- [11] Simon L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.
- [12] SL Peyton Jones, T Nordin, and D Oliva. The C-- manual. Technical report, Department of Computing Science, University of Glasgow, 1998.
- [13] SL Peyton Jones, N Ramsey, and JW Davidson. Portable support for garbage collection, debugging, and exceptions. Technical report, In preparation, Department of Computing Science, University of Glasgow, 1998.
- [14] Brian T Lewis, L Peter Deutsch, and Theodore C Goldstein. Clarity MCode: A re-targetable intermediate representation for compilation. *ACM SIGPLAN Notices*, 30(3):119–128, March 1995.
- [15] S Lucco, O Sharp, and R Wahbe. Omniware: a universal substrate for Web programming. In *Fourth International World Wide Web Conference, Boston: The Web Revolution*, Dec 1995.
- [16] G Morrisett, K Crary, N Glew, and D Walker. Stack-based typed assembly language. In *Proc Types in Compilation, Osaka, Japan*, March 1998.
- [17] G Morrisett, D Walker, K Crary, and N Glew. From system f to typed assembly language. In *Proc 25th ACM Symposium on Principles of Programming Languages, San Diego*, Jan 1998.
- [18] Kevin O'Brien, Kathryn M. O'Brien, Martin Hopkins, Arvin Shepherd, and Ron Unrau. XIL and YIL: The intermediate languages of TOBEY. *ACM SIGPLAN Notices*, 30(3):71–82, March 1995.
- [19] M Pettersson. Simulating tail calls in C. Technical report, Department of Computer Science, Linköping University, 1995.

- [20] GL Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT Lab for Computer Science, 1978.
- [21] D Tarditi, A Acharya, and P Lee. No assembly required: compiling Standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, 1992.
- [22] D Tarditi, G Morrisett, P Cheng, C Stone, R Harper, and P Lee. Til: A type-directed optimizing compiler for ml. In *Proc SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'96)*, Philadelphia. ACM, May 1996.

The Brisk Machine: A Simplified STG Machine

Ian Holyer and Eleni Spiliopoulou

Department of Computer Science, University of Bristol
Merchants Venturers Building, Woodland Road,
Bristol BS8 1UB, United Kingdom
`{ian,spilio}@cs.bris.ac.uk`

Abstract. This work presents the Brisk Machine, a machine model for the implementation of functional languages. It is especially designed to be flexible and dynamic, so that it can support a uniform and efficient implementation of multiple paradigms such as computational mobility, dynamic loading and linking, and logic programming.

The Brisk Machine is based on the STG Machine, though its model is simplified and adapted so that the various paradigms it supports can be accommodated easily without interference between them.

1 Introduction

Many different machine models have been proposed as intermediate forms in the compilation of functional languages, to act as the target of the front end of compilers and as a starting point for code generation. The resulting models tend to be complicated though, partly due to the effort to optimise them in a particular setting.

The Brisk¹ Machine has been designed as a target for Haskell compilation which is flexible enough to support a number of different run-time execution models. In particular, we aim to support concurrency, distribution and computational mobility, dynamic loading and linking, debugging tools, and logic programming in the form of the Escher [11] extensions to Haskell.

The Brisk Machine is heavily based on the STG Machine. Although the underlying model of the STG Machine is simple, design decisions made because of efficiency considerations radically restrict its flexibility and extensibility. To achieve a flexible machine model, and allow extensions to be built without interfering with the optimisations, many issues concerning the optimisation of execution are relegated to special built-in functions rather than being built into the model itself. This leaves a lean and adaptable basic execution model. Furthermore, the Brisk Machine provides a dynamic model, where functions are stored as heap nodes. This helps support distributed execution and mobility of computation where code is communicated dynamically between machines, or the development of an environment where newly compiled code is loaded dynamically into the running environment. A uniform representation of heap nodes,

¹ **Brisk** stands for Bristol Haskell [2] compiler

$prog \rightarrow decl_1 ; \dots ; decl_n$	$n \geq 1$
$decl \rightarrow fun\ arg_1 \dots arg_n = expr$	$n \geq 0$
$expr \rightarrow let\ bind_1 ; \dots ; bind_n\ in\ expr$	$n \geq 1$
$case\ var\ of\ alt_1 ; \dots ; alt_n$	$n \geq 1$
$appl$	
$bind \rightarrow var = appl$	
$alt \rightarrow con\ arg_1 \dots arg_n \rightarrow expr$	$n \geq 0$
$_ \rightarrow expr$	
$appl \rightarrow fun\ arg_1 \dots arg_n$	$n \geq 0$
$fun \rightarrow var$	
$con \rightarrow var$	
$arg \rightarrow var$	

Fig. 1. The Syntax of the Brisk Kernel Language

including those which represent functions, allows different execution strategies to coexist, e.g. compiled code, interpretive code, and code with logic programming extensions.

The compiler's intermediate language, the Brisk Kernel Language, is lower level than other functional intermediate languages. This allows the compiler to perform more low level transformations. A number of extensions and optimisations can be added easily, allowing for the efficient implementation of various computational paradigms.

There are three main sections in this paper. The first one analyses the Brisk Kernel Language, a minimal intermediate language, the second one describes the representations used in the run-time system, and the third explains the abstract machine model and its instruction set.

2 The Brisk Kernel Language

The Brisk compiler translates a source program into a minimal functional language called the *Brisk Kernel Language*, or BKL, which is described in detail in [7]. This is similar to functional intermediate languages used in other compilers, e.g. the STG language [13]. However, it is lower level than most others, allowing more work to be done in the compiler as source-to-source transformations, and thus making the run-time system simpler. Its syntax is shown in Figure 1. All mention of types, **data** statements etc. have been omitted for simplicity.

BKL provides expressions which are simple versions of *let* constructs, *case* expressions and function applications. Operationally, a *let* construct causes suspension nodes representing subexpressions to be added to the heap, a *case* expression corresponds to a simple switch since its argument is assumed to be evaluated in advance, and a function application corresponds to a tail call.

Many complications, including issues to do with evaluation and sharing, are apparently missing from the above picture. These details are hidden away in

a collection of built-in functions. This simplifies both BKL and the run-time system, and allows different approaches to evaluation and sharing to be chosen at different times by replacing these functions. The built-in functions are described briefly in section 2.1 and in more detail in [7].

In order to simplify the run-time system, and to allow a uniform representation for expressions, BKL has the following features:

1. A *case* expression represents an immediate switch; it does not handle the evaluation of its argument. The argument must represent a node which is guaranteed to be already in head normal form.
2. In BKL there is no distinction between different kinds of functions such as defined functions, constructor functions and primitive functions. All functions are assumed here to be represented in a uniform way.
3. Every function application must be saturated. That is, every function has a known arity determined from its definition, and in every call to it, it is applied to the right number of arguments.
4. In every application, whether in an expression or on the right hand side of a local definition, the function must be in evaluated form. This allows a heap node to be built in which the node representing the function acts as an info node for the application.
5. Local functions are lifted out, so that local *let* definitions define simple variables, not functions. This is described further in 2.1.

Evaluation of expressions is not assumed to be implicitly triggered by the use of *case* expressions or primitive functions. Instead (points 1, 2) the evaluation of *case* expressions or arguments to strict primitive functions is expressed explicitly using calls to built-in functions such as the **strict** family, as discussed below.

The arity restriction (point 3) means that partial applications are also dealt with explicitly by the compiler during translation into BKL, using built-in functions. This frees the run-time system from checking whether the right number of arguments are provided for each function call, and from building partial applications implicitly at run-time.

The saturation of calls, and the requirement that functions are in evaluated form before being called (point 4), allows for a uniform representation of expressions and functions in the run-time system. Every expression can be represented as a node in the form of a function call where the first word points to a function node. Arity information in the function node is always available to describe the call node. Function nodes themselves can all be represented uniformly (point 2), and in fact follow the same layout as call nodes, being built from constructors as if they were data.

2.1 Built-in Functions

The main contribution of the Brisk Machine is that it makes the run-time system match the graph reduction model more closely, by freeing it from many of the usual execution details. In order to achieve this, several issues that arise during

execution are handled via a collection of built-in functions. Built-in functions not only handle constructors and primitive functions, but also handle issues concerning evaluation and sharing. As a result, the run-time system becomes simple and flexible. Furthermore, several extensions regarding computational mobility as well as logic programming are treated as built-ins but we do not discuss these issues here. In the next sections the role of built-in functions in evaluation and sharing is discussed.

Partial Applications Every function in BKL has an arity indicating how many arguments it expects. Every call must be saturated, i.e. the correct number of arguments must always be supplied, according to the arity. This means that the compiler transformations must introduce explicit partial applications where necessary. For example, given that `(+)` has arity two, then the expression `(+) x` is translated into the Brisk Kernel Language as a partial application, using the built-in function `pap1of2`:

```
(+) x --> pap1of2 (+) x
```

The function `pap1of2` takes a function of arity two and its first argument and returns a function of arity one which expects the second argument. The function `pap1of2` is one of a family dealing with each possible number of arguments supplied and expected respectively.

The compiler may also need to deal with over-applications. For example, if `head`, which has arity one, is applied to a list `fs` of functions to extract the first, and the result applied to two more arguments `x` and `y`, then the over-application `head fs x y` would need to be translated:

```
head fs x y --> let f = head fs in strict100 f x y
```

Since `f` is an unevaluated expression, a member of the `strict` family of functions described below is used to evaluate it before applying it.

The arity restriction applies to higher order functions as well. For example, if the compiler determines that the function `map` has arity two, and that its first argument is a function of arity one, then a call `map (+) xs` must be transformed, e.g. to `let f x = pap1of2 (+) x in map f xs`.

The compiler has some choice in determining arities when translating a source program into BKL. For example, given a definition of the *compose* operator:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
(.) f g = let h x = f (g x) in h
```

the compiler may choose to implement the definition as it stands, giving the operator arity two, or to transform it by adding an extra argument so that it has arity three. It may even be appropriate to compile two versions of the operator for use with different arities.

In the rest of this paper, the arity information is carried in the type, with brackets to indicate the number of arguments expected. The number of unbracketed arrows \rightarrow forms the arity of the function. Thus, if the *compose* operator is compiled with arity two, the type signature above will be used, but if it is compiled with arity three, it would be written $(a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow c \rightarrow b$.

The arity restrictions we have described mean that functions are evaluated by calling and returning, as with other values. Other compilers include optimisations which avoid this, implementing a call to a possibly unevaluated function simply as a jump, but the approach we have taken has compensating advantages. It allows for greater uniformity and simplicity in the run-time system; every application can be represented in a direct way, without the need for argument stacks or for testing to see whether enough arguments are available. This argument testing is normally needed to check for partial applications; in Brisk, all partial application issues are dealt with at compile-time.

Evaluation and Strictness In BKL, it is assumed that the compiler has already dealt with the issue of forcing the evaluation of subexpressions, via built-in functions. Thus various subexpressions such as the argument to the *case* construct are assumed to be already in evaluated form.

Moreover, in any function application such as $f\ x\ y$, the function f is assumed to be in an evaluated form before being called, and not to be represented as a suspension node. This enables the expression $f\ x\ y$ to be represented in the heap as a 3-word node; the first word points to the function node representing f . The function node provides layout information for the application node, as well as information on how to evaluate it, so f must be in evaluated form.

To express this requirement more formally we introduce the notion of *strict type* $!t$ for each type t to indicate that a variable or a function is in head normal form. For example, in the type signature

$$f :: !a \rightarrow b \rightarrow c$$

the function f expects two arguments, the first of which has to be in evaluated form as indicated by the $!$ symbol. As another example, given:

$$g :: !(a \rightarrow b) \rightarrow c$$

the function g takes as its argument a function which must be in evaluated form.

To force explicit evaluation of expressions, a built-in family of strict functions is provided. For example, `strict01 f x` forces evaluation of x before calling f . In the family, the 0s and 1s attached to the name `strict` refer to the function and each of its arguments; a 1 indicates that the respective item needs to be evaluated before the function is called, and 0 means that no evaluation is required. There is one member of the family for each possible combination of requirements.

The expression `strict01 f x` indicates that in the call $f\ x$, f is assumed to be in evaluated form but x has to be evaluated before executing the call $f\ x$. The type signature of `strict01` is:


```
strict01 :: !(!(a -> b) -> a -> b)
strict01 f x = ...
```

where $!a \rightarrow b$ indicates that f expects x to be in evaluated form, $!(a \rightarrow b)$ indicates that f must be evaluated before being passed to `strict01` and the outermost `!` indicates that `strict01` itself is assumed to be in evaluated form. An example which illustrates the use of the `strict` family of functions is the definition of `(+)`:

```
(+) :: !(Int -> Int -> Int)
x + y = strict011 (!+) x y

(!+) :: !(Int -> Int -> Int)
x !+ y = ...
```

Here `strict011` indicates that both x and y need to be evaluated before making the call `(!+) x y`. The `(!+)` function is a built-in one which assumes that its arguments are already evaluated.

To explain the operation of the `strict` functions, take `strict011 (!+) x y` as an example. Operationally `strict011` places a pointer to the original expression `strict011 (!+) x y` on a return stack with an indication that x is under evaluation (see section 3.2), and makes x the current node. When x is evaluated to x' (say), the expression is updated in place to become `strict001 (!+) x' y` with an indication that y is under evaluation, and y is made the current node. When y is evaluated to y' , the expression is popped off the stack and updated to `(!+) x' y'` which becomes the current node.

An alternative approach which allows a little more optimisation is to treat the `strict` functions as having arity one. Then `(+)` would be defined by `(+) = strict011 (!+)` and the compiler has the opportunity to generate code for `(+)` directly, inlining the call to `strict011`.

One benefit of using the `strict` family of functions to describe evaluation, and of using built-in functions generally, is that they can be replaced by alternative versions which represent different evaluation strategies (e.g. for mobility issues or in logic programming extensions to the language).

The notation for strict types that we have introduced here can be extended to deal with unboxing [14], by associating a suitable representation with each strict type. The issue of unboxing in the compiler and the Brisk Kernel Language are not discussed further here. However, the run-time system and abstract machine described later allow for unboxed representations.

Sharing One way to implement sharing is to introduce a built-in function `share` which ensures that a subexpression is evaluated only once. The call `share x` is equivalent to x , but when it is first accessed an update frame is put on the return stack while x is evaluated. When the evaluation of x finishes, the update frame overwrites the call `share x` with an indirection to the result. An example which illustrates this approach to sharing is the `map` function with definition:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Translating this into the Brisk Kernel Language gives:

```
map f xs = strict001 map' f xs

map' f xs = case xs of
  [] -> []
  (:) y ys ->
    let z = share z'
        z' = strict10 f y
        zs = share zs'
        zs' = map f ys
    in (:) z zs
```

The definition of `map` uses a member of the `strict` family to specify that the second argument to `map` needs to be evaluated before pattern matching can occur. In `map'`, the variable `z` is defined as a shared version of `z'` and `z'` is defined as `strict10 f y` because `f` is a dynamically passed function, which may not be in evaluated form. The variable `z'` is not mentioned anywhere else, so that the reference to it from `z` is the only reference. All other references point to the “share node” `z`. When `z` is evaluated, the built-in `share` function arranges for `z'` to be evaluated, and for the share node to be updated to become an indirection to the result. Thus all references now share the update.

This approach to sharing has a high space overhead. In the absence of sharing analysis, every node representing an unevaluated expression, i.e. every application of a function which is not a constructor, has to be assumed to need sharing, and for every such node a 2-word share node is added. An alternative optimised approach to sharing is to build it into the return mechanism. Every time the current node becomes evaluated, the return mechanism can check whether the result is different from the original expression node and, if so, overwrite the original expression node with an indirection. The overhead of this test on every return is offset by the fact that far fewer update frames need to be built – they are only needed to cover certain special cases.

Lifting Lifting is an important issue in the Brisk compiler, since there are no local function definitions in BKL; local definitions represent values not functions. In Brisk, it is possible to support either the traditional approach to lifting, or the approach taken in the Glasgow compiler; we describe both briefly here.

In the traditional approach to lifting [8], the free variables (or maximal free subexpressions) of a local function are added as extra arguments, so that the function can be lifted to the top level. The local function is then replaced by a partial application of the lifted version. For example, given a local definition of a function `f`:

```
... let f x y = ... g ... h ... in e
```

two extra arguments g and h are added and the definition is lifted out to the top level as f' :

```
... let f = pap2of4 f' g h in e

f' g h x y = ... g ... h ...
```

An explicit partial application is built using `pap2of4` in order to preserve the arity restrictions of BKL.

Peyton Jones & Lester [10] describe a slightly different approach to lifting. For a local function such as f , a node is built locally to represent f and this node contains references to the free variables of f . The code for f can be lifted out to the top level; it obtains the free variables from the node representing f and obtains the other arguments in the normal way.

To express this form of lifting in BKL, the language can be extended so that free variables are represented explicitly using a tuple-like notation. For example, the lifted version of f above becomes:

```
... let f = f' (g, h) in e

f' (g,h) x y = ... g ... h ...
```

The local definition of f represents the idea that f is built directly as a function node, using f' as its code and including references to g and h . The definition of f' indicates that self-contained code should be generated which gets g and h from the function node and x and y from the call node, as shown in Figure 2.

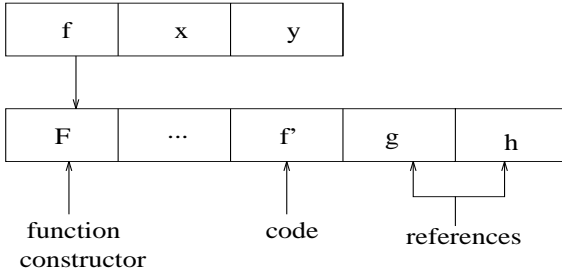


Fig. 2. A Function with References and Arguments

This approach to lifting fits in well with the way in which global functions are treated in Brisk. In order to support dynamic movement of code, a global function is represented as a node in the heap containing explicit references to the other global values which it uses. Thus a global definition:

```
f x y = ... g ... h ...
```

can also be represented in the extended notation as:

$$f = f' (g, h)$$

$$f' (g, h) \times y = \dots g \dots h \dots$$

indicating that f is represented as a node referencing g and h , and that self-contained code is generated for f' .

3 The Brisk Run-Time System

In this section we describe the way in which programs are represented in Brisk. The representation is based heavily on the one used for the STG Machine. However, it is simplified and made more uniform so that the link with conventional simple graph reduction is as clear as possible, making it easier to adapt the run-time system to alternative uses. At the same time, enough flexibility is retained to allow most of the usual optimisations to be included in some form.

The state of a program consists of a heap to hold the graph, a current node pointer to represent the subexpression which is currently being evaluated, and a return stack which describes the path from the root node down to the current node. There are no argument or update stacks.

3.1 The Heap

The heap contains a collection of nodes which holds the program graph. Every node in the heap can be thought of as a function call. A spineless representation is used; nodes vary in size, with the number of arguments being determined by the arity of the function. For example, an expression $f \ x \ y \ z$ is represented as a 4-word node in which the first word points to a node representing f and the other words point to nodes representing x , y and z . There is a single *root node* representing the current state of the program as a whole, and all active nodes are accessible from it. Free space is obtained at the end of the heap, and a copying or compacting garbage collector is used to reclaim dead nodes.

In general, functions may have both unboxed and boxed arguments, with the unboxed ones preceding the boxed ones, and the arity of a function reflects how many of each. A node may thus in general consist of a function pointer followed by a number of raw words, followed by a number of pointer words. Constructors with unboxed arguments can be used to represent raw data. A large contiguous data structure such as array can be stored in a single node of any size; a suitable constructor node can be generated dynamically to act as the function pointer.

A node representing a function contains information about the arity of the function, the code for evaluating the function, code for other purposes such as garbage collection or debugging, and references to other nodes needed by the evaluation code, i.e. the free variables. The arity information allows the function node to be used as an *info node*, i.e. one which describes the size and layout of

call nodes. To ensure that the size and layout information is always available, the function pointer in a call node must always refer to an evaluated function node, and not to a suspension which may later evaluate to a function.

This uniform representation of nodes means that any heap node can be viewed in three ways. First, a node represents an expression in the form of a function call, as in figure 3:

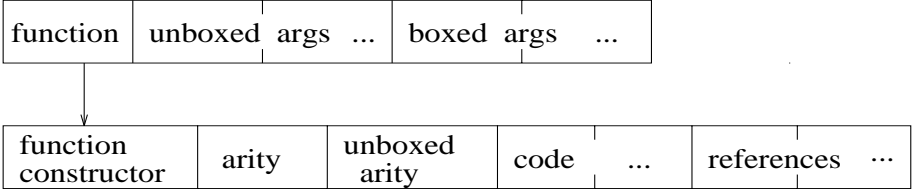


Fig. 3. A node as a function call

Second, a node can be treated as a data structure which can be manipulated, e.g. by the garbage collector or by debugging tools, in a uniform way, as in figure 4:

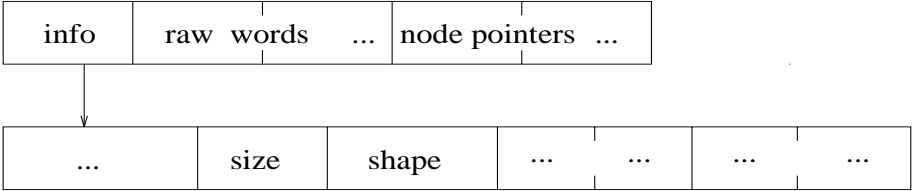


Fig. 4. A node as a uniform data structure

Third, a node can be regarded as an active object, responsible for its own execution, with methods for evaluation and other purposes available via its info pointer, as in figure 5.

Function nodes follow the same pattern. The function pointer at the beginning of a function node can be regarded as a constructor, with functions being thought of as represented using a normal data type, hidden from the programmer by an abstraction. Such a function constructor has a number of unboxed arguments representing arity and code information, and a number of boxed arguments representing references to other nodes. All functions, including global ones, are represented as heap nodes which contain references to each other. New info nodes can be created dynamically to cope with special situations, newly compiled code can be dynamically loaded, and code can be passed from one process to another, which provides greater flexibility. This contrasts with other

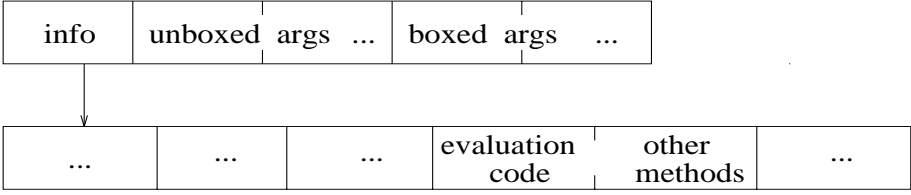


Fig. 5. A node as an object

systems [5], [13], where info structures are stored statically, which would prevent such dynamic movement of nodes.

It is becoming common, e.g. in Java [6], to use an architecture independent bytecode representation of program code, which makes it portable. This can be used to ship programs over a network and interpret them remotely. The same technique is used in Brisk. However, statically compiled code is also supported. To allow both types of code to be mixed in the same heap, the code word in a function node points to a static entry point. In the case of statically compiled code, this is precisely the evaluation code for the function. In the case of bytecode, the code word points to the static entry point of the interpretive code, and the function node contains an extra reference to a data node containing the bytecode.

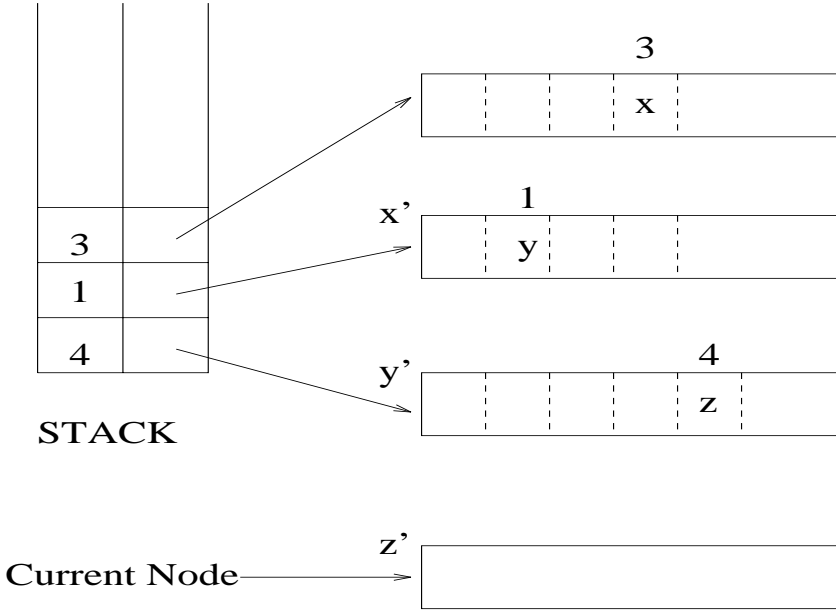
3.2 The Return Stack

The *return stack* in the Brisk run-time system keeps track of the current evaluation point. Each stack entry consists of a pointer to a node in the heap representing a suspended call, and the position within that node of an argument which needs to be evaluated before execution of the call can continue, as in figure 6.

When the evaluation of the current node is complete, i.e. the current node is in head normal form, a return is performed. This involves plugging the current node into the previous node (pointed to by the top of the stack) at the specified argument position, making the previous node into the new current node, and popping the stack.

An evaluation step consists of calling the evaluation code for the current node. If the current node is already in head evaluated form, the code causes an immediate return. Otherwise, the code carries out some processing, usually ending with a tail call. Notionally, a tail call consists of building a new node in the heap and making it current. However, this leads to a large turnover of heap space, with nodes being created for every intermediate expression.

As an optimisation to reduce the number of heap nodes created, a tail call can create a temporary node; this is a node allocated in the usual way at the end of the heap, but without advancing the free space pointer. This node will then usually immediately overwrite itself as it in turn creates new heap nodes. To support this, certain nodes such as data nodes must make themselves permanent

**Fig. 6.** The Return Stack

where necessary by advancing the free space pointer before continuing execution. This simple scheme provides many of the same advantages as more complex systems which use registers and stacks for temporary storage of intermediate expressions.

4 The Brisk Abstract Machine

Many approaches to compiling functional languages [9], [1], [16], [4], [15], use abstract machines with imperative machine instructions to provide the operational semantics of programs; see also [3], [12]. With the **STG** Machine [13], a different approach was taken. The **STG** language is a small, low level, intermediate language which is on the one hand still functional, and yet on the other can be given a direct operational state transition semantics to guide its translation into C or machine code.

The Brisk compiler uses the intermediate language **BKL** which, like **STG**, is a low level functional language. A direct semantics could similarly be given for it. However, the computational mobility issues, which Brisk addresses, mean that **BKL** needs to be compiled into interpretive bytecode in some circumstances, and C or machine code in others.

We have chosen to present the operational semantics of programs using a conventional abstract machine similar to the original **G-machine**, so that the machine instructions correspond very closely to the bytecode instructions produced

via the interpretive compiling route. They also guide direct translation into **C** or machine code and provide a link with the design of the run-time system.

The abstract machine described here is called the Brisk Abstract Machine, or **BAM** for short. First, we show how to compile the Brisk Kernel Language into **BAM** code, and then we give the semantics of the individual instructions of the **BAM** code in terms of the run-time system described above.

4.1 Compiling into **BAM** Code

The state of the Brisk Abstract Machine consists of a *frame* of local variables in addition to the return stack and the heap. The frame contains the local variables used during execution. It is completely local to each individual function call, and starts out empty. It is extended in a stack-like manner as needed. This stack-like use is very similar to the stack in the original **G-machine**, and it allows the **BAM** instructions to have at most one argument each, often a small integer, which leads to a compact bytecode.

On the other hand, the **BAM** frame is not used as an argument stack or a return stack, items don't have to be popped off so frequently, and it does not have to be left empty at the end of a function call. This allows us to use it as an array as well a stack, indexed from its base. Initial instructions in the code for each function indicate the maximum size of frame needed, and the maximum amount of heap space to be used, but we omit those here. The function \mathcal{F} compiles code for a function definition:

$$\begin{aligned} \mathcal{F} \llbracket f(g_1, \dots, g_n) \ x_1 \dots x_m = e \rrbracket = \\ & [\text{GetRefs } n, \text{GetArgs } m] ++ \\ & \mathcal{C} \llbracket e \rrbracket [g_1 \mapsto 0, \dots, g_n \mapsto n-1, x_1 \mapsto n, \dots, x_m \mapsto n+m-1] (n+m) \\ & ++ [\text{Enter}] \end{aligned}$$

A function f with references to free variables g_i and arguments x_i is compiled into code which first loads the references and arguments into the frame, then compiles code for the right hand side e which creates a pointer to a node representing e , and then ends by entering the code for this result node.

The function \mathcal{C} generates code for an expression. A call $\mathcal{C} \llbracket e \rrbracket \rho \ n$ generates code for e , where ρ is an environment which specifies where the variables appear in the frame, and n is the current size of the frame.

The code generated for a variable or a constant is:

$$\begin{aligned} \mathcal{C} \llbracket x \rrbracket \rho \ n &= [\text{GetLocal } (\rho x)] \\ \mathcal{C} \llbracket c \rrbracket \rho \ n &= [\text{GetConst } c] \end{aligned}$$

The **GetLocal** instruction gets a variable from its position in the frame and pushes it onto the end of the frame. The **GetConst** instruction pushes a one-word unboxed value onto the end of the frame. The frame is thus untyped; it contains both pointers and raw values. This does not cause any problems since the frame

only lasts for a single function call so, for example, the garbage collector need not be concerned with it.

The code generated for an application is:

$$\begin{aligned} \mathcal{C} \llbracket f \ x_1 \dots x_n \rrbracket \rho \ m = \\ & \mathcal{C} \llbracket x_n \rrbracket \rho \ m ++ \dots ++ \mathcal{C} \llbracket x_1 \rrbracket \rho \ m ++ \\ & \mathcal{C} \llbracket f \rrbracket \rho \ m ++ \\ & [\mathbf{Mkapp} \ (n + 1)] \end{aligned}$$

The code just gathers together the function and its arguments, and then creates a new application node in the heap from them.

The code generated for a **let** construct is:

$$\begin{aligned} \mathcal{C} \llbracket \mathbf{let} \ y_1 = ap_1; \dots; y_n = ap_n \ \mathbf{in} \ e \rrbracket \rho \ m = \\ & [\mathbf{Alloc} \ k_1, \dots, \mathbf{Alloc} \ k_n] \\ & \mathcal{D} \llbracket y_1 = ap_1 \rrbracket \rho' \ m' ++ \dots ++ \mathcal{D} \llbracket y_n = ap_n \rrbracket \rho' \ m' ++ \\ & \mathcal{C} \llbracket e \rrbracket \rho' \ m' \\ & \text{where} \\ & \quad k_i \text{ is the size of application } ap_i \\ & \quad \rho' = \rho \ [y_1 \mapsto m, \dots, y_n \mapsto m + n - 1] \\ & \quad m' = m + n \end{aligned}$$

The code generated here assumes that the **let** construct is recursive. As usual, if the **let** construct is known to be non-recursive, simpler code can be generated, but we don't show that here. First, space is allocated on the heap for each node to be created, and the variables y_i are made to point to these blank nodes. Then each node is filled in in turn using the \mathcal{D} function:

$$\begin{aligned} \mathcal{D} \llbracket y = f \ x_1 \dots x_n \rrbracket \rho \ m = \\ & \mathcal{C} \llbracket x_n \rrbracket \rho \ m ++ \dots ++ \mathcal{C} \llbracket x_1 \rrbracket \rho \ m ++ \\ & \mathcal{C} \llbracket f \rrbracket \rho \ m ++ \mathcal{C} \llbracket y \rrbracket \rho \ m ++ \\ & [\mathbf{Fill} \ (n + 1)] \end{aligned}$$

This generates code to fill in a blank node by gathering the function and its arguments, together with the pointer to the node to be filled, and then using the **Fill** instruction to fill in the node.

The code generated for a **case** construct is:

$$\begin{aligned} \mathcal{C} \llbracket \mathbf{case} \ v \ \mathbf{of} \ alt_1 \dots alt_n \rrbracket \rho \ m = \\ & [\mathbf{Table} \ k] ++ \\ & \mathcal{A} \llbracket alt_1 \rrbracket \rho \ m ++ \dots ++ \mathcal{A} \llbracket alt_n \rrbracket \rho \ m ++ \\ & [\mathbf{Switch} \ (\rho v)] \\ & \text{where } k \text{ is the number of constructors in the type of } v \end{aligned}$$

The code first allocates a jump table with one entry for every constructor in the type of v . This is followed by code for each alternative, generated using \mathcal{A} . An interpreter for the code scans the alternatives, filling in the jump table. Once the jump table is complete, the **case** variable is used by the **Switch** instruction to jump to the appropriate case.

The code generated for an alternative is:

```

 $\mathcal{A} \llbracket C \ x_1 \dots x_n \rightarrow body \rrbracket \rho \ m =$ 
  [Case  $i$ , Skip  $b$ , Split  $n$ ] ++
   $\mathcal{C} \llbracket body \rrbracket \rho' \ m' \ ++$ 
  [Enter]
  where
     $i$  is the sequence number of constructor  $C$ 
     $b$  is the amount of code generated for the body
     $\rho' = \rho \ [x_1 \mapsto m, \dots, x_n \mapsto m + n - 1]$ 
     $m' = m + n$ 
 $\mathcal{A} \llbracket \_ \rightarrow body \rrbracket \rho \ m =$ 
  [Default, Skip  $b$ ] ++
   $\mathcal{C} \llbracket body \rrbracket \rho \ m \ ++$ 
  [Enter]

```

The **Case** instruction is used to fill in an entry in the previously allocated jump table. The entry points to the code for the body of the alternative, just after the **Skip** instruction. The **Skip** instruction skips past the code for the alternative (including the **Split** and **Enter** instructions). The **Split** instruction loads the constructor arguments into the frame, and the environment is extended with their names. A default **case** alternative causes all the unfilled entries in the jump table to be filled in.

4.2 The BAM Instructions

Here, we give transition rules which describe the action of each of the instructions on the run-time system state. The state consists of the current sequence of instructions i , the frame f , the return stack s , and the heap h . When it is needed, the current node will be indicated in the heap by the name cn .

The frame will be represented in a stack-like way as $f = x : y : z : \dots$, but indexing of the form $f!i$ will also be used, with the index being relative to the base of the stack.

The **GetLocal** n instruction pushes a copy of the n 'th frame item onto the end of the frame:

```

GetLocal  $n$  :  $i$        $f$    $s$    $h$ 
               $i$    $f!n$  :  $f$    $s$    $h$ 

```

The **GetConst** c instruction pushes a constant onto the end of the frame:

```

GetConst  $c$  :  $i$        $f$    $s$    $h$ 
               $i$    $c$  :  $f$    $s$    $h$ 

```

The **GetRefs** n instruction loads n references into the frame, from the function node mentioned in the current node:

$$\boxed{\begin{array}{l} \text{GetRefs } n : i \quad f \quad s \quad h \left[\begin{array}{l} cn \mapsto \langle g, \dots \rangle \\ g \mapsto \langle \dots, g_1, \dots, g_n \rangle \end{array} \right] \\ i \quad g_n : \dots : g_1 : f \quad s \quad h \end{array}}$$

The **GetArgs** n instruction extracts n arguments from the current node into the frame:

$$\boxed{\begin{array}{l} \text{GetArgs } n : i \quad f \quad s \quad h[cn \mapsto \langle g, x_1, \dots, x_n \rangle] \\ i \quad x_n : \dots : x_1 : f \quad s \quad h \end{array}}$$

The **Mkap** n instruction assumes that the top of the frame contains a function and its $n - 1$ arguments, from which a node of size n is to be built:

$$\boxed{\begin{array}{l} \text{Mkap } n : i \quad g : x_1 : \dots : x_{n-1} : f \quad s \quad h \\ i \quad p : f \quad s \quad h[p \mapsto \langle g, x_1, \dots, x_{n-1} \rangle] \end{array}}$$

The **Alloc** n instruction allocates space for a node of size n on the heap, with uninitialised contents:

$$\boxed{\begin{array}{l} \text{Alloc } n : i \quad f \quad s \quad h \\ i \quad p : f \quad s \quad h[p \mapsto \langle ?_1, \dots, ?_n \rangle] \end{array}}$$

The **Fill** n instruction fills in a previously allocated node, given its pointer and contents:

$$\boxed{\begin{array}{l} \text{Fill } n : i \quad p : g : x_1 : \dots : x_{n-1} : f \quad s \quad h[p \mapsto \langle ?_1, \dots, ?_n \rangle] \\ i \quad f \quad s \quad h[p \mapsto \langle g, x_1, \dots, x_{n-1} \rangle] \end{array}}$$

To implement the **case** construct, a jump table is needed. We temporarily extend the state, adding the table as an extra component t . The **Table** n instruction allocates an uninitialised table of size n :

$$\boxed{\begin{array}{l} \text{Table } n : i \quad f \quad s \quad h \\ i \quad f \quad s \quad h \quad t \mapsto \langle ?_1, \dots, ?_n \rangle \end{array}}$$

Each **Case** k instruction causes the k 'th table entry to be filled in with the position i in the instruction sequence, just after the subsequent **Skip** instruction:

$$\boxed{\begin{array}{l} \text{Case } k : \text{Skip } b : i \quad f \quad s \quad h \quad t \mapsto \langle \dots, ?_k, \dots \rangle \\ \text{Skip } b : i \quad f \quad s \quad h \quad t \mapsto \langle \dots, i, \dots \rangle \end{array}}$$

A **Default** instruction causes any unfilled entries in the table to be filled in with the relevant position in the instruction sequence:

$$\boxed{\begin{array}{l} \text{Default} : \text{Skip } b : i \quad f \quad s \quad h \quad t \mapsto \langle \dots, ?, \dots, ?, \dots \rangle \\ \text{Skip } b : i \quad f \quad s \quad h \quad t \mapsto \langle \dots, i, \dots, i, \dots \rangle \end{array}}$$

Once the table has been filled in, the **Switch** instruction gets the **case** variable v from the frame and causes a jump to the relevant alternative. The sequence

number k of the constructor C used to build v is used to index the jump table. The sequence number of C can be recorded in its heap node. Execution continues with position i_k in the instruction sequence, where i_k is the k 'th table entry. The table is not needed any more:

$\begin{array}{llll} \text{Switch } n : i & f & s & h[f!n \mapsto \langle C, \dots \rangle] \quad t \mapsto \langle \dots, i_k, \dots \rangle \\ & i_k & f & s \quad h \end{array}$

While the jump table is being built, the **Skip** instruction is used to skip past the code for the alternatives. The instruction **Skip** b simply skips over b instructions (or b bytes if the instructions are converted into bytecode):

$\begin{array}{llll} \text{Skip } b : i & f & s & h \\ & \text{drop } b \ i & f & s \quad h \end{array}$
--

At the beginning of the code for a **case** alternative, the instruction **Split** n is used to extract the fields from the **case** expression, which is still at the top of the frame at this moment:

$\begin{array}{llll} \text{Split } n : i & & v : f & s \quad h[v \mapsto \langle C, x_1, \dots, x_n \rangle] \\ & i \quad x_n : \dots : x_1 : f & s & h \end{array}$
--

At the end of the code for a function, or at the end of an alternative, the **Enter** instruction transfers control to a new function. The node on the top of the frame becomes the current node, a new instruction sequence is extracted from its function node, and execution of the new code begins with an empty frame:

$\begin{array}{llll} \text{Enter} : i & p : f & s & h \left[\begin{array}{l} p \mapsto \langle g, \dots \rangle \\ g \mapsto \langle \dots, fi, \dots \rangle \end{array} \right] \\ & fi & [] & s \quad h [cn = p] \end{array}$

Instructions to manipulate the return stack are handled by special functions such as constructors and the *strict* family, rather than being produced by the compiler. As an example, the code for a constructor is [**Return**, **Enter**] where the **Return** instruction puts the current node into the appropriate argument position of the previous node and pops the return stack:

$\begin{array}{llll} \text{Return} : i & [] & \langle k, p \rangle : s & h \left[\begin{array}{l} p \mapsto \langle \dots, x_k, \dots \rangle \\ cn \mapsto \langle \dots \rangle \end{array} \right] \\ & i \quad [p] & s & h [p \mapsto \langle \dots, cn, \dots \rangle] \end{array}$

5 Conclusions and Further Work

The Brisk Machine been designed so that it provides a flexible model which allows for general experimentation with different execution models. Dynamic loading of newly compiled modules are supported by the ability to load functions as newly created nodes in the heap, and to mix compiled and interpretive bytecode on a function-by-function basis.

The problems of supporting debugging tools are eased by the uniform representation of nodes, the ease with which the layout of nodes can be changed dynamically, and the close relationship of the run-time system to simple graph reduction, without complex stacks and registers.

Concurrency is implemented, as usual, by providing multiple lightweight threads of execution, with time slicing, in the run-time system. The concurrency issues are easily separated from other compilation and execution issues.

Computational mobility in a distributed setting is facilitated by the direct and uniform representation of all expressions, including functions, in the heap. This allows functions to be shipped between processors. In general, communication between processors is implemented in such a way that it is equivalent to having a single distributed heap; this is eased by the ability to insert built-in functions to hide the communication details. A distributed program is thus kept equivalent to its non-distributed counterpart, which helps to make distribution orthogonal to other issues.

The Brisk model is being used by another Bristol group to support work on logic programming extensions. This involves adding run-time variables (in the logic programming sense), changing the details of execution order, and changing the approach to such things as strictness, sharing and updating, largely by replacing built-in functions by alternatives.

Although experiments on all these fronts have been carried out, more work is needed to provide the Brisk system with a convenient development environment, and to investigate its efficiency, particularly in comparison to the STG Machine. Almost all of the optimisations which can be carried out before reaching the STG language (or from STG to STG) are also applicable in the Brisk setting; however, they have not been implemented in Brisk, making direct comparisons difficult at present. In particular, we have yet to investigate the implementation of unboxing. Optimisations which come after reaching the STG language, i.e. during code generation are more difficult to incorporate since they involve stacks, registers, return conventions etc. Nevertheless, some of these have been investigated to see if similar optimisations are possible in Brisk. For example, in the STG approach, intermediate expressions are stored implicitly on stacks, which considerably reduces heap usage. In Brisk, intermediate expressions are stored as heap nodes. Nevertheless, by treating the last node in the heap as a temporary node which can usually be overwritten by a node representing the next intermediate expression, most of the gain of using stacks can be obtained without complicating Brisk's simple approach to representation. Another interesting question is to compare the two approaches to partial applications; on the one hand Brisk avoids the overheads of run-time argument testing and possible subsequent building of partial application nodes. On the other, unevaluated functions have to be evaluated before being called. The tradeoffs are not clear.

Although more work needs to be done, the conceptual simplicity of the Brisk Machine has already proven to be a great advantage in adding experimental features to the system.

References

- [1] Lennart Augustsson. *Compiling lazy functional languages, part II*. PhD thesis, Chalmers University, 1987.
- [2] Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnson, Mark Jones, Erik Meijer, Simon L. Peyton Jones, Alastair Reid, and Philip Wadler. Haskell 1.4, A Non-strict, Purely Functional Language. Technical report, Yale University, 1997.
- [3] Geoffrey Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Pitman, 1991.
- [4] Geoffrey Burn and Simon L. Peyton Jones. The Spineless G-machine. In *Conference on Lisp and Functional programming*, 1988.
- [5] Manuel M.T. Chakravarty and Hendrik C.R. Lock. The JUMP-machine: a Generic Basis for the Integration of Declarative Paradigms. In *Post-ICLP Workshop*, 1994.
- [6] J. Gosling, J. B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.
- [7] Ian Holyer. The Brisk Kernel Language. Technical report, University of Bristol, Department of Computer Science, 1997.
- [8] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In Jouannaud, editor, *Functional Programming and Computer Architecture*, volume 201 of *LNCS*, pages 190–205. Springer Verlag, 1985.
- [9] Thomas Johnsson. *Compiling lazy functional languages*. PhD thesis, Chalmers University, 1987.
- [10] David Lester and Simon L. Peyton Jones. A modular fully-lazy lambda lifter in Haskell. *Software Practice and Experience*, 21(5), 1991.
- [11] J. W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.
- [12] Simon L. Peyton Jones. *Implementing Functional Languages*. Prentice Hall, 1992.
- [13] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, July 1992.
- [14] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming and Computer Architecture*, Sept 1991.
- [15] Simon L. Peyton Jones and John Salkild. The Spineless Tagless G-machine. In MacQueen, editor, *Functional Programming and Computer Architecture*. Addison Wesley, 1989.
- [16] Stuart Wray and J. Fairbairn. Tim - a simple lazy abstract machine to execute supercombinators. In *Functional Programming and Computer Architecture*, volume 274 of *LNCS*, Portland, Oregon, 1987. Springer Verlag.

A Haskell to Java Virtual Machine Code Compiler

David Wakeling

Department of Computer Science, University of Exeter,
Exeter, EX4 4PT, United Kingdom.

(web: <http://www.dcs.exeter.ac.uk/~david>)

Abstract. For some time now, we have been interested in using Haskell to program inexpensive embedded processors, such as those in SUN's new Java family. This paper describes our first attempt to produce a Haskell to Java Virtual Machine code compiler, based on a mapping between the G-machine and the Java Virtual Machine. Although this mapping looks good, it is not perfect, and our first results suggest that the compiled Java Virtual Machine code may be rather larger and slower than one might hope.

1 Introduction

For some time now, we have been interested in the efficient implementation of lazy functional programming languages on very small computers, such as those found in consumer electronics devices. So far, all of our implementations have assumed that next-generation products will be controlled by previous-generation RISC processors [8]. But Java processors, with their compact instruction encoding, are an attractive alternative [7]. This paper investigates whether these processors could successfully run lazy functional programs.

The paper has two parts. The first part points out the similarity between the virtual machine usually used to implement Java [3] and the Chalmers G-machine [5], a virtual machine often used to implement lazy functional languages. Section 2 gives a quick tour of the Java Virtual Machine, Section 3 gives a quick tour of the G-machine, and Section 4 describes a mapping between the two virtual machines that can serve as the basis of a lazy functional language implementation. The second part assesses the effectiveness of the mapping, and suggests how it could be improved. Section 5 presents some benchmark figures, Section 6 discusses these figures, and Section 7 has some ideas for future improvements. Section 8 mentions some related work, and Section 9 concludes.

2 The Java Virtual Machine

In principle, Java could be compiled for any machine, but in practice it is usually compiled for a standard virtual machine. This section gives a quick tour of the Java Virtual Machine; more detail can be found in [3]. Throughout this paper,

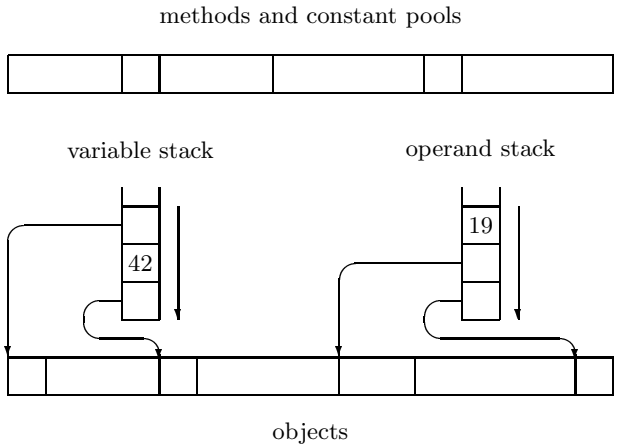


Fig. 1. The Java Virtual Machine.

Java source code and Java Virtual Machine code will be written in **typewriter** font.

As Figure 1 shows, the Java Virtual Machine is a stack-based virtual machine which works with *methods and constant pools*, *objects*, and two *stacks*.

2.1 Methods and Constant Pools

A Java program is organised into *classes*, each of which may have some *methods* (or functions) for performing computation. For every class, the Java Virtual Machine stores the virtual machine code for each method, and a *constant pool* of literals, such as numbers and strings, used by the methods. To ensure the binary portability of Java programs, the layout and byte-order of the stored form is precisely specified. Nevertheless, before the Java Virtual Machine runs any untrusted code it *verifies* it in an attempt to ensure that it is well-behaved.

2.2 Objects

As well as providing methods, classes also describe the structure of *objects*. An object is a record whose fields may be either scalar values, methods or references to other objects. There are virtual machine instructions for allocating a new object, for setting and getting the value of a field, and for invoking a method. But there is no instruction for disposing of an object. A *garbage collector* is assumed to run from time-to-time to recover the memory occupied by objects that are no longer in use.

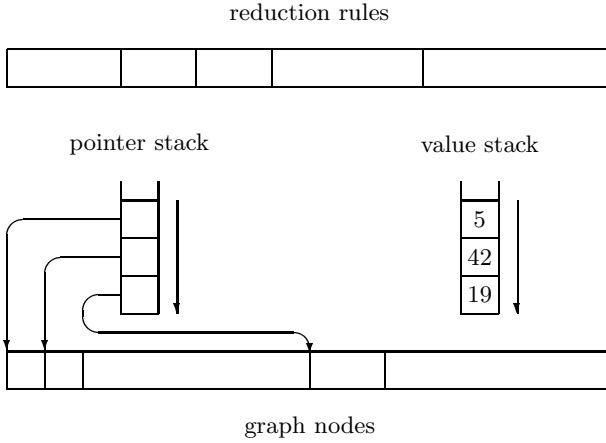


Fig. 2. The G-machine.

2.3 Stacks

The Java Virtual Machine stack is divided into *frames*, each of which stores the local state of a method invocation. In order to simplify our exposition, we have chosen to separate this stack into a *variable stack* from which space for actual parameters and local variables is allocated, and an *operand stack* from which space for the intermediate results of expression evaluations is allocated.

3 The G-machine

Many of the most successful lazy functional language implementations are based on some variant of the Chalmers G-machine. This section gives a quick tour of the G-machine; more detail can be found in [5]. Throughout this paper, G-machine instructions will be written in upper-case **SANS SERIF** font.

As Figure 2 shows, the G-machine is a stack-based virtual machine which works with *graph nodes*, *reduction rules* and two *stacks*.

3.1 Graph Nodes

A lazy functional program is executed by evaluating an expression to normal form, printing the result as it becomes available. The G-machine represents the expression by a *graph*, and it evaluates it by *graph reduction*. As the graph is reduced, new nodes are attached to it and existing nodes are detached from it. From time-to-time, a *garbage collector* recovers the memory occupied by detached nodes.

3.2 Reduction Rules

Each function or primitive operation serves as a *reduction rule* for the graph. By way of example, Figure 3 describes simple functional programs as a set of reduction rules of varying arity. The body of each rule is an expression which can be either a function, an argument, an integer, an addition, a conditional, or an application. Continuing with the example, Figure 4 shows how the reduction rules can be compiled into G-machine code. Executing the G-machine code for a rule reduces the graph in the manner required by that rule.

$$\begin{array}{l}
 p ::= f_1 = \lambda x_{11} \cdots \lambda x_{1m_1}.e_1 \\
 \vdots \\
 f_n = \lambda x_{n1} \cdots \lambda x_{nm_n}.e_n \\
 \\
 e ::= f \\
 \quad | \quad x \\
 \quad | \quad i \\
 \quad | \quad e_1 + e_2 \\
 \quad | \quad \text{if } e_1 \ e_2 \ e_3 \\
 \quad | \quad e_1 \ e_2
 \end{array}$$

Fig. 3. Program syntax.

3.3 Stacks

In order to make garbage collection easier, the G-machine stores the local state of a function application on two stacks, both of which can be divided into *frames*. The *pointer stack* stores pointers to argument and intermediate graph nodes, and the *value stack* stores unboxed constants.

4 A Mapping Between Virtual Machines

Figures 1 and 2 and the accompanying text are intended to suggest the mapping between the G-machine and the Java Virtual Machine discussed below.

4.1 Mapping Graph Nodes to Objects

The mapping of the G-machine's graph nodes to the Java Virtual Machine's objects is largely straightforward. It is natural to represent general graph nodes by a class, `N`, and particular graph nodes by subclasses of that class. Figure 5 shows the abstract class, `N`. Here, the `ev` method returns a node representing this

$\mathcal{F} \llbracket f = \lambda x_1 \cdots \lambda x_n. e \rrbracket$	$= \mathcal{R} \llbracket e \rrbracket \llbracket x_1 = n, x_2 = n - 1 \cdots x_n = 1 \rrbracket n$
$\mathcal{R} \llbracket e_1 + e_2 \rrbracket \rho n$	$= \mathcal{E} \llbracket e_1 + e_2 \rrbracket \rho n; \text{UPDATE } (n + 1); \text{POP } n; \text{UNWIND}$
$\mathcal{R} \llbracket \text{if } e_1 \ e_2 \ e_3 \rrbracket \rho n$	$= \mathcal{B} \llbracket e_1 \rrbracket \rho n \text{ JFALSE } l_1;$ $\quad \mathcal{R} \llbracket e_2 \rrbracket \rho n;$ $\quad \text{LABEL } l_1;$ $\quad \mathcal{R} \llbracket e_3 \rrbracket \rho n$
$\mathcal{R} \llbracket e \rrbracket \rho n$	$= \mathcal{C} \llbracket e \rrbracket \rho n; \text{UPDATE } (n + 1); \text{POP } n; \text{UNWIND}$
$\mathcal{E} \llbracket i \rrbracket \rho n$	$= \text{PUSHINT } i$
$\mathcal{E} \llbracket e_1 + e_2 \rrbracket \rho n$	$= \mathcal{B} \llbracket e_1 + e_2 \rrbracket \rho n; \text{MKINT}$
$\mathcal{E} \llbracket \text{if } e_1 \ e_2 \ e_3 \rrbracket \rho n$	$= \mathcal{B} \llbracket e_1 \rrbracket \rho n; \text{JFALSE } l_1;$ $\quad \mathcal{E} \llbracket e_2 \rrbracket \rho n; \text{JMP } l_2;$ $\quad \text{LABEL } l_1;$ $\quad \mathcal{E} \llbracket e_3 \rrbracket \rho n;$ $\quad \text{LABEL } l_2$
$\mathcal{E} \llbracket e \rrbracket \rho n$	$= \mathcal{C} \llbracket e \rrbracket \rho n; \text{EVAL}$
$\mathcal{B} \llbracket i \rrbracket \rho n$	$= \text{PUSHBASIC } i$
$\mathcal{B} \llbracket e_1 + e_2 \rrbracket \rho n$	$= \mathcal{B} \llbracket e_2 \rrbracket \rho n; \mathcal{B} \llbracket e_1 \rrbracket \rho (n + 1); \text{ADD}$
$\mathcal{E} \llbracket \text{if } e_1 \ e_2 \ e_3 \rrbracket \rho n$	$= \mathcal{B} \llbracket e_1 \rrbracket \rho n; \text{JFALSE } l_1;$ $\quad \mathcal{B} \llbracket e_2 \rrbracket \rho n; \text{JMP } l_2;$ $\quad \text{LABEL } l_1;$ $\quad \mathcal{B} \llbracket e_3 \rrbracket \rho n;$ $\quad \text{LABEL } l_2$
$\mathcal{B} \llbracket e \rrbracket \rho n$	$= \mathcal{E} \llbracket e \rrbracket \rho n; \text{GET}$
$\mathcal{C} \llbracket f \rrbracket \rho n$	$= \text{PUSHGLOBAL } f$
$\mathcal{C} \llbracket x \rrbracket \rho n$	$= \text{PUSH } n - \rho(x)$
$\mathcal{C} \llbracket i \rrbracket \rho n$	$= \text{PUSHINT } i$
$\mathcal{C} \llbracket e_1 + e_2 \rrbracket \rho n$	$= \mathcal{C} \llbracket e_2 \rrbracket \rho n; \mathcal{C} \llbracket e_1 \rrbracket \rho (n + 1);$ $\quad \text{PUSHGLOBAL PLUS}; \text{MKAP}; \text{MKAP}$
$\mathcal{C} \llbracket \text{if } e_1 \ e_2 \ e_3 \rrbracket \rho n$	$= \mathcal{C} \llbracket e_3 \rrbracket \rho n; \mathcal{C} \llbracket e_2 \rrbracket \rho (n + 1); \mathcal{C} \llbracket e_1 \rrbracket \rho (n + 2);$ $\quad \text{PUSHGLOBAL COND}; \text{MKAP}; \text{MKAP}; \text{MKAP};$
$\mathcal{C} \llbracket e_1 \ e_2 \rrbracket \rho n$	$= \mathcal{C} \llbracket e_2 \rrbracket \rho n; \mathcal{C} \llbracket e_1 \rrbracket \rho (n + 1); \text{MKAP}$

Fig. 4. Compilation schemes.

```

abstract public class N {
    public abstract N    ev();
    public abstract void uw();
    public abstract int  gt();
    public abstract N    rn();
    public abstract void ud(N g);
}

```

Fig. 5. The abstract graph node class, N.

node in normal form; the `uw` method initiates a further reduction if this node is not in normal form; the `gt` method returns the tag of this node; the `rn` method runs the code associated with this (function) node; and the `ud` method updates

```
final public class AP extends N {
    N ind, f, a;

    public AP(N f, N a) {
        this.ind = null;
        this.f    = f;
        this.a    = a;
    }

    public N ev() {
        if (this.ind == null) {
            return this.ind = RT.APev(this);
        } else {
            return this.ind.ev();
        }
    }

    public void uw() {
        if (this.ind == null) {
            RT.APuw(this);
        } else {
            this.ind.uw();
        }
    }

    public N rn() {
        RT.Stop("A.rn()");
        return null;
    }

    public void ud(N g) {
        this.ind = g;
    }
}
```

Fig. 6. The application node class AP.

this (application) node with a node representing its normal form. Usually, an update involves overwriting the node with either a *copy* of the normal form node, or with an *indirection node* that points to it. Sadly, the Java Virtual Machine does not provide a way to copy one object over another, so one must update by indirection. Thus, each updatable node is represented by an object with an extra indirection field. The indirection field is initially `null`, and the

update method sets it to reference a normal form node object. During graph reduction, the indirection field of a node must be followed if it is not `null`. As an example, Figure 6 gives Java code for the application node class `AP`.

Constant Applicative Forms (or CAFs) can also sometimes be a source of trouble for functional language implementors, but here they give no difficulty. There is a class for each CAF, and these classes are all subclasses of the graph node class. Like other updatable nodes, a CAF node object has an indirection field that is initially `null`, and is later set to reference a normal form node object. However, in this case the indirection field is a *class variable* shared between all instances of the CAF class, so that when one is evaluated, the others “feel the benefits”.

For a while, we represented graph nodes by a more elaborate class structure. There were subclasses of the node class gathering together constant nodes, function nodes, updatable nodes, and so on. The intention was to gain security by using the Java Virtual Machine’s type-checking to trap errors in the implementation of graph reduction, and to save space by having the methods for graph node operations in just the classes where they were needed. This did not work out well, however, because many extra Java Virtual Machine instructions were needed for type-checking. An update, for example, can only take place once it has been checked that the node to be updated is in the subclass of updatable nodes. In a correct implementation, of course, it always is.

4.2 Mapping Reduction Rules to Methods and Constant Pools

Figure 7 shows how G-machine instructions are mapped onto Java Virtual Machine instructions. For a rule of arity n whose G-machine code is gs , $\mathcal{T} \llbracket gs \rrbracket n$ gives the Java Virtual Machine code for the `rn` method. As well as producing Java Virtual Machine instructions, some mappings add new entries to the class constant pool, seen here as a map between integers and entries.

4.3 Mapping the Stacks

The G-machine’s value stack is easily mapped onto the Java Virtual Machine’s operand stack. However, mapping the G-machine’s pointer stack onto the Java Virtual Machine’s variable stack is a bit harder. The `rn` method supposes that once an application spine has been unwound and rearranged, its arguments can be efficiently accessed as local variables (see Figure 8). This is achieved by unwinding the spine onto a stack array in the run-time system, and then having some prologue code at the start of the `rn` method do the rearrangement whilst transferring the arguments from the stack array to the local variables. Figure 9 gives the methods for evaluating and unwinding an application spine.

4.4 Tail Calls

A tail call occurs when the result of one function is given by a call to another function with exactly the right number of arguments supplied. In this case, an

$\mathcal{T} \llbracket \text{ADD}.gs \rrbracket sp$	$= \text{iadd}; \mathcal{T} \llbracket gs \rrbracket sp$
$\mathcal{T} \llbracket \text{EVAL}.gs \rrbracket sp$	$= \text{aload } sp; \text{invokevirtual } i; \text{astore } sp; \mathcal{T} \llbracket gs \rrbracket sp$ $\text{pool} + [i \mapsto \text{MethodRef } N \text{ ev } ()N]$
$\mathcal{T} \llbracket \text{GET}.gs \rrbracket sp$	$= \text{aload } sp; \text{invokevirtual } i; \mathcal{T} \llbracket gs \rrbracket (sp - 1)$ $\text{pool} + [i \mapsto \text{MethodRef } N \text{ gt } ()I]$
$\mathcal{T} \llbracket \text{JFALSE } l.gs \rrbracket sp$	$= \text{ifeq } l; \mathcal{T} \llbracket gs \rrbracket sp$
$\mathcal{T} \llbracket \text{JMP } l.gs \rrbracket sp$	$= \text{goto } l; \mathcal{T} \llbracket gs \rrbracket sp$
$\mathcal{T} \llbracket \text{MKAP}.gs \rrbracket sp$	$= \text{new } i; \text{dup}; \text{aload } sp; \text{aload } (sp - 1);$ $\text{invokespecial } j; \text{astore } (sp - 1); \mathcal{T} \llbracket gs \rrbracket (sp - 1)$ $\text{pool} + [i \mapsto \text{Class } AP,$ $\quad j \mapsto \text{MethodRef } AP <\text{init}> (NN)V]$
$\mathcal{T} \llbracket \text{MKINT}.gs \rrbracket sp$	$= \text{new } i; \text{dup}; \text{dup2_x1}; \text{pop2};$ $\text{invokespecial } j; \text{astore } (sp + 1); \mathcal{T} \llbracket gs \rrbracket (sp + 1)$ $\text{pool} + [i \mapsto \text{Class } INT,$ $\quad j \mapsto \text{MethodRef } INT <\text{init}> (I)V]$
$\mathcal{T} \llbracket \text{POP } n.gs \rrbracket sp$	$= \mathcal{T} \llbracket gs \rrbracket (sp - n)$
$\mathcal{T} \llbracket \text{PUSH } n.gs \rrbracket sp$	$= \text{aload } (sp - n); \text{astore } (sp + 1);$ $\mathcal{T} \llbracket gs \rrbracket (sp + 1)$
$\mathcal{T} \llbracket \text{PUSHBASIC } i.gs \rrbracket sp$	$= \text{sipush } i; \mathcal{T} \llbracket gs \rrbracket sp$
$\mathcal{T} \llbracket \text{PUSHGLOBAL } f.gs \rrbracket sp$	$= \text{new } i; \text{dup}; \text{invokespecial } j; \text{astore } (sp + 1);$ $\mathcal{T} \llbracket gs \rrbracket (sp + 1)$ $\text{pool} + [i \mapsto \text{Class } f, j \mapsto \text{MethodRef } i <\text{init}> ()V]$
$\mathcal{T} \llbracket \text{PUSHINT } i.gs \rrbracket sp$	$= \mathcal{T} \llbracket \text{PUSHBASIC } i.\text{MKINT}.gs \rrbracket sp$
$\mathcal{T} \llbracket \text{UPDATE } n.gs \rrbracket sp$	$= \text{aload } (sp - n); \text{aload } sp; \text{invokevirtual } i;$ $\mathcal{T} \llbracket gs \rrbracket (sp - 1)$ $\text{pool} + [i \mapsto \text{MethodRef } N \text{ ud } (N)V]$
$\mathcal{T} \llbracket \text{UNWIND}.gs \rrbracket sp$	$= \text{aload } sp; \text{invokevirtual } i; \mathcal{T} \llbracket gs \rrbracket sp$ $\text{pool} + [i \mapsto \text{MethodRef } N \text{ uw } (N)V]$

Fig. 7. Instruction translation scheme.

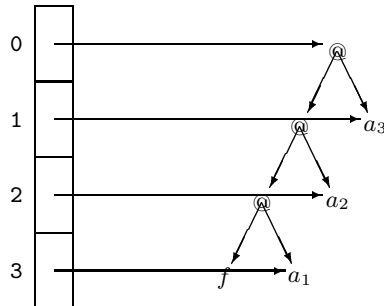


Fig. 8. Argument access via local variables.

```

public static N APev(AP a) {
    APuw(a);
    return S[ sp++ ];
}

public static void APuw(AP a) {
    S[ --sp ] = a;
    a.f.uw();
}

```

Fig. 9. Application node methods.

efficient implementation discards the frame of the first function before allocating that for the second. Unfortunately, the Java Virtual Machine is not required to implement tail calls like this, and so we must use a well-known trick to achieve the proper behaviour. The trick is a variant of the “UUO handler” invented for the Rabbit Scheme compiler [6] and later popularised as the “tiny interpreter” by the Glasgow Haskell compiler [2]. A tail call is made by returning a function node object whose `rn` is to be invoked, rather than by invoking this method directly. Execution is then controlled by the one-line tiny interpreter:

```
while (f != null) f = f.rn();
```

The `UNWIND` instruction eventually encounters a normal form, which it leaves on the top of the stack array and returns a `null` object. This implementation of tail calls looks costly because a temporary function node object must be created for each tail call. It can be avoided in the important special case when a function makes a tail-call to itself; here a `goto` instruction suffices. But in general it seems to be the best that can be done, given that there are no function pointers in the Java Virtual Machine.

5 Benchmarks

A compiler from Haskell to Java Virtual Machine code has been constructed using the ideas described above, based on version 0.9999.3 of the Chalmers HBC compiler. This compiler has been used to compile seven benchmark programs: `nfib30`, the unfashionable benchmarking function with type `Int -> Int` and argument 30; `calendar`, a program that formats 7 calendars; `clausify`, which converts propositional formulae to clausal form; `soda`, which performs a word search in a 20×30 grid; `infer`, a type-checker written in the monadic style; `parser`, a Haskell parser; and `prolog`, a logic programming system solving the Towers of Hanoi problem for six discs.

All of the benchmark figures reported here were recorded on a SUN Ultra 140 workstation with 64Mbytes of memory, running the Solaris 2.5.1 operating system. The machine was running version 1.1.3 of the SUN Java Developer's Kit, which performs "just-in-time" compilation of Java Virtual Machine code to native code. Tables 1 and 2 compare three implementations: our compiler, an unmodified version 0.9999.3 Chalmers HBC compiler, and the Nottingham Hugs interpreter. For our compiler, program sizes are obtained by adding up the sizes of all the generated ".class" files; for the HBC compiler, they are as measured by the `size` command; for the Hugs interpreter, no program size figures are given because the G-machine code that it interprets cannot easily be extracted from the interactive environment. None of the sizes includes type classes and functions from the standard prelude, or support routines from the run-time system. It is not easy to pick-out bits of the prelude, and in any case none of the implementations treat it significantly differently from other Haskell code. Since our compiler has only a minimal run-time system, comparisons of run-time system size would not be fair. For our compiler and the HBC compiler, the timings are an average of those for five consecutive runs after an initial "warm up" run; for Hugs they are an average of five runs, each made immediately after starting the interpreter.

Benchmark	Our compiler Time (s)	Hugs Time (s)	HBC Time (s)
<code>nfib30</code>	50.0	106.7	1.5
<code>calendars</code>	9.6	4.8	0.1
<code>clausify</code>	14.2	7.4	0.5
<code>soda</code>	8.0	2.7	0.2
<code>infer</code>	35.7	8.2	0.9
<code>parser</code>	61.5	8.8	0.9
<code>prolog</code>	76.2	9.9	1.3

Table 1. Execution times.

Benchmark	Our compiler Size (bytes)	Hugs Size (bytes)	HBC Size (bytes)
<code>nfib30</code>	2,729	—	2,545
<code>calendars</code>	28,517	—	46,540
<code>clausify</code>	24,152	—	33,693
<code>soda</code>	12,574	—	24,497
<code>infer</code>	204,807	—	222,335
<code>parser</code>	254,007	—	339,192
<code>prolog</code>	99,081	—	124,923

Table 2. Program sizes.

6 Discussion

These benchmark results are disappointing. Our compiler produces programs that are between half and three quarters of the size of those produced by the ordinary HBC compiler, but run between 30 and 60 times more slowly. Indeed, the programs run between 2 and 9 times more slowly than with the Hugs interpreter.

From the point-of-view of embedded applications, the program size is the real concern because a hardware implementation of the Java Virtual Machine can (only) improve program speed. So why are programs so large? Mostly because of the way that tail-calls are implemented. Recall that a tail-call is made by returning a function node object to the tiny interpreter. For each such object, there must be a class, and for each such class, there must be a “.class” file. This amounts to a “.class” file for every function and CAF in the program. Each of these has its own methods and constant pool, with no sharing possible between them. In typical programs, we have found that the byte code for Java Virtual Machine instructions accounts for only 40% of the space.

It would be unreasonable, of course, to expect the SUN Java Virtual Machine to run the programs that our compiler produces as fast as the SUN SPARC processor can run the programs that HBC produces. But it is reasonable to expect it to run them as fast as the Hugs interpreter does, because the G-machine used by our compiler is much more sophisticated than the one used by Hugs (for example, it has n -ary application nodes instead of just binary ones). So why are programs so slow? Consider the small program in Figure 10. Figure 10). Its result is the character ‘a’, the last of a list of 50,000. Computing

```
main = putChar (last legion)

legion = take 50000 (repeat 'a')
```

Fig. 10. A small, allocation-intensive benchmark.

this result involves (lazily) allocating 50,000 “cons” nodes to represent the list, and 50,000 application nodes to represent suspended applications of `take`. In addition, 50,000 function nodes are returned to the tiny interpreter. VSD runs the program in 41.4 seconds, and Hugs in just 3.1 seconds. Even allowing that Hugs does not have to return function nodes to a tiny interpreter, it seems that memory allocation/reclamation is an order of magnitude more expensive in the Java Virtual Machine than in the Hugs run-time system. A more sophisticated G-machine can get back some, but not all of this. Other small, allocation-intensive benchmarks give similar results.

Garbage collection is also a problem. Both the stack array and the local variables are potential sources of space-leaks because the Java Virtual Machine garbage collector cannot know that they are being used as a stack. Thus, it holds onto everything referenced from the array and variables, regardless of where the

stack pointer is. This is serious. The input to the larger programs has had to be made smaller to avoid running out of memory because of space-leaks.

Three other points are worth mentioning. Firstly, the Java Virtual Machine uses dynamic linking: at the start of each program run, method references are resolved to memory addresses by loading files from disk. This costs, but not much because the computer's operating system caches ".class" files in memory during the "warm-up" run, and then does not have to access the disk again. Next, the Java Virtual Machine performs/requires run-time checks that are unnecessary for a strongly-typed functional language, such as Haskell. Although memory is never accessed through a `null` pointer, and the pattern-matching code that splits apart a pair never gets anything else, these things are checked anyway. Omitting check instructions can make programs upto 5% faster, but the Java Virtual Machine code is no longer verifiable. Finally, there are the run-time support routines. In the Hugs implementation they are written in C and compiled; in our implementation they are written in Java and interpreted. Compiled code, of course, runs much faster than interpreted code.

7 Future Work

One problem with our implementation is the mapping of the G-machine's pointer stack to a Java array in the run-time system and the Java Virtual Machine's local variables. Pointers must constantly be moved between the array and local variables, and space-leaks occur because the garbage collector cannot know that both are being used as a stack. To some extent, the garbage collection problem could be solved by mapping the pointer stack to a linked-list of frames instead of an array. Taking this idea further, the movement of pointers could also be avoided by using the $\langle \nu, G \rangle$ -machine [1], where stack space is allocated as part of the graph nodes, instead of the ordinary G-machine. We have just built such an implementation, and it looks more promising.

Another problem is with SUN's implementation of the Java Virtual Machine. The high cost of memory allocation/reclamation is quite surprising, and tail calls are not dealt with properly by discarding the frame for one method invocation before allocating a frame for the next. Of course, one can hope that better commercial implementations will appear in future, but in the meantime it might be interesting to fit out one of the free ones with more efficient storage management — perhaps using semispaces instead of mark/sweep — and a proper treatment of tail-calls.

8 Related Work

The instant success of Java has attracted the attention of others in the functional programming community, notably Odersky and Wadler [4]. Their Pizza implementation extends Java with parametric polymorphism, higher-order functions and algebraic data types in order to make these ideas more widely accessible.

Pizza can be used as a functional programming language, and object-oriented programming language, or something in between.

Compilers for many other programming languages to Java Virtual Machine code are either available now, or are on the way. For example, Cygnus Support have a compiler called Kawa for Scheme, INRIA-Lorraine have a compiler called SmallEiffel for Eiffel, and Intermetrics have a compiler called Ada Magic for Ada 95.

9 Conclusions

In this paper we have investigated whether Java processors could be programmed with lazy functional languages in the context of embedded applications. Our work is based on a mapping between the G-machine and the Java Virtual Machine. In principle, the mapping is a good one, but in practice programs are not nearly as small or as fast as one might hope. This is largely due to the high cost of memory allocation/reclamation, and the difficulty of implementing tail-calls in the Java Virtual Machine. Despite our disappointing early results, we are not downhearted, and intend to continue developing a Haskell compiler for the Java Virtual Machine.

Acknowledgements

Our thanks as always to Lennart Augustsson and Thomas Johnsson, whose work on the HBC compiler forms the basis of our own.

This work was supported by the University of Exeter Research Fund, and by Canon Research Centre Europe Limited.

References

- [1] L. Augustsson and T. Johnsson. Parallel Graph Reduction with the $\langle\nu, g\rangle$ -machine. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, pages 202–213. ACM Press, 1989.
- [2] S. L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, pages 127–202, April 1992.
- [3] T. Lindholm and F. Yellin. *The Java Virtual Machine*. Addison-Wesley, September 1996.
- [4] M. Odersky and P. Wadler. Pizza into Java: Translating Theory into Practice. In *Proceedings of the 1997 ACM Symposium on the Principles of Programming Languages*, pages 146–149, January 1997.
- [5] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- [6] G. L. Steele. Rabbit: A Compiler for Scheme. Technical Report AI-TR-474, MIT Laboratory for Computer Science, 1978.

- [7] SUN Microsystems Inc. PicoJava I Microprocessor Core Architecture. Technical Report WPR-0014-01, SUN Microsystems, 1997.
- [8] D. Wakeling. A Throw-away Compiler for a Lazy Functional Language. In M. Takeichi and T. Ida, editors, *Fuji International Workshop on Functional and Logic Programming*, pages 287–300. World Scientific, July 1995.

Common Subexpressions Are Uncommon in Lazy Functional Languages

Olaf Chitil

Lehrstuhl für Informatik II, Aachen University of Technology, Germany
`chitil@informatik.rwth-aachen.de`
<http://www-i2.informatik.RWTH-Aachen.de/~chitil>

Abstract. Common subexpression elimination is a well-known compiler optimisation that saves time by avoiding the repetition of the same computation. In lazy functional languages, referential transparency renders the identification of common subexpressions very simple. More common subexpressions can be recognised because they can be of arbitrary type whereas standard common subexpression elimination only shares primitive values. However, because lazy functional languages decouple program structure from data space allocation and control flow, analysing its effects and deciding under which conditions the elimination of a common subexpression is beneficial proves to be quite difficult. We developed and implemented the transformation for the language Haskell by extending the Glasgow Haskell compiler. On real-world programs the transformation showed nearly no effect. The reason is that common subexpressions whose elimination could speed up programs are uncommon in lazy functional languages.

1 Transformation of Different Language Classes

The purpose of common subexpression elimination (CSE) is to reduce the runtime of a program through avoiding the repetition of the same computation. The transformation statically identifies a repeated computation by locating multiple occurrences of the same expression. Repeated computations are eliminated by storing the result of evaluating the expression in a variable and accessing this variable instead of reevaluating the expression.

1.1 Imperative Languages

CSE is a well-known standard optimisation which is implemented in most compilers for imperative languages ([1]). The program to be optimised is represented as a flow graph whose nodes are basic blocks, that is sequences of 3-address instructions. An expression on the right hand side of an assignment is a common subexpression if it has been computed before and there is no assignment to any variable of the expression in between. For languages with pointers the latter condition is more complicated. Local elimination of all common subexpressions

of a basic block is straightforward. Global elimination requires a data flow analysis, since an expression can only be eliminated, if it is computed on every path leading to its basic block.

It is important to note that some transformations are not feasible on source code level, because the required details are still hidden there. For example, Pascal only permits to access an array by an index, e.g. `a[i] := a[i]+1`. Assuming an array component requires 4 bytes this assignment is translated into the following 3-address code:

```

t1    := 4 * i;
t2    := a[t1]
t3    := t2 + 1;
t4    := 4 * i;
a[t4] := t3;
```

The programmer cannot avoid the repeated computation of `4 * i` which is eliminated by CSE.

Note that 3-address code only handles primitive data such as integers and floating point values and that the temporary variables `t1, ..., t4` are held in processor registers. The recomputation of complete arrays for example cannot be eliminated.

1.2 Strict Functional Languages

Appel implemented CSE in a compiler for the strict functional language ML ([2], Chapter 9). He uses continuation passing style as intermediate language on which all transformations operate. Whereas 3-address code consists of a sequence of instructions, continuation passing style code makes control flow explicit by nesting. Hence an expression is evaluated before another expression, if it syntactically dominates that expression. Only in case of syntactic domination common subexpressions can be eliminated. To increase the applicability of the transformation, an additional hoisting transformation is implemented which hoists a continuation expression above another. The following simplified example from ([2], Chapter 9) shows the transformation of

```
let f c = c (x+y) in f (x+y) k
```

This expression is written in continuation passing style as

```

FIX([(f, [c], PRIMOP(+, [VAR x, VAR y], [z], [
    APP(VAR c, [VAR z])]))],
    PRIMOP(+, [VAR x, VAR y], [w], [
    APP(VAR f, [VAR w, VAR k])]))]
```

which is transformed by hoisting into

```

PRIMOP(+, [VAR x, VAR y], [w], [
    FIX([(f, [c], PRIMOP(+, [VAR x, VAR y], [z], [
        APP(VAR c, [VAR z])]))],
        APP(VAR f, [VAR w, VAR k])]))]
```

and by CSE into

```
PRIMOP(+, [VAR x, VAR y], [w], [
  FIX([(f, [c], APP(VAR c, [VAR w]))],
    APP(VAR f, [VAR w, VAR k]))])
```

Common subexpressions are restricted to expressions built from primitive operations which operate only on primitive types. We conclude that CSE for continuation passing style programs is very similar to CSE for imperative programs.

Appel reports that the transformation has no effect on run-time and only a minor positive effect on program size. However, Appel gives no explanation for this disappointing result. Subsequently developed ML-compilers still use CSE ([16]).

1.3 Lazy Functional Languages

To our knowledge, CSE has not yet been implemented in any compiler for lazy functional languages. As seen in the previous subsections, classic CSE is based on an explicit representation of control flow and data flow. In contrast, lazy functional languages decouple program structure from both control flow and data space allocation. That makes it hard, first, to ascertain that repeated expressions are evaluated repeatedly and, second, to predict the effect of the elimination of a common subexpression on space usage, a problem that we will discuss shortly.

These discrepancies suggest that CSE should be applied at a lower level in a compiler for a lazy functional language. The Glasgow Haskell compiler produces C programs. However, these C programs contain many indirect function calls via pointers ([11]). We suppose that these function calls severely limit the ability of the GNU C-compiler gcc to find common subexpressions. Unfortunately we are not able to verify this claim, because gcc does not provide an option for suppressing CSE.

On the other hand, there are several advantages of applying CSE directly to lazy functional programs.

First, lazy functional languages like Haskell are referentially transparent, that is, two identical expressions always denote the same value, independent of the time of evaluation. Hence the recognition of common subexpressions is easier to implement than for imperative languages or strict functional languages like Scheme and ML that have to take account of destructive updates and side effects. Thus even more common subexpressions may be recognised.

Second, CSE for lazy functional languages automatically recognises common subexpressions of arbitrary type. Therefore it is able to transform

```
sum [1..1000] + sum [-1000..1] + sum [1..1000]
```

into

```
let v = sum [1..1000] in v + sum [-1000..1] + v
```

CSE for imperative languages and as used by Appel can only eliminate an expression, if all its subexpressions including itself only handle primitive values.

The disadvantage of eliminating expressions of arbitrary type is that it can lead to a considerable increase in space requirements (cf. [10], Sections 14.7.2 and 23.4.2). Consider the transformation of the expression

```
sum [1..1000] + sum [-1000..1] + prod [1..1000]
```

into

```
let v = [1..1000] in sum v + sum [-1000..1] + prod v
```

The first expression creates three times a list of 1000 elements. The space of a list can be reclaimed immediately after the list is used by `sum` and `prod` respectively. Hence the amount of space required by one list suffices for the evaluation of the whole expression. In the second expression the space allocated for the list `[1..1000]` is not available when evaluating `sum [-1000..1]`, but can only be reclaimed after the evaluation of `prod v` has finished (assuming a left to right evaluation). In the case of such a "space leak" it could be cheaper to recompute the common expression. Santos shortly discusses CSE for the lazy functional language Haskell in [15] and points out this danger of "space leaks". He suggests restricting the type of common subexpression that are eliminated. We will follow up this idea in Section 4.2. However, if the lifetime of the original expressions overlap, then sharing compound values is even beneficial for space consumption.

To evaluate the usefulness of CSE for lazy functional languages in practise, we implemented it as an extension of the Glasgow Haskell compiler (GHC). In the next section we give a short introduction to GHC and present the simple lazy language Core on which our transformation operates. In Section 3 the transformation is developed in detail and in Section 4 we discuss, how the problems mentioned above are (partially) overcome. Because we do not want to loose the advantage of simplicity by performing a complex analysis, we have to find simple syntactic conditions under which the elimination of a common subexpression is beneficial. Section 5 discusses the implementation of the transformation. Afterwards, Section 6 presents measurements of the effects of the transformation on several real-world programs. In Section 7 we discuss the main result of this paper: CSE is not effective for our test programs because they have only few common subexpressions. This lack of common subexpressions is likely to be a characteristics of lazy functional languages. We conclude in Section 8.

2 The Glasgow Haskell Compiler and Core

We chose to implement CSE by extending GHC for the following reasons. First, GHC is heavily based on the "compilation by transformation" approach, that is, it consists of a front end which translates Haskell into a small lazy functional language named Core, a number of transformations which optimise Core programs, and a back end which translates Core into C. As much work as possible is done in the middle part. Furthermore, GHC has been designed with the goal

Program	$Prog \rightarrow Bind_1; \dots; Bind_n$	$n \geq 1$
Binding	$Bind \rightarrow var = Expr$ $\quad \text{rec } var_1 = Expr_1;$ $\quad \quad \dots;$ $\quad \quad var_n = Expr_n;$	Non-recursive Recursive $n \geq 1$
Expression	$Expr \rightarrow Expr \ Atom$ $\quad Expr \ ty$ $\quad \lambda var_1 \dots var_n \rightarrow Expr$ $\quad Atyvar_1 \dots tyvar_n \rightarrow Expr$ $\quad \text{case } Expr \text{ of } \{Alts\}$ $\quad \text{let } Bind \text{ in } Expr$ $\quad con \ var_1 \dots var_n$ $\quad prim \ var_1 \dots var_n$ $\quad Atom$	Application Type application Lambda abstraction Type abstraction Case expression Local definition Constructor $n \geq 0$ Primitive op. $n \geq 0$
Atoms	$Atom \rightarrow var$ $\quad Literal$	Variable Unboxed object
Literals	$Literal \rightarrow integer \mid float \mid \dots$	
Alternatives	$Alts \rightarrow Calt_1; \dots; Calt_n; Default \ n \geq 0$ $\quad Lalt_1; \dots; Lalt_n; Default \ n \geq 0$	
Constr. alt.	$Calt \rightarrow con \ var_1 \dots var_n \rightarrow Expr \quad n \geq 0$	
Literal alt.	$Lalt \rightarrow Literal \rightarrow Expr$	
Default alt.	$Default \rightarrow NoDefault$ $\quad var \rightarrow Expr$	

Fig. 1. Syntax of the Core language

that other people can extend it with new optimising transformations ([8], [3]). Second, it is one of the standard compilers for the lazy language Haskell. This permits us to test our transformation on real-world Haskell programs instead of toy programs in a toy language. The compiler itself is written in Haskell. We added our transformation to version 2.08 which implements Haskell 1.4 ([4], [9]).

The intermediate language of GHC, Core, is essentially the second-order λ -calculus augmented with **let**, **case**, data constructors, constants and primitive operations. The syntax of the language is given in Figure 1. To avoid always having to speak of global bindings and (local) **let** bindings, we refer to both kind of bindings as **let** bindings. The syntax does not include algebraic data type definitions, but data constructors are used in the patterns of case alternatives. Note that function arguments must be atoms to simplify the operational semantics of Core and many transformations. Core has a fixed operational semantics

besides the usual denotational semantics to enable reasoning about the usefulness of a transformation. Hence we shortly describe the main characteristics of this operational semantics.

Type abstraction and application are only needed for the type system. No program code is generated for these constructs, because no types are passed at run-time.

The operational model of Core requires a garbage-collected heap. A heap object, also named closure, contains a data value, a function value, or is a thunk for suspended values. Like a function value, a thunk contains a pointer to its unevaluated code and an environment. The environment is the list of values of the free variables of the code. After a thunk has been evaluated it is overwritten by its freshly computed value. Thus lazy evaluation is implemented.

let bindings and only **let** bindings performs heap allocation. When a **let** binding is evaluated, a closure is allocated for the bound expression. If the bound expression is in weak head normal form (WHNF), a data value or function value is allocated, otherwise a thunk (trivial **let** bindings, i.e., **let** $x = y$ **in** \dots , are eliminated before code generation). Afterwards the body of the **let** is evaluated.

case expressions and only **case** expressions trigger evaluation. The evaluation of a case expression triggers the evaluation of the scrutinised expression to WHNF. The result is compared with the patterns of the alternatives and execution proceeds with the appropriate alternative.

A more detailed description of Core and the objectives of its design is given in [13].

3 Transformation Rules

We define CSE for Core by giving three transformation rules and three assisting rules. To argue that these rules suffice, we show how other transformations implemented in GHC transform a program into a form suitable for our transformation. In Section 4 we analyse in detail under which conditions our transformation rules are sure to reduce the execution costs of the transformed program and we restrict the application of the rules accordingly.

3.1 Dominating Expressions

The suggestive, general transformation rule

$$e'[e, e] \quad \rightsquigarrow \quad \text{let } x = e \text{ in } e'[x, x]$$

certainly cannot be used, because we want only to eliminate a common subexpression when it is safe, that is, costs are not increased. The two occurrences of the expression e may be far apart and the chances that the value of both are needed during the evaluation of the program is low. Worse, a closure is always allocated for e in the transformed program, while this may not be the case for the original program. If the whole expression is inside the body of a λ -abstraction,

then the transformation may lead to the allocation of an unbound number of closures. A closure for e also has a long lifetime: First, it is allocated before the value of e is needed. Then, all occurrences of x in e' reference the closure which can only be deallocated when it is no longer referenced. If the added **let** binding is global, then the closure will even never be deallocated at all.

We decided on a simple implementation that performs no complicated analysis. Hence, similar to Appel (cf. Section 1.2), we only look for common subexpressions when a named expression syntactically dominates another equal expression, that is, we use the transformation rule

$$\text{let } x = e \text{ in } e'[e] \quad \rightsquigarrow \quad \text{let } x = e \text{ in } e'[x] \quad (1)$$

The language Core can also express two other kinds of named, syntactically dominating expression, exemplified by the following expressions:

case tail xs of {ys -> tail xs}

case 6 * 7 of {I# x# -> fib (6 * 7)}

Whereas a programmer hardly writes such code, other program transformations may produce it. In fact, a strictness based transformation transforms **let** $x = e$ **in** e' into **case** e **of** $\{C \ x_1 \dots x_n \rightarrow e'[C \ x_1 \dots x_n / x]\}$, if e' is strict in x and the type of x has only a single data constructor ([15], Section 3.6). All unboxed data types such as **Int#** have exactly one data constructor (see Section 5.2). Hence our transformation additionally applies the following two rules:

$$\text{case } e \text{ of } \{\dots; x \rightarrow e'[e]; \dots\} \rightsquigarrow \text{case } e \text{ of } \{\dots; x \rightarrow e'[x]; \dots\} \quad (2)$$

$$\begin{aligned} & \text{case } e \text{ of } \{\dots; C \ x_1 \dots x_n \rightarrow e'[e]; \dots\} \\ \rightsquigarrow & \text{case } e \text{ of } \{\dots; C \ x_1 \dots x_n \rightarrow e'[C \ x_1 \dots x_n]; \dots\} \end{aligned} \quad (3)$$

Our three rules do not perform an optimisation if the expression e is a variable. Furthermore, rule (3) cannot be applied in that case, because according to Core syntax a variable occurring as an argument of an application cannot be replaced by a constructor application. Sections 3.3 and 5.1 will show that if e is a variable the following reversed transformation rules are important for eliminating nested common subexpressions:

$$\text{let } x = y \text{ in } e'[x] \quad \rightsquigarrow \quad \text{let } x = y \text{ in } e'[y] \quad (1')$$

$$\text{case } y \text{ of } \{\dots; x \rightarrow e'[x]; \dots\} \rightsquigarrow \text{case } y \text{ of } \{\dots; x \rightarrow e'[y]; \dots\} \quad (2')$$

$$\begin{aligned} & \text{case } y \text{ of } \{\dots; C \ x_1 \dots x_n \rightarrow e'[C \ x_1 \dots x_n]; \dots\} \\ \rightsquigarrow & \text{case } y \text{ of } \{\dots; C \ x_1 \dots x_n \rightarrow e'[y]; \dots\} \end{aligned} \quad (3')$$

Whereas it is still not guaranteed (but more probable) that the eliminated expression is computed twice in the original program, the transformation is safe in that the new program allocates no additional closures. The restriction to syntactically dominating expressions renders the transformation more similar to standard CSE in imperative languages. Standard CSE eliminates not all common subexpressions but only those that are already computed before on every execution path.

3.2 Flattening of `let` and `case` Expressions

Considering only syntactically dominating expressions is not as severe a restriction as it seems. First of all, a Core program contains significantly more nested `let` expressions than a normal functional program, because Core requires the arguments of functions to be atoms. Regard the Haskell expression

```
sum [1..1000] + sum [-1000..1] + sum [1..1000]
```

In Core it looks as follows:¹

```
let si =
  let s1 = let l1 = [1..1000] in sum l1 in
    let s2 = let l2 = [-1000..1] in sum l2 in
      s1 + s2
in
  let s3 = let l3 = [1..1000] in sum l3 in
    si + s3
```

Our transformation rules cannot be applied to this program. However, GHC includes several transformations that flatten nested `let` and `case` expressions and thus bring a program into a form more suitable for our transformation. Note that in Core programs every bound variable is unique so that in the following transformation rules variable capture cannot arise.

- float `let` from `let`. ([15], Section 3.4.2)

$$\text{let } x = (\text{let } y = e_y \text{ in } e_x) \text{ in } e \rightsquigarrow \text{let } y = e_y \text{ in } (\text{let } x = e_x \text{ in } e)$$

This transformation is only applied, if e_y is in WHNF or the whole expression is strict in y .

- float `case` from `let`. ([15], Section 3.5.3)

$$\begin{array}{ccc} \text{let } x = \text{case } e \text{ of} & & \text{case } e \text{ of} \\ \quad alt_1 \rightarrow e_1 & & alt_1 \rightarrow \text{let } x = e_1 \text{ in } e' \\ \quad \dots & \rightsquigarrow & \dots \\ \quad alt_n \rightarrow e_n & & alt_n \rightarrow \text{let } x = e_n \text{ in } e' \\ \text{in } e' & & \end{array}$$

This transformation requires e' to be strict in x .

¹ The examples are simplified. In particular, the overloaded numbers of Haskell require the use of dictionaries.

- float **let** from **case**. ([15], Section 3.4.3)

$\text{case } (\text{let } x = e \text{ in } e') \text{ of } \dots \rightsquigarrow \text{let } x = e \text{ in } (\text{case } e' \text{ of } \dots)$

- float **case** from **case**. ([15], Section 3.5.2)

$$\begin{array}{ccc}
 \text{case } \left(\begin{array}{c} \text{case } e \text{ of} \\ \quad alt_1 \rightarrow e_1 \\ \quad \dots \\ \quad alt_n \rightarrow e_n \end{array} \right) \text{ of} & \rightsquigarrow & \begin{array}{c} \text{case } e \text{ of} \\ alt_1 \rightarrow \left(\begin{array}{c} \text{case } e_1 \text{ of} \\ \quad alt'_1 \rightarrow e'_1 \\ \quad \dots \\ \quad alt'_n \rightarrow e'_m \end{array} \right) \\ \dots \\ alt_n \rightarrow \left(\begin{array}{c} \text{case } e_n \text{ of} \\ \quad alt'_1 \rightarrow e'_1 \\ \quad \dots \\ \quad alt'_n \rightarrow e'_m \end{array} \right) \end{array} \\
 alt'_1 \rightarrow e'_1 & & \\
 \dots & & \\
 alt'_n \rightarrow e'_m & &
 \end{array}$$

Join points are used to avoid code duplication ([13], Section 5.1; [15], Section 3.5.2).

Furthermore, GHC performs a full laziness transformation ([15], Section 5.2) which lifts expressions from a λ -abstraction. The full laziness transformation may introduce new application possibilities for CSE, similar to the hoisting transformation implemented by Appel.

3.3 Application of the Transformation Rules: An Example

Consider our **sum** example of the previous subsection. Strictness analysis infers that the expression is strict in all subexpressions and thus all **let** bindings of numbers are transformed into **case** expressions. A subsequent application of the transformations listed in the previous subsection leads to the following program. The data constructor **I#** is part of the unboxed representation of integers (see Section 5.2).

```

let l1 = [1..1000] in
  case (sum l1) of
    I# s1 -> let l2 = [-1000..1] in
      case (sum l2) of
        I# s2 -> case (s1 +# s2) of
          I# si -> let l3 = [1..1000] in
            case (sum l3) of
              I# s3 -> case (si +# s3) of
                s -> I# s

```

Applying rule (1) removes the second occurrence of `[1..1000]`:

```
let l1 = [1..1000] in
  case (sum l1) of
    I# s1 -> let l2 = [-1000..1] in
      case (sum l2) of
        I# s2 -> case (s1 +# s2) of
          I# si -> let l3 = l1 in
            case (sum l3) of
              I# s3 -> case (si +# s3) of
                s -> I# s
```

Rule (1') renames variable `l3` to `l1` in the body of the `let` binding so that rule (3) can be applied:

```
let l1 = [1..1000] in
  case (sum l1) of
    I# s1 -> let l2 = [-1000..1] in
      case (sum l2) of
        I# s2 -> case (s1 +# s2) of
          I# si -> let l3 = l1 in
            case I# s1 of
              I# s3 -> case (si +# s3) of
                s -> I# s
```

The existing transformations of GHC finally yield the simplified program:

```
let l1 = [1..1000] in
  case (sum l1) of
    I# s1 -> let l2 = [-1000..1] in
      case (sum l2) of
        I# s2 -> case (s1 +# s2) of
          I# si -> case (si +# s1) of
            s -> I# s
```

3.4 Swapping of Independent `let` and `case` Expressions

A disadvantage of the restriction to textually dominating expressions is that our transformation rules may not be applicable because of the accidental order of independent `let` or `case` expressions. The program

```
let y = 42 in (let x = 42 + 1 in f x y)
```

is transformed whereas

```
let x = 42 + 1 in (let y = 42 in f x y)
```

is not. Unfortunately the independence of several `let` expressions cannot be made explicit in the Core language; similarly for `case` expressions.

It is possible to extend our implementation of CSE, that we present in Section 5, to eliminate common subexpressions even in such cases. We did not yet do so, because we do not expect this case to occur very often.

4 Analysis of the Effects of the Transformation

Here we discuss application conditions for our rules which assure that common subexpression elimination reduces costs. For lazy functional languages the costs of major interest are run-time, total heap allocation, maximal heap residency, that is the maximal space required by live objects on the heap at one time, and size of the program code. Finally we take up Santos' idea of restricting the type of eliminated subexpressions to avoid space leaks.

The observant reader will notice that our transformation rule (1) is just the inverse of another well-known transformation: inlining. Inlining replaces occurrences of a `let`-bound variable by its defining expression to remove function-call overhead and to expose the defining expression to local context information and thus to increase the possibility of other transformations being applied.

Transformations that are inverses of each other occur quite often in GHC, it applies for example a `let` floating inward and a `let` floating outward transformation ([15], Sections 5.1 and 3.4). For determining the conditions for applying CSE we just have to reverse the known arguments for inlining ([15], Section 6; [13], Section 4).

4.1 Run-Time and Code Size

We have to distinguish two kinds of common subexpressions. If the expression concerned is a WHNF, that is, a variable, a literal, a constructor application, or a λ -abstraction, then CSE cannot save execution time, because the expression is already evaluated. Replacing the expression by a variable may even increase run-time, because an additional indirection is introduced ([15], Section 3.2.3). Eliminating a common non-WHNF saves execution time, if the value of at least two of its occurrences in the original program are needed.

A special case is a common subexpression that is the right hand side of a `let` binding. Replacing this expression by a variable yields a trivial binding of the form `let x = y in ...` which is later eliminated by an existing transformation. Thus run-time and code size are reduced.

In addition to decreasing run-time, CSE can also decrease program size. It should however be noted that eliminating small expressions like constructor applications has probably no effect on program size.

4.2 Avoidance of Space Leaks

Whereas the transformation may only decrease total heap usage, it may considerably influence heap residency both positively and negatively. The latter case is demonstrated in Section 1 by an example and has to be avoided.

Eliminating common subexpressions that are in WHNF does not increase heap residency, but we have seen that except for right hand sides of `let` bindings their elimination is not advantageous.

Santos suggests to eliminate only expressions of certain types ([15], Sections 8.5.11 and 8.6.2), similar to his approach to the full laziness transformation. If

the values of the expressions only take a small, fixed amount of space, then the increased lifetime of the values on the heap should hardly matter.

Santos considers only types that are not recursive and do not contain recursive types as subcomponents. This restriction is not sufficient. A partially evaluated expression of a structured type may require an unbound amount of heap space, because it may contain an arbitrary number of (linked) thunks. The following example illustrates this.

```
f 0      = (0, 0)
f (n+1) = case (f n) of (l, m) -> (0, m+1)

let z = f 1000 in
  case z of
    (x, y) -> -> e[f 1000]
```

After **z** has been evaluated by the **case** expression, **y** is represented by 1001 thunks denoting the unevaluated expression $0 + 1 + 1 + \dots + 1$. If neither **y** nor **z** occur in the body of the case expression this space can immediately be reclaimed by the garbage collector. However, if CSE replaces the second occurrence of **f 1000** by **z**, then this space cannot be reclaimed and thus heap residency is increased considerably.

Hence we see that a partially evaluated expression is certain to require only a small, fixed amount of space, if and only if its WHNF is already its normal form and it is not a function, whose environment may refer to arbitrary large data structures. Examples are all expressions of type **Bool**, **Char**, **Int**, and **Float**. We call such types *safe*.

To assure that a common expression is evaluated before sharing takes place, we apply rule (1) only if the **let** expression is strict in the **let**-bound variable. GHC applies a **let** to **case** transformation after all other transformations which guarantees that the **let**-bound expression is evaluated before the body of the **let** expression.

Eliminating only those non-WHNFs that are evaluated and of safe type assures that space leaks cannot occur. The transformation is still more general than standard CSE, because the subexpressions of the eliminated expression may be of arbitrary type.

5 Implementation of the Transformation

The restriction of the transformation to the elimination of subexpressions which are **let**- or **case**-bound in the same scope permits a single recursive traversal of the Core program. The named, dominating expressions are collected in a tree data structure with logarithmic lookup time. The comparison of two expressions modulo α -conversion by recursive descent is nearly always decided after examination of the top of the two expressions. Thus, compared to the rest of the compiler, the time spend on the transformation is not noticeable in practise.

The implementation is available from the author.

5.1 Recognition of Common Subexpressions

For each expression we first transform its subexpressions and then test, if the whole transformed expression occurred before. This order is necessary to gain a cumulative effect and it thus assures that the transformation is idempotent, that is, after being applied once a second application has no effect. For example we get

$$\begin{aligned} & \text{let } x = 3 \text{ in let } y = 2+3 \text{ in } 2+3+4 \\ \rightsquigarrow & \text{let } x = 3 \text{ in let } y = 2+x \text{ in } y+4 \end{aligned}$$

while doing lookup first and then recursive transformation leads to

$$\rightsquigarrow \text{let } x = 3 \text{ in let } y = 2+x \text{ in } 2+x+4$$

Section 4 showed that the transformation rules may only be applied under a given condition. Only common subexpressions that are evaluated non-WHNFs of safe type or that are WHNFs which occur as the right hand side of a `let` binding are eliminated.

This condition may restrict our transformation more than desirable. Consider the example of Section 3.3. The subexpression `[1..1000]` is not to be eliminated because it is not of safe type. In consequence rules (1') and (3) cannot be applied either. Thus the program is not optimised.

To solve this problem the function that transforms a subexpression returns two expressions. The first is the result of CSE under observance of the given condition, but the second is the subexpression with all common subexpressions eliminated. The latter version is memorised in the mentioned tree data structure and is thus the basis for comparison of expressions.

Hence in the example 13 is eliminated although `[1..1000]` is not. The reader may also assure himself that the expression

$$\text{sum } [1..1000] + \text{sum } [-1000..1] + \text{prod } [1..1000]$$

is not modified by the transformation.

5.2 Considerations Specific to the Glasgow Haskell Compiler

Unboxed Values. Implementations of non-strict languages like Haskell process so called boxed values, that is pointers into the heap that either point to a delayed computation or the actual value. In order to improve efficiency Core is also able to handle actual values directly, which are named unboxed values and are distinguished by their types. These unboxed values have been added carefully, so that — although they introduce explicit strictness into the otherwise non-strict language — they do not invalidate any program transformation, provided the produced code observes the following two restrictions: no polymorphic function is applied to an expression of unboxed type and every expression of unboxed type which appears as the argument of an application or as the right-hand side of a binding is in head normal form, that is, a literal, an application of an unboxed constructor, or a variable (see [12] for details).

Fortunately, the transformation handles unboxed data types correctly: the types of arguments of (polymorphic) functions are not changed and the transformation never turns an expression which is in head normal form into one that is not.

We can also add simple unboxed data types like `Bool#`, `Char#`, `Int#`, `Float#` and `Double#` to the list of safe types. However, since the right-hand side of a `let` binding that is of unboxed type has to be in head normal form, only rules (2) and (3) will benefit from this extension.

Uses Type System. Based on ideas from linear logic the type system of Core has been extended in version 2.01 by so called uses, which record when a value is used (accessed) at most once. This knowledge permits to avoid update of closures which are not accessed again, enables update in place of data structures whose old value is no longer needed, and may guide program optimisations, especially safe situations for inlining can be determined. Uses are attached to types. The use 1 of a type indicates that its values are used at most once, while the use ω indicates that the values of the type may be used any number of times. A program transformation has to produce Core programs that respect the use type system (see [6] for details).

The usage information is hardly useful for CSE. Only if a `let` or `case` bound variable has use 1, then common subexpression elimination very probably saves execution time iff after the transformation a repeated uses type inference yields use ω for the variable.

Our implementation of common subexpression evaluation violates the uses type system. Consider the following transformation:

$$\text{let } x = 1+2 \text{ in let } y = 1+2 \text{ in } x+y \quad \rightsquigarrow \quad \text{let } x = 1+2 \text{ in } x+x$$

In the left expression the type of the variable `x` may have use 1, and in the right expression it should have use ω . There does not seem to be any good solution to this problem. The transformation may either be restricted to variables of use ω , or the use of all variables used for subexpression elimination is set to ω , or the program has to be type checked again after the transformation. Currently this does not matter since version 2.08 of GHC does not yet make use of the use information for code generation or any program transformation.

Cost Centres. Finally, programs that are compiled for profiling are annotated with cost centres. Not respecting these annotations, that is, moving subexpressions from the scope of one cost centre to another, does not change the semantics of the program, but it invalidates the profiling measurements. Sansom and Peyton Jones suggest to curtail transformations to never move costs across a cost centre annotation. This means however, that observing a program (annotating it with cost centres) influences the behaviour to be observed! Alternatively, a subexpressions that is moved out of the scope of a cost centre can be annotated with its original cost centre. Nonetheless this usually moves a small cost to another cost centre and it evidently complicates every transformation (see [14] for details).

Our transformation eliminates subexpressions without caring about the scope of cost centres. It is not even clear how the cost of a common subexpression could be shared between cost centres. Cost centre annotations also limit the applicability of the transformation since terms which differ only by their annotation are regarded as different.

6 Measurement of the Effects of the Transformation

The optimisation option `-O2` of GHC turns on a long sequence of optimisations. We inserted our transformation three times into this sequence. Our transformation is invoked for the first time after the strictness analysis, because we observed in Sections 3 and 4 that the applicability of CSE is increased by various `let` and `case` floating transformations and by strictness analysis. As standard for comparison we use the same sequence of optimisations without CSE.

The objects of our comparison are the `sum` example of Section 3.2 and nine programs from the Glasgow nofib test suite, version 2.5 [7]. The latter are real applications, that is, applications that were not designed as benchmarks but to solve particular problems, for instance text compression (`compress`) and Monte Carlo photon transport (`gamteb`).

Table 1 shows the results of our measurements. The size of each program's source is given in number of lines. Afterwards the maximum of the number of common subexpressions that were recognised in each of the three passes of CSE is given. Obviously common subexpressions that are not eliminated are usually rediscovered in latter CSE passes. The subsequent number is the sum of common subexpressions eliminated by all three passes. Remember that all type information is dropped by GHC during code generation. To obtain meaningful numbers those common subexpressions which are just applications of variables to types are not counted.

The last four columns show the measured effects of CSE on costs, that is, run-time, the total amount of heap allocated, maximal heap residency, and code size. These numbers were obtained by using the profiling facilities of GHC. For all times we took the minimum of at least six runs. All measurements are given as differences in per cent between the numbers for the program compiled with CSE and those for the program compiled without. Horizontal lines signify that no meaningful numbers could be gained because the run-time was too short.

Both the numbers of recognised and of eliminated subexpressions have a very weak relationship with the size of the respective program. Most programs have few common subexpressions and even fewer which can be eliminated safely.

The measurements prove that our `sum` example profits considerably from the transformation. Both run-time and total heap allocation are reduced by the expected one third. However, the effect on the real-world programs of the nofib suite is disappointing. Only the run-time of `reptile` is slightly improved. Additionally the code size of all programs is slightly decreased.

To evaluate the effect of the condition for avoiding space leaks that was introduced in Section 4.2, Table 2 shows the measurements of an older version of our

program	lines	max. recogn.	elim.	time	total alloc.	max. residency	code
sum	2	3	1	-30 %	-33.3 %	0 %	-0.08 %
compress	320	3	0	0 %	0 %	0 %	0 %
fulsom	1357	93	22	-0.17 %	0 %	0 %	-0.55 %
gamteb	718	30	30	0 %	+0.02 %	-0.02 %	-0.08 %
grep	356	124	24	–	0 %	–	-0.59 %
lift	2033	97	5	–	0 %	0 %	-0.11 %
pic	544	29	0	0 %	0 %	0 %	0 %
prolog	539	31	5	–	-0.25 %	+0.5 %	-0.27 %
reptile	1522	115	31	-0.8 %	0 %	0 %	-0.34 %
rsa	74	7	0	0 %	0 %	0 %	0 %

Table 1. Eliminated common subexpressions and effect on costs

transformation which does not implement the condition. The increase of maximal heap residency of **prolog** exhibits the lack of the condition. Beware that the two tables are not directly comparable, because the old version did not implement rules (2') and (3') and the transformation was only applied once instead of three times. Nonetheless, the larger impact of the old transformation proves that the space safety condition does prohibit useful transformations. An analysis of the program **rsa** reveals that only two common subexpressions, **int2Integer# 2** and **int2Integer# 128**, are responsible for the decrease of run-time by 1.7 %. These expressions are not eliminated by the safe transformation because the strictness analyser could not infer the fact that they are demanded.

program	elim.	time	total alloc.	max. residency
sum	2	-26 %	-33.4 %	+0.001 %
compress	0	–	–	–
fulsom	20	+0.3 %	-0 %	-0.01 %
gamteb	15	+1.4 %	+0.02 %	-0.3 %
grep	17	–	-3.1 %	–
lift	7	0 %	-0.09 %	-0.03 %
pic	6	+0.5 %	+0.4 %	-0.01 %
prolog	6	0 %	-0.3 %	+6 %
reptile	32	-1.7 %	-0.2 %	-0.02 %
rsa	4	-1.7 %	-0.24 %	-2.5 %

Table 2. CSE without considerations for space leaks

7 Lack of Common Subexpressions

Unfortunately our transformation optimises only our demonstration example but none of our real-world programs. Obviously the elimination of a single common subexpressions cannot generally be expected to have an effect as large as in the **sum** example. The measurements suggest that the reason for the lack of

speedup is that the real-world programs have few common subexpressions and even fewer which can be eliminated safely. We suppose that the lack of common subexpressions is also the cause of Appel's disappointing results of CSE.

Common subexpressions may either already exist in the source program or be introduced by the compiler. A programmer avoids repeated computations. The purpose of CSE for imperative languages is to eliminate repeated computations that are introduced by the compiler. The classical example is array indexing. However, first, arrays are seldom used in functional programs and, second, they are implemented by calling C-functions which are not reachable by transformations working on Core level. Our measurements suggest that GHC introduces few repeated computations. The only significant exception are overloaded numerical constants. In Haskell a constant `42` has to be replaced by `fromInteger 42` by the compiler. Nonetheless in most cases the existing specialisation and full-laziness transformation optimise these hidden common subexpressions sufficiently.

The idea comes up that the existence of a CSE phase may cause programmers to write their programs without worrying about repeated computations. A programming style that encourages the use of abstract data types may also lead to an increase of common subexpressions. Similar to the example of array indexing the limited interface of an abstract data type would prevent a programmer from avoiding duplicated computations himself. Furthermore, compiler phases could be simplified or improved if they were permitted to create common subexpressions. For example, in deforestation code duplication is a problem ([5]).

However, lazy functional languages decouple program structure from data space allocation and control flow. Consequently, as we discussed in Chapters 3 and 4, CSE has to be restrained by complex conditions to avoid a decrease of performance. Hence CSE cannot be transparent, that is neither a functional programmer nor an implementor of an optimisation phase can easily assure that common subexpressions introduced by him will be eliminated.

8 Conclusion

In this paper we have developed a version of CSE for a lazy functional language, implemented it by adding it to GHC, and measured its effects on real-world programs.

Generally, the development of CSE in Section 3 demonstrates the importance of close interaction of transformations. Several existing optimisations increase the opportunities for applying CSE. Especially an improved strictness analysis would improve our transformation. Because the effect of CSE depends highly on the operational semantics of Core it is unfortunately difficult to transfer the discussion of Section 4 to another implementation of a lazy functional programming language. In fact, Section 5.2 proves that detailed knowledge of GHC specific properties like the underlying abstract machine (the STG-machine), unboxed data types, the use type system, cost centres, etc. is required for a real implementation.

The danger of creating space leaks are the same for CSE as for the full laziness transformation. We showed that the full laziness transformation implemented in GHC is not safe and gave a safe alas more restrictive transformation condition. Both transformations would profit from the development of a less restrictive safety condition.

Our implementation is fast and simple. Unfortunately it optimises only our demonstration example but none of our real-world programs. There is still room for improving the transformation. We already mentioned that possibly the space safety conditions could be partially lifted. More use of the context of common subexpressions could be made than only handling the right hand sides of `let` bindings specially. Cross-module CSE could enable the replacement of expressions by variables defined in imported modules. Finally, analysis techniques could be developed, which ascertain that repeated expressions are actually evaluated repeatedly.

However, the fundamental reason why our transformation is unsuccessful is that our test programs have only few common subexpressions. In Chapter 7 we gave good reasons for common subexpressions generally being uncommon in lazy functional languages. Note that the full laziness transformation is probably more effective, because expressions that can be lifted from a λ -abstraction are more difficult to spot and hence to avoid for a programmer than common subexpressions.

We claim that whereas CSE is an important optimisation for imperative languages it is not suitable for reducing the run-time of lazy functional programs. It remains to be investigated if the transformation could be used for reducing code size and thus also code generation time by eliminating large common WHNFs which were introduced by inlining but which did not enable other transformations.

Acknowledgement

I thank all members of the implementation of functional programming languages group at Aachen for numerous discussions and Simon Peyton Jones for answering many questions about GHC. I am grateful to one of the referees for suggesting the title of this paper.

References

- [1] A Aho, R Sethi, and J Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [2] Adrew W Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- [3] Olaf Chitil. Adding an optimisation pass to the Glasgow Haskell compiler. <http://www-i2.informatik.rwth-aachen.de/~chitil/>, November 1997.
- [4] The Glasgow Haskell compiler. <http://www.dcs.gla.ac.uk/fp/software/ghc/>.
- [5] Simon D Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, University of Glasgow, September 1995.

- [6] Christian Mossin, David N Turner, and Philip Wadler. Once upon a type. Technical Report TR-1995-8, University of Glasgow, 1995. Extended version of [?].
- [7] Will Partain. The Nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, Workshops in Computing, pages 195–202. Springer-Verlag, 1992.
- [8] Will Partain. How to add an optimisation pass to the Glasgow Haskell compiler (two months before version 0.23). Part of the GHC 0.29 distribution, October 1994.
- [9] John Peterson, Kevin Hammond, et al. Report on the Programming Language Haskell, version 1.4. <http://www.haskell.org>, 1997.
- [10] Simon L Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- [11] Simon L Peyton Jones. Implementing lazy functional languages on stock hardware: the Spinless Tagless G-machine. *J. Functional Programming*, 2(2):127–202, 1992.
- [12] Simon L Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Conf. on Functional Programming Languages and Computer Architecture*, pages 636–666, 1991.
- [13] Simon L Peyton Jones and André L M Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 1997.
- [14] Patrick M Sansom and Simon L Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *22nd ACM Symposium on Principles of Programming Languages*, January 1995.
- [15] André L M Santos. *Compilation by transformation in non-strict functional languages*. PhD thesis, University of Glasgow, July 1995.
- [16] D Tarditi, G Morrisett, P Cheng, C Stone, R Harper, and P Lee. TIL: A type-directed optimizing compiler for ML. In *Sigplan Symposium on Programming Language Design and Implementation*, 1996.

WITH-Loop-Folding in SAC - Condensing Consecutive Array Operations

Sven-Bodo Scholz

Dept of Computer Science
University of Kiel

24105 Kiel, Germany sbs@informatik.uni-kiel.de

Abstract. This paper introduces a new compiler optimization called *WITH-loop-folding*. It is based on a special loop construct, the *WITH-loop*, which in the functional language SAC (for Single Assignment C) serves as a versatile vehicle to describe array operations on an element-wise basis. A general mechanism for combining two of these *WITH-loops* into a single loop construct is presented. This mechanism constitutes a powerful tool when it comes to generate efficiently executable code from high-level array specifications. By means of a few examples it is shown that even complex nestings of array operations similar to those available in APL can be transformed into single loop operations which are similar to hand-optimized *WITH-loop* specifications. As a consequence, the way a complex array operation is combined from primitive array operations does not affect the runtime performance of the compiled code, i.e., the programmer is liberated from the burden to take performance considerations into account when specifying complex array operations.

1 Introduction

The suitability of functional languages for numerical applications critically depends on the compilation of array operations into efficiently executable code. In the context of lazy functional languages, e.g. HASKELL[13] or CLEAN[20], it seems to turn out that this can only be achieved with strict arrays and if single threading for most of the array operations can be guaranteed statically [12, 23]. Whereas the former restriction in most cases does not affect the style of programming, the latter requires the programmer to make opportunities for destructive array updates explicit to the compiler by the introduction of uniqueness types [24] or state monads [17].

In contrast to lazy languages, SISAL[18] demonstrates that strict languages can be compiled into efficiently executable code without forcing the programmer to think about single threading of array operations [8]. However, SISAL allows for low-level specifications only. Neither the expressive power of polymorphism, higher-order functions, and partial applications nor any high-level array operations are supported. As a consequence, despite a new syntax SISAL offers few benefits in terms of expressiveness when compared with imperative languages, e.g. FORTRAN[26] or C[16].

The development of the strict functional language SAC[21] constitutes a different approach. Although SAC as well does not yet support higher-order functions nor partial applications it differs from SISAL in three respects. Most important, SAC offers substantial support for the specification of high-level array operations, i.e., it provides array operations similar to those available in APL[14] as well as a special loop construct, the WITH-loop, which allows array operations to be specified element-wise. All of these language constructs are designed to facilitate the specification of array operations that can be applied to arrays of arbitrary dimensionalities and shapes. Furthermore, the syntax of SAC sticks as close as possible to that of C which does not only facilitate the compilation process but might also increase the acceptance of SAC by programmers that are used to program in C. Last but not least, SAC provides states, state-modifications, and I/O-operations [11] which are incorporated safely into the functional framework by means of uniqueness typing [24].

The SAC compiler is supported by a type inference system based on a hierarchy of array types which allows dimension-independent SAC specifications to be specialized to shape-specific C programs. Although this concept leads to code which can be executed very efficiently in terms of runtime and space consumption [22], the compilation of array operations in the current compiler version is still implemented straightforwardly. Each array operation is compiled into a separate piece of code, i.e., nestings of array operations are broken up into sequences of single array operations by the introduction of temporary arrays. Since there is no optimization between two or several consecutive array operations, the way a complex array operation is composed from the language constructs available has a strong influence on the code generated. Different specifications lead to the introduction of different temporary arrays and thus to different runtime overheads. Redundancies within consecutive array operations are not detected and thus have a direct impact on runtimes as well. As a consequence, the programmer has to know about the implementation when it comes to writing programs that can be compiled to efficiently executable code. Since this runs counter to the idea of functional programming, which is to liberate the programmer from low level concerns, an optimization is needed that tries to eliminate temporary arrays as well as redundancies of consecutive array operations whenever possible.

The basic idea is to use WITH-loops as a universal representation for array operations and to develop a general transformation scheme which allows to transform two consecutive WITH-loops into a single one. With this mechanism, called WITH-loop-folding, any nesting of primitive array operations can be transformed stepwise into a single loop construct which contains an element-wise specification of the resulting array.

The consequences of this optimization are far-reaching:

- The code generated for any complex array operation specified in SAC becomes invariant against the way it is composed from more primitive operations, i.e., the programmer is not concerned with the low level details any more.

- In combination with the application of other, well-known optimizations (e.g. constant-folding, loop unrolling, etc.) most of the redundancies which would result from a straightforward compilation can be eliminated.
- All primitive array operations of SAC that are similar to those available in APL can be defined through WITH-loops and are implemented via the standard library rather than by the compiler itself. The only array operation implemented by the compiler is the WITH-loop.
- Since the primitive array operations are defined in the standard library, they can be adjusted to the programmers needs easily.

The paper is organized as follows: While the following section gives a short introduction into the basic language concepts of SAC, Section 3 presents the basic WITH-loop-folding mechanism. An extended example is discussed in Section 4. It demonstrates nicely how even complex nestings of array operations can be folded into a simple loop construct. Some implementation issues are discussed in Section 5. These include some restrictions which are imposed on WITH-loop-folding in order to guarantee its statical applicability as well as some performance considerations. Section 6 points out the relationship of WITH-loop-folding to other optimization techniques. Some concluding remarks finally can be found in Section 7.

2 SAC - A Short Introduction

SAC is a strict, purely functional language whose syntax in large parts is identical to that of C. In fact, SAC may be considered a functional subset of C extended by high-level array operations which may be specified in a shape-invariant form. It differs from C proper mainly in that

- it rules out global variables and pointers to keep functions free of side-effects,
- it supports multiple return values for user-defined functions, as in many dataflow languages[3, 1, 6],
- it supports high-level array operations, and
- programs need not to be fully typed.

With these restrictions / enhancements of C a transformation of SAC programs into an applied λ -calculus can easily be defined. The basic idea for doing so is to map sequences of assignments that constitute function bodies into nestings of LET-expressions with the RETURN-expressions being transformed into the innermost goal expressions. Loops and IF-THEN-ELSE statements are transformed into (local) LETREC- expressions and conditionals respectively. For details see [21].

An array in SAC is represented by a shape vector which specifies the number of elements per axis, and by a data vector which lists all entries of the array.

For instance, a 2×3 matrix $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ has shape vector $[2, 3]$ and data vector $[1, 2, 3, 4, 5, 6]$. The set of legitimate indices can be directly inferred from the shape vector as

$$\{[i_1, i_2] \mid 0 \leq i_1 < 2, \quad 0 \leq i_2 < 3\}$$

where $[i_1, i_2]$ refers to the position $(i_1 * 3 + i_2)$ of the data vector. Generally, arrays are specified as expressions of the form

$$\text{reshape}(\text{shape_vector}, \text{data_vector})$$

where *shape_vector* and *data_vector* are specified as lists of elements enclosed in square-shaped brackets. Since 1-dimensional arrays are in fact vectors, they can be abbreviated as

$$[v_1, \dots, v_n] \equiv \text{reshape}([n], [v_1, \dots, v_n]) \quad .$$

Several primitive array operations similar to those in APL, e.g. **dim** and **shape** for array inspection, **take**, **drop**, **cat**, and **rotate** for array manipulation, and **psi** for array element / subarray selection, are made available. For a formal definition of these see [21, 22].

All these operations have in common that they, in one way or another, affect all elements of the argument array(s) in the same way. To have a more versatile language construct at hands which allows to specify array operations dimension independently on arbitrary index ranges, SAC also supports so called WITH-loops. They are similar to array comprehensions as known from HASKELL or CLEAN as well as to the FOR-loops in SISAL.

The syntax of WITH-loops¹ is outlined in Fig. 1. Basically, they consist of

$$\begin{aligned} \textit{WithExpr} &\Rightarrow \textbf{with} \ (\textit{Generator}) \ \textit{Operation} \\ \textit{Generator} &\Rightarrow \textit{Expr} \leq \textit{Identifier} \leq \textit{Expr} \\ \textit{Operation} &\Rightarrow [\{ \textit{LocalDeclarations} \}] \textit{ConExpr} \\ \textit{ConExpr} &\Rightarrow \textbf{genarray} \ (\textit{Expr} \ , \ \textit{Expr} \) \\ &\quad | \ \textbf{modarray} \ (\textit{Expr} \ , \ \textit{Expr} \ , \ \textit{Expr} \) \end{aligned}$$

Fig. 1. WITH-loops in SAC.

two parts: a generator part and an operation part. The generator part defines lower and upper bounds for a set of index vectors and an 'index variable' which represents a vector of this set. The operation part specifies the operation to be performed on each element of the index vector set. Two different kinds of

¹ Actually, only a restricted form of the WITH-loops in SAC is presented here which suffices to explain the WITH-loop-folding mechanism. An extension to the full-featured WITH-loops is straightforward and will not be further addressed.

operation parts for the generation of arrays are available in SAC (see *ConExpr* in Fig. 1). Their functionality is defined as follows:

Let *shp* and *idx* denote SAC-expressions that evaluate to vectors, let *array* denote a SAC-expression that evaluates to an array, and let *expr* denote an arbitrary SAC-expression. Then

- **genarray**(*shp*, *expr*) generates an array of shape *shp* whose elements are the values of *expr* for all index vectors from the specified set, and 0 otherwise;
- **modarray**(*array*, *idx*, *expr*) returns an array of shape **shape**(*array*) whose elements are the values of *expr* for all index vectors from the specified set, and the values of *array*[*idx*] at all other index positions.

To increase program readability, local variable declarations may precede the operation part of a **WITH**-loop. They allow for the abstraction of (complex) subexpressions from the operation part.

3 Folding WITH-Loops

The key idea of **WITH**-loop-folding is to provide a universal mechanism that transforms functional compositions of array operations into single array operations which realize the functional composition on an element-wise basis. This avoids the creation of temporary arrays as well as redundancies within consecutive operations. Since most of the primitive array operations of SAC can be specified as **WITH**-loops the folding scheme not only can be applied to user-defined **WITH**-loops but serves as tool for the optimization of nested primitive array operations as well.

The most basic situation for such a transformation is the composition of two **WITH**-loops which simply map functions **f** and **g** to all elements of an array. From the well known fact that $\text{map } \mathbf{f} \circ \text{map } \mathbf{g} \equiv \text{map } (\mathbf{f} \circ \mathbf{g})$ we directly obtain:

Let **A** be an array of arbitrary shape with elements of type τ and let **f** and **g** be functions of type $\tau \rightarrow \tau$. Then

```
{...
  B = with( 0*shape(A) <= i_vec <= shape(A)-1)
        modarray( A, i_vec, f( A[i_vec] ));
  C = with( 0*shape(B) <= j_vec <= shape(B)-1)
        modarray( B, j_vec, g( B[j_vec] ));
...}
```

can be substituted by

```
{...
  C = with( 0*shape(A) <= j_vec <= shape(A)-1)
        modarray( A, j_vec, g( f( A[j_vec] ) ));
...}
```

provided that **B** is not referenced anywhere else in the program which can be checked statically since the scope of **B** is well defined (for details see [21]).

However, as soon as the WITH-loop is used in a more general fashion the transformation scheme becomes more complicated since it does not correspond to a simple mapping function anymore. In the following we want to generalize the above scheme in three respects:

1. the WITH-loops to be folded may have non-identical index sets in their generator parts;
2. the second WITH-loop may contain several references to the array defined by the first one;
3. the access(es) to the array defined by the first WITH-loop may be non-local, i.e., instead of $B[j_vec]$ expressions of the form $B[I_op(j_vec)]$ are allowed where I_op projects index vectors to index vectors.

An example that covers all these aspects and therefore will be referred to throughout the whole section is the piece of a SAC program given in the upper part of Fig.2. It consists of two WITH-loops which successively compute vectors B

```
{...
  B = with( [0]<= i_vec <= [39])
        modarray( A, i_vec, A[i_vec] + 3);
  C = with( [20]<= j_vec <= [79])
        modarray( B, j_vec, B[j_vec] + B[j_vec - [10]]);
... }
```

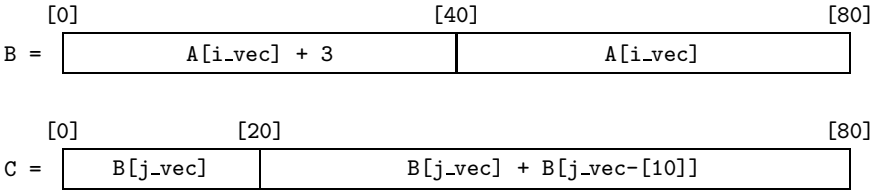


Fig. 2. Two successive WITH-loops with overlapping index ranges and multiple references.

and C from a given vector A . Each of these vectors consists of 80 integer numbers. While the first WITH-loop defines B to differ from A in that the first 40 elements are increased by 3, the second WITH-loop defines C to differ from B in that the last 60 elements of C are computed as the sum of two elements of B , the actual one and the one that is located at the actual index position minus 10.

A graphical representation of these WITH-loops is given in the lower part of Fig.2: Each horizontal bar represents all elements of the vector named to the left of it. The index vectors on top of the bars indicate the positions of the respective elements within the bars. The SAC expressions annotated in the bars define how the vector elements are computed from the elements of other vectors. Since different computations are required in different index vector ranges the bars are divided up by vertical lines accordingly.

Instead of first computing B from A then C from B, the array C can be computed from the array A directly. This operation requires four index ranges of C to be treated differently, as depicted in Fig.3.

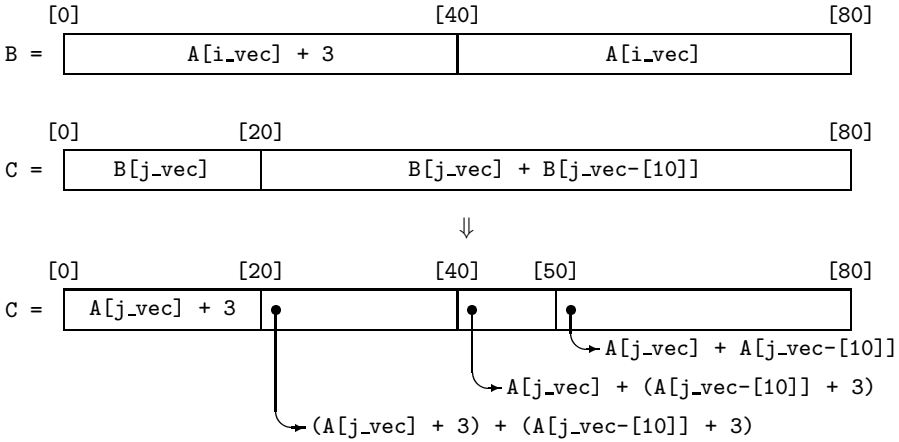


Fig. 3. Substituting two successive array modifications by a single one.

Due to the four index ranges whose array elements have to be treated differently the resulting operation cannot be expressed by a single WITH-loop. Therefore, at least at the level of compilation, a more general version of WITH-loops has to be introduced. It generalizes the possibility to specify a single operation on a subset of array indices in that it allows different operations on arbitrary partitions of array indices to be specified. Throughout this paper the following notation will be used to denote such internal WITH-loop representations:

Let S denote the set of legal indices for an array of shape $[\mathbf{s}_1, \dots, \mathbf{s}_n]$, i.e., $S := \{ [\mathbf{i}_1, \dots, \mathbf{i}_n] \mid \forall j \in \{1, \dots, n\} \ 0 \leq \mathbf{i}_j < \mathbf{s}_j \}$. Furthermore, let $\text{IV}_1, \dots, \text{IV}_m$ be a partition of S and let $\text{Op}_1(\mathbf{i_vec}), \dots, \text{Op}_m(\mathbf{i_vec})$ be expressions that evaluate to data structures of type τ if $\mathbf{i_vec} \in S$. Then

$$A = \text{internal_with}(\text{i_vec}) \left\{ \begin{array}{l} IV_1 : \text{Op}_1(\text{i_vec}) \\ \vdots \\ IV_m : \text{Op}_m(\text{i_vec}) \end{array} \right\}$$

defines an array \mathbf{A} of shape $[\mathbf{s}_1, \dots, \mathbf{s}_n]$ with element type τ where

$$A[\mathbf{i_vec}] := \text{Op}_j(\mathbf{i_vec}) \Leftrightarrow \mathbf{i_vec} \in \mathbb{IV}_j. \quad ^2$$

With this notation at hands, our example problem can be specified as follows:

² Note, that \mathbf{A} is well defined since $\text{IV}_1, \dots, \text{IV}_m$ is a partition of S

Find a transformation scheme which transforms

```

B = internal_with( i_vec) {
    [0]→[39] : A[i_vec] + 3
    [40]→[79] : A[i_vec]
}
C = internal_with( j_vec) {
    [0]→[19] : B[j_vec]
    [20]→[79] : B[j_vec] + B[j_vec - [10]]
}

```

into

```

C = internal_with( j_vec) {
    [0]→[19] : A[j_vec] + 3
    [20]→[39] : (A[j_vec] + 3) + (A[j_vec - [10]] + 3)
    [40]→[49] : A[j_vec] + (A[j_vec - [10]] + 3)
    [50]→[79] : A[j_vec] + A[j_vec - [10]]
}

```

where $[\text{from}_1, \dots, \text{from}_n] \rightarrow [\text{to}_1, \dots, \text{to}_n]$ denotes the set of index vectors $\{[i_1, \dots, i_n] \mid \forall j \in \{1, \dots, n\} : \text{from}_j \leq i_j \leq \text{to}_j\}$.

The basic idea to a general solution is to formulate a scheme which stepwise replaces all references to “temporary arrays” (the array B in our example) by their definitions. Once all references to a temporary array are replaced, the WITH-loop by which it is defined does not contribute to the program’s result anymore and thus can be eliminated.

The central rule which defines the replacement of array references by their definitions is specified in Fig.4.

For an application of the rule, it is anticipated that all WITH-loops have been replaced by equivalent internal WITH-loop constructs beforehand. The upper part of Fig.4 shows the most general situation in which a replacement can be made: two array definitions are given within one scope (e.g. a function body) whose second one refers to (an) element(s) of the first one. For the sake of generality it is assumed that the first array, named A, is defined by m expressions $\text{Op}_{1,1}(\text{i_vec})$, ..., $\text{Op}_{1,m}(\text{i_vec})$ on m disjoint sets of index vectors $\text{IV}_{1,1}$, ..., $\text{IV}_{1,m}$, and that the second array B is defined by n expressions $\text{Op}_{2,1}(\text{j_vec})$, ..., $\text{Op}_{2,n}(\text{j_vec})$ on sets of index vectors $\text{IV}_{2,1}$, ..., $\text{IV}_{2,n}$. Furthermore, we assume that $\text{Op}_{2,i}(\text{j_vec})$ is an expression that contains a subexpression $A[\text{I_op}(\text{j_vec})]$ as indicated by $\vdash \dots A[\text{I_op}(\text{j_vec})] \dots \vdash$. If $A[\text{I_op}(\text{j_vec})]$ is to be replaced by parts of the definition of A, it has to be determined to which element of A the index vector $\text{I_op}(\text{j_vec})$ refers. Since these index vectors, in general, may be spread over the whole array A, the set of index vectors $\text{IV}_{2,i}$ has to be divided up into sets $\text{IV}_{2,i,1}$, ..., $\text{IV}_{2,i,m}$ with respective expressions $\text{Op}_{2,i,1}(\text{j_vec})$, ..., $\text{Op}_{2,i,m}(\text{j_vec})$ where each expression $\text{Op}_{2,i,j}(\text{j_vec})$ is derived from $\text{Op}_{2,i}(\text{j_vec})$ by replacing $A[\text{I_op}(\text{j_vec})]$ by $\text{Op}_{1,j}(\text{I_op}(\text{j_vec}))$ as specified in Fig.4.

Applying this transformation rule to our example problem we get a sequence of program transformations as shown in Fig.5. Starting out from the two given

```

{ ...
  A = internal_with( i_vec) {
    IV1,1 : Op1,1( i_vec)
    ⋮      ⋮
    IV1,m : Op1,m( i_vec)
  }
  ...
  B = internal_with( j_vec) {
    IV2,1 : Op2,1( j_vec)
    ⋮      ⋮
    IV2,i : Op2,i( j_vec)=¬...A[ I_op( j_vec)]...⊢
    ⋮      ⋮
    IV2,n : Op2,n( j_vec)
  }
  ...}

↓

{ ...
  A = internal_with( i_vec) {
    IV1,1 : Op1,1( i_vec)
    ⋮      ⋮
    IV1,m : Op1,m( i_vec)
  }
  ...
  B = internal_with( j_vec) {
    IV2,1 : Op2,1( j_vec)
    ⋮      ⋮
    IV2,i,1 : Op2,i,1( j_vec)=¬...Op1,1( I_op( j_vec))...⊢
    ⋮      ⋮
    IV2,i,m : Op2,i,m( j_vec)=¬...Op1,m( I_op( j_vec))...⊢
    ⋮      ⋮
    IV2,n : Op2,n( j_vec)
  }
  ...}

  with IV2,i,1 := { j_vec | j_vec ∈ IV2,i ∧ I_op( j_vec) ∈ IV1,1 }
    ⋮      ⋮
  IV2,i,m := { j_vec | j_vec ∈ IV2,i ∧ I_op( j_vec) ∈ IV1,m }

```

Fig. 4. Single WITH-loop-folding step.

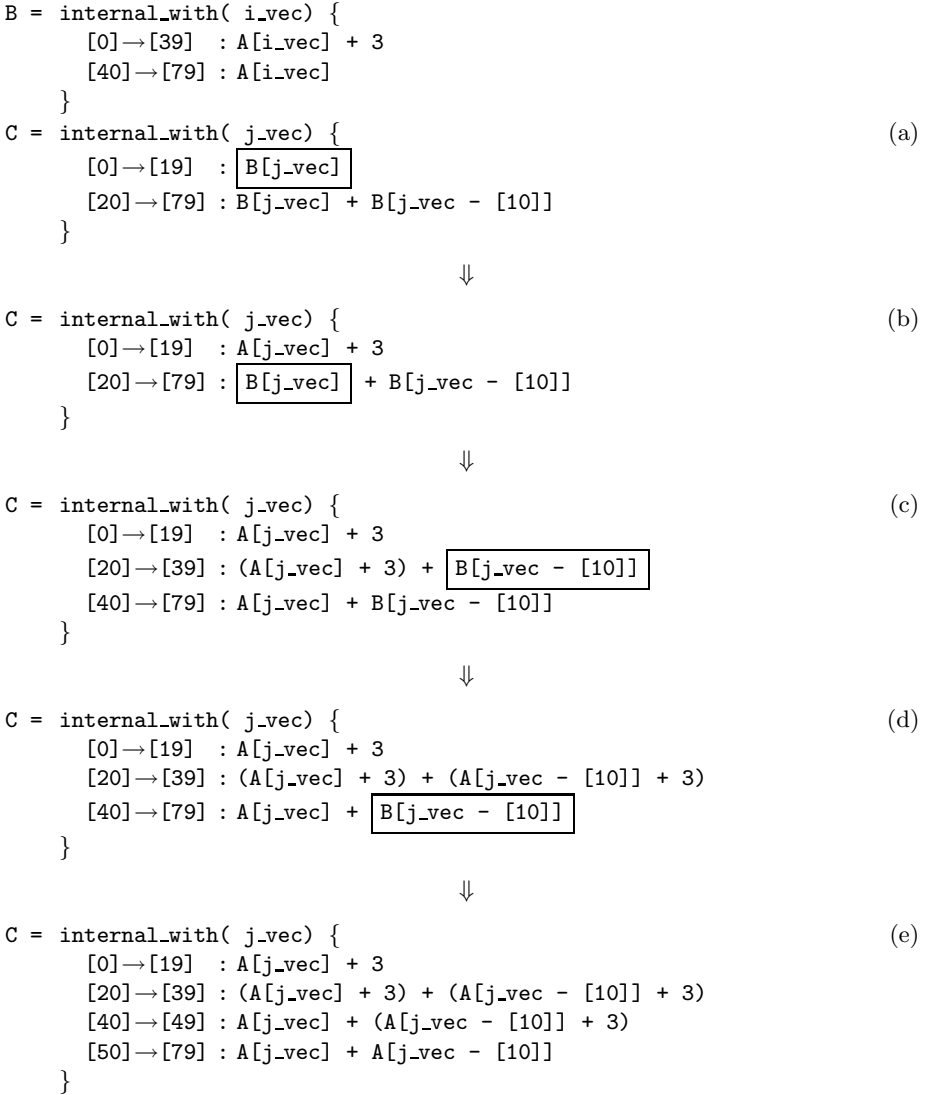


Fig. 5. Stepwise WITH-loop-folding at the example presented in Fig.3

WITH-loops in internal representation (Fig.5(a)), a stepwise transformation of the second WITH-loop construct is presented until the final version in Fig.5(e) is reached which does not contain any references to the array **B** anymore. Each of these steps results from a single application of the WITH-loop-folding rule from Fig.4; the references to **B** which are to be replaced in the next transformation step in each of the intermediate forms Fig.5(a) to Fig.5(d) are marked by boxes which surround them.

4 Simplifying Nestings of APL-Like Operations

As mentioned in the previous section, `WITH`-loop-folding is not solely ment for the optimization of user defined `WITH`-loops but as a universal tool for the optimization of nested primitive array operations as well. This requires all, or at least most, of the primitive array operations to be defined in terms of `WITH`-loops.

At this point, one of the major design principles for `WITH`-loops in SAC, namely to support the specification of shape-invariant array operations, pays off. It turns out that all primitive array operations of SAC can be defined as `WITH`-loops within a standard library rather than being implemented as part of the compiler. As an example, the definitions for `take`, and `rotate` are given in Fig.6.

```
inline double[] take( int[] new_shp, double[] A)
{
    B = with ( 0*new_shp <= i_vec <= new_shp-1)
        genarray( new_shp, A[i_vec]);
    return(B);
}

inline double[] rotate( int dim, int num, double[] A)
{
    max_rotate = shape(A)[[dim]];
    num = num % max_rotate;
    if( num < 0)
        num = num + max_rotate;
    offset = modarray( 0*shape(A), [dim], num);
    slice_shp = modarray( shape(A), [dim], num);
    B = with ( offset <= i_vec <= shape(A)-1)
        modarray( A, i_vec, A[i_vec-offset]);
    B = with ( 0*slice_shp <= i_vec <= slice_shp-1)
        modarray( B, i_vec, A[shape(A)-slice_shp+i_vec]);
    return(B);
}
```

Fig. 6. Library definitions of `take` and `rotate`.

`take` expects two arguments, a vector `new_shape` and an array `A`, where `new_shape` indicates the number of elements to be selected from the "upper left corner" of `A`. While this operation can easily be specified as a single `WITH`-loop that simply copies the requested elements from `A`, the definition of `rotate` is a bit more complex.

`rotate` expects three arguments: two integer numbers `dim` and `num`, and an array `A`, whose elements are to be rotated `num` positions along the axis `dim`. In order to avoid modulo operations for the computation of each element of the resulting array `B`, two different offsets to the index vectors `i_vec` are computed: one for those elements which have to be copied from positions with lower index vectors to positions with higher index vectors (`offset`) and another one

(`shape(A)-slice_shp`) for those elements that have to be copied from positions with higher index vectors to positions with lower index vectors. The usage of two different offsets on two disjoint ranges of index vectors leads to the specification of two consecutive WITH-loops.

With these definitions at hands, even rather complex nestings of primitive array operations can be transformed into sequences of WITH-loops which subsequently can be folded into single loop constructs. To illustrate this, the transformation of a simplified version of two-dimensional Jacobi relaxation [7] is outlined in the remainder of this section. Given a two-dimensional array `A` with shape `[m,n]` all inner elements, i.e., all those elements with non-maximal and non-minimal index components, have to be replaced by the sums of their respective neighbor elements. Although this algorithm could be specified elegantly by a single WITH-loop, we want to examine an artificially complex specification that uses the primitive array operations of SAC only in order to illustrate the strengths of WITH-loop-folding in the context of these operations:

```
double[] relax( double[] A)
{
    m = psi( [0], shape(A));
    n = psi( [1], shape(A));

    B = rotate( 0, 1, A) + rotate( 0, -1, A)
        + rotate( 1, 1, A) + rotate( 1, -1, A);

    upper_A = take( [1,n], A);
    lower_A = drop( [m-1,0], A);
    left_A  = drop( [1,0], take( [m-1,1], A));
    right_A = take( [m-2,1], drop( [1,n-1], A));
    inner_B = take( [m-2,n-2], drop( [1,1], B));

    middle = cat( 1, left_A, cat( 1, inner_B, right_A));
    result = cat( 0, upper_A, cat( 0, middle, lower_A));
    return(result);
}
```

The key idea of this piece of SAC-program is to specify the summation of neighbor elements as the summation of rotated arrays. Since in the resulting array `B` the boundary elements are modified as well, these elements have to be replaced by those of the initial array `A`. This is done by first cutting off the first row of `A` (`upper_A`), the last row of `A` (`lower_A`), the first and last column of the inner rows of `A` (`left_A` and `right_A` respectively), as well as the inner elements of `B`. Subsequently, these vectors/arrays are re-combined by successive catenation operations.

Due to the complexity of this example, we want to focus on the transformation of the nesting of array operations that specifies the computation of the array `B` first. Before an inlining of the function applications of `rotate` takes place, the nesting of array additions is transformed into a sequence of assignments whose right hand sides only consist of one array operation each, i.e.,

```
{ ...
  B = rotate( 0, 1, A) + rotate( 0, -1, A)
      + rotate( 1, 1, A) + rotate( 1, -1, A);
  ... }
```

is transformed into

```
{ ...
  tmp0 = rotate( 0, 1, A);
  tmp1 = rotate( 0, -1, A);
  tmp2 = tmp0 + tmp1;
  tmp3 = rotate( 1, 1, A);
  tmp4 = tmp2 + tmp3;
  tmp5 = rotate( 1, -1, A);
  B     = tmp4 + tmp5;
  ... }
```

After the complete program is transformed in a similar way, standard optimizations e.g. function inlining, constant folding, constant propagation or variable propagation (for surveys see [2, 4, 19, 27, 28]) are applied, which introduce several WITH-loops into our example. For the inlined version of the first application of the function `rotate` we obtain:

```
{ ...
  tmp0_B = with ( [1,0] <= i_vec <= shape(A)-1)
      modarray( A, i_vec, A[i_vec-[1,0]]);
  tmp0 = with ( [0,0] <= i_vec <= [0,n-1])
      modarray( tmp0_B, i_vec, A[[m-1,0]+i_vec]);
  ... }
```

Transforming the WITH-loops into internal representations yields:

```
{ ...
  tmp0_B = internal_with( i_vec) {
      [0,0]→[0,n-1] : A[i_vec]
      [1,0]→[m-1,n-1] : A[i_vec-[1,0]]
  }
  tmp0 = internal_with( i_vec) {
      [0,0]→[0,n-1] : A[[m-1,0]+i_vec]
      [1,0]→[m-1,n-1] : tmp0_B[i_vec]
  }
  ... }
```

To these two WITH-loops the folding mechanism from Section 3 can be applied, which leads to the elimination of array `tmp0_B`:

```
{ ...
  tmp0 = internal_with( i_vec) {
      [0,0]→[0,n-1] : A[[m-1,0]+i_vec]
      [1,0]→[m-1,n-1] : A[i_vec-[1,0]]
  }
  ... }
```

Likewise, we obtain for the next application of **rotate** as well as for the application of **+** to the two intermediates **tmp0** and **tmp1**:

```
{ ...
  tmp0 = internal_with( i_vec) {
    [0,0]→[0,n-1] : A[[m-1,0]+i_vec]
    [1,0]→[m-1,n-1] : A[i_vec-[1,0]]
  }
  tmp1 = internal_with( i_vec) {
    [0,0]→[m-2,n-1] : A[[1,0]+i_vec]
    [m-1,0]→[m-1,n-1] : A[i_vec-[m-1,0]]
  }
  tmp2 = internal_with( i_vec) {
    [0,0]→[m-1,n-1] : tmp0[i_vec]+tmp1[i_vec]
  }
  ... }
```

This sequence of WITH-loops again allows for the application of WITH-loop-folding. As a consequence, the temporary arrays **tmp0** and **tmp1** are of no further use and the specification of the loop construct which defines **tmp2** is split up into three disjoint ranges of index vectors: the left column, all inner columns, and the right column.

```
{ ...
  tmp2 = internal_with( i_vec) {
    [0,0]→[0,n-1] : A[[m-1,0]+i_vec] + A[[1,0]+i_vec]
    [1,0]→[m-2,n-1] : A[i_vec-[1,0]] + A[[1,0]+i_vec]
    [m-1,0]→[m-1,n-1] : A[i_vec-[1,0]] + A[i_vec-[m-1,0]]
  }
  ... }
```

Applying these optimizations further, we obtain for the generation of **B** a single loop construct with nine disjoint sets of array elements to be computed differently.

Similarly, the other temporary arrays specified explicitly in the function **relax** can be folded into single WITH-loops, e.g.

```
{ ...
  inner_B = take( [m-2,n-2], drop( [1,1], B));
  ... }
```

is transformed into

```
{ ...
  inner_B = internal_with( i_vec) {
    [0,0]→[m-3,n-3] : B[i_vec+[1,1]]
  }
  ... }
```

which finally leads to a single WITH-loop that implements the entire relaxation step:

```

double[] relax( double[] A)
{
    m = psi( [0], shape(A));
    n = psi( [1], shape(A));

    result = internal_with( i_vec) {
        [0,0]→[0,n-1]      : A[i_vec]
        [1,0]→[m-2,0]      : A[i_vec]
        [1,1]→[m-2,n-2]    : A[i_vec-[1,0]] + A[[1,0]+i_vec]
                           + A[i_vec-[0,1]] + A[[0,1]+i_vec]
        [1,n-1]→[m-2,n-1] : A[i_vec]
        [m-1,0]→[m-1,n-1] : A[i_vec]
    }

    return(result);
}

```

5 Implementing WITH-Loop-Folding

After a formal description of WITH-loop-folding in Section 3 and a case study of its applicability in the context of APL-like operations in the last section a few implementation issues have to be discussed. Since this is work in progress only two questions will be addressed here:

- Which impact does WITH-loop-folding have on the runtime performance of compiled SAC programs?
- How can WITH-loop-folding be implemented as a (statical) compiler optimization?

The general problem concerning runtimes is that we have performance gains due to the elimination of "temporary arrays" on one side, and potential performance losses due to the loss of sharing of the computations for individual elements of these arrays on the other side. Therefore, an exact prediction of the runtime impact of WITH-loop-folding would require a cost analysis for all operations involved, which in case of element computations that depend on the evaluation of conditionals is not statically decidable.

To guarantee speedups, as a first conservative approach, we restrict WITH-loop-folding to situations where it can be guaranteed that no sharing is lost at all. A simple criterion which is sufficient to exclude a loss of sharing is to demand that the "temporary arrays" may only be referenced once within the body of the second WITH-loop, and that the index vector projection used for the selection of elements of the "temporary array" (`I_op` from Fig.4) is surjective. Despite being quite restrictive these two conditions are met for the relaxation example from the last section. However, practical experiences from the application of WITH-loop-folding to real world examples will have to show whether this restriction is acceptable, or more elaborate criteria have to be developed.

The implementation of WITH-loop-folding as part of the compilation process imposes other restrictions on its applicability. The central problem in this context is that the index vector sets involved have to be known statically. Although this for the general case is not possible, in most situations it nevertheless can be done. The reason for this situation is that most generator parts of WITH-loops are specified relative to the shape of their argument arrays. Since the type inference system of SAC infers the exact shapes of all argument arrays, most of these index vector sets can be computed statically by simply applying constant folding. But inferring the index vector sets of the initial WITH-loops does not suffice to do WITH-loop-folding statically. Another important prerequisite is to be able to compute intersections of such index vector sets as well as their projection by index vector mapping functions (**I_op** from Fig.4). Therefore, the projection functions have to be restricted to a set of functions which are "simple enough" to render these computations possible.

As a conservative approach we restrict these projection functions to linear projections, i.e., we allow element-wise multiplications and element-wise additions with n-ary vectors. For this restriction we can easily define a simple notation for index vector sets which is closed under intersection building and linear projection:

Let $l = [l_1, \dots, l_n]$, $u = [u_1, \dots, u_n]$, and $s = [s_1, \dots, s_n]$ denote vectors of length $n \in \mathbb{N}$. Then $l \xrightarrow{s} u$ defines the set of Index vectors

$$\{[i_1, \dots, i_n] \mid \forall j \in \{1, \dots, n\} : (l_j \leq i_j \leq u_j \wedge \exists k_j \in \mathbb{N}_0 : i_j = l_j + k_j s_j)\} \quad .$$

With this notation of index vector sets, intersections can be computed as follows: Let $A = l_A \xrightarrow{s_A} u_A$ and $B = l_B \xrightarrow{s_B} u_B$ denote two index vector sets of index vectors of length $n \in \mathbb{N}$. Then we have

$$A \cap B = \begin{cases} l \xrightarrow{lcd(s_A, s_B)} \min(u_A, u_B) & \text{iff } \exists x, y \in \mathbb{N}_0^n : \wedge (max(l_A, l_B) \leq l) \\ & \wedge (l < max(l_A, l_B) + lcd(s_A, s_B)) \\ \emptyset & \text{otherwise} \end{cases}$$

where $lcd(a, b)$ denotes the element-wise least common denominator of the n-ary vectors a and b , and max , and min denote the element-wise maxima and minima respectively.

Furthermore, linear projections of such index vector sets can directly be expressed by other index vector sets of this kind, i.e., with $m = [m_1, \dots, m_n]$ denoting a vector of n integers we have

$$m * (l \xrightarrow{s} u) = (m * l) \xrightarrow{(m * s)} (m * u)$$

and

$$(l \xrightarrow{s} u) + m = (l + m) \xrightarrow{s} (u + m)$$

where $*$ and $+$ are understood as element-wise extensions of multiplication and summation for sets and vectors.

Similar to the restrictions imposed in order to guarantee runtime improvements empiric tests on real world examples will have to show whether these restrictions are appropriate or more complex projection functions have to be included.

6 Related Work

Although WITH-loop-folding is tailor-made for the WITH-loops provided by SAC and makes use of the particular properties of that language construct, some relations to other compiler optimizations can be observed.

From the functional point of view, it can be considered a special case of *deforestation* [25, 9, 10]. Both optimizations, WITH-loop-folding and the deforestation approach, aim at the elimination of intermediate data structures that are used only once. The basic idea of deforestation is to identify nestings of functions that recursively consume data structures with other functions that recursively produce these structures. Once such a nesting can be identified, the two recursions can be fused into a single one that directly computes the result of the consuming function.

Considering the basic case of WITH-loop-folding, where both WITH-loops apply a uniform operation to all array elements the first WITH-loop corresponds to a producing function whereas the second WITH-loop corresponds to a consuming function. For this case the only difference between the two optimizations is that WITH-loops operate on a single data structure whose size is statically known whereas deforestation deals with a recursive nesting of data structures of unknown depth.

Turning to the general case of WITH-loop-folding, the situation becomes more difficult since it allows the WITH-loops to apply different operations on disjoint sets of index vectors. In fact, such WITH-loops represent several different functions each of which computes a part of the elements of the resulting array, namely those that are characterized by a single set of index vectors. In order to be able to apply deforestation, a one-to-one correspondence between the producing and consuming functions is needed, i.e., the index vector sets have to be split up accordingly, which is exactly what WITH-loop-folding does. For the deforestation approach these particular situations cannot be detected as easily since that approach applies to the more general case where neither the complete data structure nor the producing and consuming functions are known explicitly as for WITH-loops in SAC.

Besides the relationship to deforestation, WITH-loop-folding can also be seen as a special combination of *loop fusion*, *loop splitting* and *forward substitution*, all of which are well-known optimization techniques in the community of high-performance computing (for surveys see [4, 19, 27, 28]). Although it is possible to draw the connection between these optimizations and WITH-loop-folding in the same manner as done for the deforestation approach, again the traditional optimization techniques suffer from their generality. In contrast, WITH-loop-folding can utilize several properties which for WITH-loops per definition hold, but for

other loop constructs are often undecidable: There are no dependencies between the computations for different elements of the same array, and there is no other effect of these loops but the creation of a single array; any temporary variable used within the "loop body" cannot be referenced anywhere else.

7 Conclusions

SAC is specifically designed for the compilation of high-level array operations into efficiently executable code. One of the major design principles of SAC is to support the specification of shape-invariant array operations. As a consequence, the programmer can stepwise abstract high-level array operations from primitive array operations by the definition of SAC functions. This includes the definition of functions, which are similar to the array operations available in APL or other languages that focus on the manipulation of arrays (e.g. NESL[5], or NIAL[15]). The basic language construct for the definition of such functions is the WITH-loop, a special loop construct for element-wise specifications of array manipulations.

This paper introduces a new compiler optimization, called WITH-loop-folding, for the transformation of two consecutive WITH-loops into a single one. As an extended example, a simplified version of Jacobi relaxation, specified by means of high-level array operations similar to those available in APL, is examined. It is shown that WITH-loop-folding in combination with standard optimizations, e.g. function inlining, constant folding, etc., allows the relaxation algorithm to be folded into a single loop construct that directly implements one relaxation step. A runtime comparison shows that this loop compiles to code which executes about 5 times faster than code which is compiled directly from the initial nesting of APL-constructs.

From a theoretical point of view, any nesting of primitive array operations can be folded into a single loop which defines the nested array operation as an element-wise direct computation of the resulting array from the argument array(s). As a consequence, the code compiled from an array operation becomes invariant against the number and kind of temporary arrays specified, i.e., the programmer is liberated from low level concerns.

However, with respect to the implementation of WITH-loop-folding, a couple of questions have not been addressed yet: Since the SAC compiler heavily specializes the source programs in order to be able to infer the exact shapes of all arrays statically, for real world examples, it may be necessary to apply WITH-loop-folding very often. Due to the complexity of the optimization this may lead to large compiler runtimes. Therefore, a sophisticated scheme for the execution order of WITH-loop-folding and standard optimizations may be required.

Another problem not yet addressed is the influence of caches on the relationship between WITH-loop-folding and the runtime behavior of compiled SAC programs. A trivial example for such a situation is a multiple access to the elements of an array in column major order. Since the arrays are stored row major order, such accesses lead to bad cache behavior. This can be improved

significantly by the explicit creation of a transposed array. To avoid slowdowns due to such problems, rules have to be established that determine which folding operations are favorable and which are unfavorable. For a more general solution, the development of other transformation schemes may be needed which analyze the array access schemes and after the application of WITH-loop-folding (re-)introduce temporary arrays whenever appropriate.

References

- [1] W.B. Ackerman and J.B. Dennis. VAL-A Value-Oriented Algorithmic Language: Preliminary Reference Manual. TR 218, MIT, Cambridge, MA, 1979.
- [2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN 0-201-10194-7.
- [3] Arvind, K.P. Gostelow, and W. Plouffe. The ID-Report: An asynchronous Programming Language and Computing Machine. Technical Report 114, University of California at Irvine, 1978.
- [4] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.
- [5] G.E. Blelloch. *NESL: A Nested Data-Parallel Language (Version 3.0)*. Carnegie Mellon University, 1994.
- [6] A.P.W. Böhm, D.C. Cann, R.R. Oldehoeft, and J.T. Feo. SISAL Reference Manual Language Version 2.0. CS 91-118, Colorado State University, Fort Collins, Colorado, 1991.
- [7] D. Braess. *Finite Elemente*. Springer, 1996. ISBN 3-540-61905-4.
- [8] D.C. Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, 1992.
- [9] W.-N. Chin. Safe Fusion of Functional Expressions II: Further Improvements. *Journal of Functional Programming*, 4(4):515–550, 1994.
- [10] D. Gilly et al. *Unix in a Nutshell*. A Nutshell Handbook. O'Reilly & Associates, Inc., 1992. ISBN 1-56592-001-5.
- [11] C. Grelck and S.B. Scholz. Classes and Objects as Basis for I/O in SAC. In T. Johnsson, editor, *Proceedings of the Workshop on the Implementation of Functional Languages'95*, pages 30–44. Chalmers University, 1995.
- [12] J. Hammes, S. Sur, and W. Böhm. On the effectiveness of functional language features: NAS benchmark FT. *Journal of Functional Programming*, 7(1):103–123, 1997.
- [13] K. Hammond, L. Augustsson, B. Boutel, et al. *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language*. University of Glasgow, 1995. Version 1.3.
- [14] K.E. Iverson. *A Programming Language*. Wiley, New York, 1962.
- [15] M.A. Jenkins and W.H. Jenkins. *The Q'Nial Language and Reference Manuals*. Nial Systems Ltd., Ottawa, Canada, 1993.
- [16] B.W. Kernighan and D.M. Ritchie. *Programmieren in C*. PC professionell. Hanser, 1990. ISBN 3-446-15497-3.
- [17] J. Launchbury and S. Peyton Jones. Lazy Functional State Threads. In *Programming Languages Design and Implementation*. ACM Press, 1994.
- [18] J.R. McGraw, S.K. Skedzielewski, S.J. Allan, R.R. Oldehoeft, et al. SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2. M 146, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1985.

- [19] D.A. Padua and M.J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Comm. ACM*, 29(12):1184–1201, 1986.
- [20] M.J. Plasmeijer and M. van Eckelen. *Concurrent Clean 1.0 Language Report*. University of Nijmegen, 1995.
- [21] S.-B. Scholz. *Single Assignment C – Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen*. PhD thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1996.
- [22] S.-B. Scholz. On Programming Scientific Applications in SAC - A Functional Language Extended by a Subsystem for High-Level Array Operations. In Werner Kluge, editor, *Implementation of Functional Languages, 8th International Workshop, Bad Godesberg, Germany, September 1996, Selected Papers*, volume 1268 of *LNCS*, pages 85–104. Springer, 1997.
- [23] P.R. Serrarens. Implementing the Conjugate Gradient Algorithm in a Functional Language. In Werner Kluge, editor, *Implementation of Functional Languages, 8th International Workshop, Bad Godesberg, Germany, September 1996, Selected Papers*, volume 1268 of *LNCS*, pages 125–140. Springer, 1997.
- [24] S. Smetsers, E. Barendsen, M. van Eeklen, and R. Plasmeijer. Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. Technical report, University of Nijmegen, 1993.
- [25] P.L. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.
- [26] H. Wehnes. *FORTTRAN-77: Strukturierte Programmierung mit FORTRAN-77*. Carl Hanser Verlag, 1985.
- [27] M.J. Wolfe. *High-Performance Compilers for Parallel Computing*. Addison-Wesley, 1995. ISBN 0-8053-2730-4.
- [28] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, 1991.

Types for Register Allocation

Johan Agat

Department of Computing Science
Chalmers University of Technology
agat@cs.chalmers.se

Abstract. We have set out to explore a type based approach to register allocation for purely functional languages. To lay a theoretical ground for that work, we have first developed a functional typed intermediate language with explicit register usage that we present in this paper. The language, which is a lambda calculus with flavours of assembly language, is able to express the register assignments resulting from a global, graph coloring based register allocation. We prove that our system is consistent and correct with respect to a standard semantics.

1 Introduction

Register allocation is the process of deciding where, in which register or in the memory, to store each value computed by the program being compiled. In modern RISC architectures, the difference in access time between a register and a memory cell can be as much as 4 to 10 times (or much more if a cache miss occurs) [1]. The register utilisation thus has a significant impact on the execution speed of a program.

By nature, register allocation must be done late in the compilation process when the evaluation order of sub-terms has been determined. In all compilers we know about, register allocation is made on the assembly-code level where the structure and type system of the compiler's intermediate language normally is lost. Figure 1 shows the imagined result of register-allocating the expression $3 + 5$.

```
mov 3,%r8
mov 5,%r6
add %r8,%r6,%r2
```

Fig. 1. Assembly code for $3+5$.

In this paper we present a *call by value* lambda calculus with register usage annotations and a *type and effect system* [2, 3], capable of expressing the register behaviour of such lambda terms. Our language has strong flavours of assembler and is capable of

$$\vdash \left(\begin{array}{l} \text{let } x = 3_{R8} \text{ in} \\ \text{let } y = 5_{R6} \text{ in} \\ \text{ADD}_{R8,R6,R2} \ x \ y \end{array} \right) : \text{Int}_{R2} ! \{R2, R6, R8\}$$

Fig. 2. Typing judgement of the annotated lambda term for $3+5$.

expressing the register assignments resulting from a global, graph coloring based register allocation. Terms are annotated with registers and stack-slots to determine their semantics and type. While still being a lambda calculus, the language can express register transfers and other low-level operations. In our system, types

propagate information about where in the register file, or on the stack, a value is located or expected to be and for functions also how they behave when applied. Figure 2 shows the equivalent of figure 1 in our language. The result type of the expression is Int_{R2} , indicating an integer residing in register 2.

For the simple example $3 + 5$, the register utilisation in our language is the same as that in the assembly language. However, by lifting the register allocation phase from the untyped assembly language to our typed language, where the structure of the intermediate language of the compiler is kept, we expect improvements in register utilisation when functions are involved:

Interprocedural Register Allocation.

Since functional languages are very call intensive, it's important to keep the overhead for function calls as low as possible. Usually, some rigid calling convention is used. For instance, some registers are used to pass arguments, some are saved by the caller, some are saved by the callee and one register is dedicated for the return value. Calling every function with a rigid convention like this is far from optimal. Registers will most likely be unnecessarily saved both by the caller and the callee. To get low cost function calls, a very flexible calling convention is important.

The types assigned to functions in our system are a very natural way of propagating information about where the function expects its arguments, where the return value will be and also which registers that might be modified when the function is called. If the function f has the type:

$$(\text{Int}_{\text{R3}} \xrightarrow{\{\text{R6}, \text{R8}\}} \text{Int}_{\text{R6}})_{\text{R2}}$$

we know that it expects an integer argument in register 3, produces an integer result in register 6 and may alter the contents of registers 6 and 8 in doing so. The pointer to the closure for f can be found in register 2. We call the set of registers that may be altered by f for it's *kill set*.

With this information available, local variables don't have to be saved around call sites unless they occupy any of the registers in the kill set of the applied function.

We also allow a function to have its environment partially loaded into registers:

$$\begin{array}{ll} \text{let } x = 42_{\text{R5}} \text{ in} & x : \text{Int}_{\text{R5}} \\ \text{let } g = \lambda_{\text{R8}} y. \text{ADD}_{\text{R5}, \text{R4}, \text{R3}} \ x \ y \text{ in} & g : (\text{Int}_{\text{R4}} \xrightarrow{\{\text{R3}\}} \text{Int}_{\text{R3}})_{\text{R8}; \{\text{R5}\}} \\ \dots & \end{array}$$

In this example, the function g will reference x in register 5 when applied. This is reflected in the type of g as shown above.

Types carrying such register usage information seem to provide a good setting to produce custom made calling conventions for every function and do inter-procedural register allocation [4].

Efficient Calling of Unknown Functions.

Unknown functions, e.g. function-valued arguments, are a bit of a problem to interprocedural register allocators since there is no syntactic information about which function that will actually be called at run-time. Since higher order functions are very commonly used in functional programs, efficient calling of argument functions becomes important. When compiling calls to such functions, some fixed calling convention must be used unless a global call-graph analysis is made prior to register allocation.

We provide an alternative treatment of unknown functions. The type of a higher order function contains information about how it will call its argument functions and thus which calling convention they must support. Consider the type:

$$((\text{Int}_{\text{R3}} \xrightarrow{\{\text{R6}, \text{R8}\}} \text{Int}_{\text{R6}})_{\text{R2}} \xrightarrow{\{\text{R3}, \text{R6}, \text{R7}, \text{R8}\}} \text{Int}_{\text{R7}})_{\text{R1}}$$

A function of this type requires its argument function to accept its argument in register 3, produce its result in register 6 and at most modify registers 6 and 8 when applied. It's then up to the register allocator to make sure that all functions passed to this function meet those requirements. If necessary, a “wrapper” of lambdas and register manipulating constructs can be built around the argument function. An example of this is given in section 2.2.

For higher order functions which repeatedly call their argument functions, e.g. *map*, there is a substantial speedup to be gained by passing such argument functions with most their environments loaded into registers.

Intermodular Interprocedural Register Allocation.

Since types are suitable for propagating register usage information across module boundaries, we hope our system will enable good inter-procedural register allocation to be done while still maintaining separate compilation.

Reasoning About Correctness.

With our formal system as a base, the register allocator can validate its own work by type checking the annotated term. Our system also provides a good foundation for reasoning about the correctness of a register allocation algorithm.

The language of annotated terms and the type system that we present in this paper are capable of expressing register usage. The type system ensures that well typed terms do not overwrite registers containing live data. The theorem stating this and its proof are discussed in section 3. After presenting the semantics of annotated terms and the type system in section 2, we give examples of annotated terms to point at different aspects of our system.

To produce the register annotations, conventional techniques for register allocation, such as graph colouring, can be used. We discuss this in sections 6.2 and 6.3.

The contribution of this work is showing that register assignments for the lambda calculus can be expressed at the lambda calculus level and that a type and effect system can be used to determine the correctness of such an assignment.

Variables	x, y, z, f, g
Literal values	$v ::= \text{True} \text{False} 0 1 2 \dots$
Stack-slots	$s ::= \text{S0} \text{S1} \dots$
Registers	$r ::= \text{R0} \text{R1} \dots \text{Rmax}$
Locations	$l ::= r s$
Primitive operators	$o ::= \text{ADD}_{r_1, r_2, r_3} \text{MUL}_{r_1, r_2, r_3} \text{DEC}_{r_1, r_2} \text{CMP0}_{r_1, r_2} \dots$
Terms	$ \begin{aligned} e ::= & v_r \mid x \mid o \ x_1 \dots x_n \mid \lambda_r x. e \mid f @_r x \\ & \mid \text{if}_r x \text{ then } e_1 \text{ else } e_2 \mid \text{let } x = e_1 \text{ in } e_2 \\ & \mid \text{letrec}_r f \ x = e_1 \text{ in } e_2 \\ & \mid \text{move}_{l_1 \leftarrow l_2} x = y \text{ in } e \mid \text{save}_r e \end{aligned} $

Fig. 3. Syntax of terms.

2 A Language with Explicit Register Usage

Central to our system is the concept of *locations*. A location is either a register or a stack slot. There are a finite number of registers available, $\text{R0}, \text{R1}, \dots, \text{Rmax}$, and infinitely many stack slots, $\text{S0}, \text{S1}, \dots$. We will use r to range over registers, s to range over stack slots and l to range over locations.

The syntax of terms is given in figure 3. Terms in our language consist of the standard constructs (variables, application etc.) plus the two constructs **move** and **save** used for manipulation of registers and stack slots. One thing to note about the language is that it is flat in the sense that all values must be bound to variables before they are used. Application, for example, is only allowed on variables, i.e. both the function and argument must be variables. This restriction simplifies the typing rules by making the evaluation order visible in the syntax of a term.

2.1 The Stack-Register Semantics

In figure 4 we give a semantics of register utilisation for annotated terms. We call this semantics the stack-register semantics, as opposed to the standard semantics introduced in section 4. Semantic judgements are of the form $\mathcal{S} | \mathcal{R} \vdash e \Downarrow \mathcal{R}_1$, stating that in the combined environment of the stack \mathcal{S} and the register file \mathcal{R} , the expression e evaluates by rewriting \mathcal{R} to \mathcal{R}_1 . The value of the evaluated expression will thus be located somewhere in \mathcal{R}_1 . The stack maps stack slots to values and the register file maps registers to values. Using a combined environment is absolutely necessary since the register file can only store a finite number of values. When more values than the register file can hold are live at once, some values must be stored in the stack and loaded into registers when they are needed. Transferring values between the stack and the register file (and also within the register file) is accomplished by use of the **move** construct.

We use v to range over values, consisting of integers and booleans, like 1, 2, 3, *True*, *False*, normal function closures $\langle x, e, \mathcal{S} \rangle$ and closures for (potentially) recursive functions $\langle f, x, e, \mathcal{S} \rangle$. Note that closures only contain the stack part of

$$\begin{array}{c}
\text{Lit} \frac{}{\mathcal{S}|\mathcal{R} \vdash v_r \Downarrow \mathcal{R}, r := v} \quad \text{Var} \frac{}{\mathcal{S}|\mathcal{R} \vdash x \Downarrow \mathcal{R}} \quad \text{Op} \frac{R_1 = \llbracket \mathbf{o} \rrbracket_R R}{\mathcal{S}|\mathcal{R} \vdash \mathbf{o} \ x_1 \dots x_n \Downarrow \mathcal{R}_1} \\
\\
\text{Lam} \frac{}{\mathcal{S}|\mathcal{R} \vdash \lambda_r x. e \Downarrow \mathcal{R}, r := \langle x, e, \mathcal{S} \rangle} \quad \text{App} \frac{\mathcal{R}(r) = \langle y, e, \mathcal{S}_1 \rangle \quad \mathcal{S}_1|\mathcal{R} \vdash e \Downarrow \mathcal{R}_1}{\mathcal{S}|\mathcal{R} \vdash f @_r x \Downarrow \mathcal{R}_1} \\
\\
\text{If}_T \frac{\mathcal{R}(r) = \text{True} \quad \mathcal{S}|\mathcal{R} \vdash e_1 \Downarrow \mathcal{R}_1}{\mathcal{S}|\mathcal{R} \vdash \text{if}_r x \text{ then } e_1 \text{ else } e_2 \Downarrow \mathcal{R}_1} \quad \text{AppR} \frac{\mathcal{R}(r) = \langle g, y, e, \mathcal{S}_1 \rangle \quad \mathcal{S}_1|\mathcal{R} \vdash e \Downarrow \mathcal{R}_1}{\mathcal{S}|\mathcal{R} \vdash f @_r x \Downarrow \mathcal{R}_1} \\
\\
\text{If}_F \frac{\mathcal{R}(r) = \text{False} \quad \mathcal{S}|\mathcal{R} \vdash e_2 \Downarrow \mathcal{R}_1}{\mathcal{S}|\mathcal{R} \vdash \text{if}_r x \text{ then } e_1 \text{ else } e_2 \Downarrow \mathcal{R}_1} \quad \text{Let} \frac{\mathcal{S}|\mathcal{R} \vdash e_1 \Downarrow \mathcal{R}_1 \quad \mathcal{S}|\mathcal{R}_1 \vdash e_2 \Downarrow \mathcal{R}_2}{\mathcal{S}|\mathcal{R} \vdash \text{let } x = e_1 \text{ in } e_2 \Downarrow \mathcal{R}_2} \\
\\
\text{Letrec} \frac{\mathcal{S}|\mathcal{R}, r := \langle f, x, e_1, \mathcal{S} \rangle \vdash e_2 \Downarrow \mathcal{R}_1}{\mathcal{S}|\mathcal{R} \vdash \text{letrec}_r f x = e_1 \text{ in } e_2 \Downarrow \mathcal{R}_1} \quad \text{Save} \frac{\mathcal{S}|\mathcal{R} \vdash e \Downarrow \mathcal{R}_1}{\mathcal{S}|\mathcal{R} \vdash \text{save}_r e \Downarrow \mathcal{R}_1, r := \mathcal{R}(r)} \\
\\
\text{Mv}_{S \leftarrow} \frac{\mathcal{S}, s := v|\mathcal{R} \vdash e \Downarrow \mathcal{R}_1 \quad v = \mathcal{R}(r)}{\mathcal{S}|\mathcal{R} \vdash \text{move}_{s \leftarrow r} x = y \text{ in } e \Downarrow \mathcal{R}_1} \quad \text{Mv}_{\mathcal{R} \leftarrow} \frac{\mathcal{S}|\mathcal{R}, r := v \vdash e \Downarrow \mathcal{R}_1 \quad v = \mathcal{S}(\mathcal{R}(l))}{\mathcal{S}|\mathcal{R} \vdash \text{move}_{r \leftarrow l} x = y \text{ in } e \Downarrow \mathcal{R}_1}
\end{array}$$

Fig. 4. The stack-register semantics.

the environment. A function value can thus have parts of its environment loaded in registers and thereby depend on these registers to be kept intact. To say that a closure is a value is hence somewhat misleading since the actual function value can occupy several registers, one containing a pointer to the closure and others containing parts of its environment.

We write $\mathcal{R}, r := v$ for the register file that maps r to v , and otherwise behaves like \mathcal{R} . A corresponding notation is used for updates to the stack. We use the notation $\mathcal{S}|\mathcal{R}(l)$ to mean $\mathcal{R}(l)$ if l is a register and $\mathcal{S}(l)$ if l is a stack-slot. We write \emptyset for the stack or register file that is undefined for all arguments.

The semantics of a term is determined solely by its annotations. Although variables play no part in the stack-register semantics, they cannot be omitted from the terms, or arbitrarily altered, since the type system will use them to check the correctness of the annotations. One can think of the annotations as defining the actual semantics of a term and the variables as defining its intended semantics. The type system then link these together so that, for well typed terms, the actual semantics will be the same as the intended one. Value-producing terms are annotated with a register to hold the produced value and annotations on value-consuming terms determine where to look for the value to be consumed. Literals, lambda abstraction and **letrecs** are annotated with which register to load the literal value or closure into. The annotations on function application and if-expressions determine where to look for the closure or boolean. Upon function application, the closure is simply retrieved and entered, assuming that

the closure expression will access its argument in the correct register. It will be up to the type system to ensure that this is the case.

Variable and **let** terms are not annotated at all. Since there is no computation involved directly in these terms, variables are names of already computed values and **let** is just used to sequence subcomputations, there is no need for any annotations. The semantics of primitive operators are determined by the operator denotation function $\llbracket - \rrbracket_R$ that maps each defined operator to a register file transforming function. The primitive operators are intended to reflect the available arithmetic and test instructions in the target architecture. Each operator is parametrised with the registers of its operands and result. These registers must of course be fixed for each occurrence of an operator since decoding register information at runtime to determine which registers to add, for instance, would be far too costly to be allowed in any realistic implementation. One example of the denotation of an instance of the add operator would be $\llbracket \text{ADD}_{R4,R5,R7} \rrbracket_R = \lambda \mathcal{R}. \mathcal{R}, R7 := \mathcal{R}(R4) + \mathcal{R}(R5)$.

The two constructs **move** and **save** are intended for register manipulation purposes. Values can be spilled to the stack with the **move** construct and later reloaded with another **move**. To decrease the destructive effects an expression will have on the register file, a number of **saves** can be wrapped around it.

A short note on implementation: In the rule for **let** expressions, both subexpressions are evaluated in the same stack environment. An implementation does not have to save and restore the entire stack however, if all **moves** to the stack environment are implemented by allocating a new stack cell at the top and writing the value into to that one. Recall that the stack environment is just a mapping from stack slots (names of places) to values, there is nothing that forces stack slot **S2** to be located at offset 2 from the stack pointer for instance.

Function calls occurring in tail-call-position inside a **save** cannot be implemented as tail calls. This is because the flow of control must return to the **save** in order to restore the the saved register.

The closures built by the rules **Lam** and **Letrec** contain the whole current stack environment. An implementation will of course only copy the live stack values into the closure when building it. When a function closure is entered, it can copy the values in the closure to the top of the system stack and upon return throw these values away by resetting the stack pointer to the value it had when the closure was entered.

2.2 Types and Typing Rules

In a traditional type system every term is assigned a type describing what kind of value that term will compute. A type and effect system goes one step further and assigns also an effect to every term, which is a description of *how* the term computes. The effect assigned to a term is a conservative approximation of the observable side-effects that the term may have when it is evaluated. Our type and effect system ensures that terms are correctly annotated in the respect that no subcomputation will overwrite registers containing values possibly needed

further on in the computation. Typing judgements take the form $\Gamma \vdash e : \tau ! k$, stating that in assumption environment Γ the term e has type τ and *kill effect* k , where k is a conservative approximation of the registers modified upon evaluation of e . Kill effects are simply sets of registers. Since the stack environment is protected from side effects and only propagated syntactically downwards in the evaluating term (see the Let rule in figure 4), no altered stack slots need be kept track of in the kill effect. The assumption environment Γ is a mapping from variables to types. To keep the system simple, we have used a monomorphic type-language without algebraic data types. We are well aware that this type system is too inexpressive to be used in a realistic implementation but it serves our purpose good in showing that types can express register utilisation. In section 6.1 we discuss extensions of the system to include polymorphism and algebraic data types.

Types have syntax as follows:

$$\begin{array}{lll} \text{Types:} & \mathcal{T} ::= \mathcal{C}_{l,d} & , \text{ ranged over by } \tau \\ \text{Constructors:} & \mathcal{C} ::= \mathcal{A} | \mathcal{T} \xrightarrow{k} \mathcal{T} & , \text{ ranged over by } \sigma \\ \text{Base constructors:} & \mathcal{A} ::= \text{Int} | \text{Bool} & , \text{ ranged over by } A \\ \text{Sets of registers:} & d, k & \end{array}$$

A type is a constructor annotated with a location and a *displacement*, which is the set of registers a value of that type may depend on to be kept unaltered. No stack slots need to be recorded in the displacement because all stack-residing values was moved into the closure when it was built. Base types have an empty displacement but a function will have a type with a displacement containing all registers occupied by the register-file part of the functions environment. We omit empty displacements, writing only σ_l instead of $\sigma_{l,\emptyset}$. The function space constructor is annotated with the latent kill effect of the function, which is the set of registers that may be altered when the function is applied. In the Lit rule we treat the base constructors Int and Bool as sets of values, i.e. $\text{Bool} = \{\text{True}, \text{False}\}$ and $\text{Int} = \{\dots, -1, 0, 1, 2, \dots\}$.

The type rules are given in figure 5. Some auxiliary functions have been used in stating them. We write $\Gamma[\text{FV}(e)]$ for the restriction of Γ to the set of free variables, $\text{FV}(e)$, in the term e . To get the *register pressure* and *stack pressure* of a type, i.e. the set of registers or stack-slots occupied by a value of that type, we use the functions rP and sP , defined as follows:

$$\begin{array}{ll} rP(\sigma_{r,d}) = \{r\} \cup d & sP(\sigma_{r,d}) = \emptyset \\ rP(\sigma_{s,d}) = d & sP(\sigma_{s,d}) = \{s\} \\ rP(\Gamma) = \bigcup_{x \in \text{Dom}(\Gamma)} rP(\Gamma(x)) & sP(\Gamma) = \bigcup_{x \in \text{Dom}(\Gamma)} sP(\Gamma(x)) \end{array}$$

Many rules have premises, or rather side conditions, requiring that two sets are disjoint, or that some register is not a member of a particular set. These premises check the register assignments of the term. They disallow incorrectly annotated terms, i.e. terms that would overwrite registers containing values possibly needed

$$\begin{array}{c}
\text{Lit} \frac{n \in A}{\Gamma \vdash v_r : A_r ! \{r\}} \quad \text{Var} \frac{\Gamma(x) = \sigma_{r,d}}{\Gamma \vdash x : \sigma_{r,d} ! \emptyset} \quad \text{Op} \frac{\Gamma \vdash_{i=1}^n x_i : \tau_i ! \emptyset \quad \llbracket o \rrbracket : = (\tau_1, \dots, \tau_n) \xrightarrow{k} \tau}{\Gamma \vdash o x_1 \dots x_n : \tau ! k} \\
\\
\text{Lam} \frac{\Gamma, x : \tau_1 \vdash e : \tau ! k \quad r \notin d \quad d = rP(\Gamma[FV(e)])}{\Gamma \vdash \lambda_r x. e : (\tau_1 \xrightarrow{k} \tau)_{r,d} ! \{r\}} \quad \text{App} \frac{\Gamma \vdash x : \tau_1 ! \emptyset \quad \Gamma \vdash f : (\tau_1 \xrightarrow{k} \tau_2)_{r,d} ! \emptyset}{\Gamma \vdash f @_r x : \tau_2 ! k} \\
\\
\text{If} \frac{\Gamma \vdash e_1 : \tau ! k \quad \Gamma \vdash e_2 : \tau ! k \quad \Gamma(x) = \text{Bool}_r}{\Gamma \vdash \text{if}_r x \text{ then } e_1 \text{ else } e_2 : \tau ! k} \quad \text{Let} \frac{\Gamma \vdash e_1 : \tau_1 ! k_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 ! k_2 \quad k_1 \cap rP(\Gamma[FV(e_2)]) = \emptyset}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 ! k_1 \cup k_2} \\
\\
\text{Letrec} \frac{\Gamma, f : (\tau \xrightarrow{k_1} \tau_1)_{r,d}, x : \tau \vdash e_1 : \tau_1 ! k_1 \quad \Gamma, f : (\tau \xrightarrow{k_1} \tau_1)_{r,d} \vdash e_2 : \tau_2 ! k_2 \quad d = rP(\Gamma, f : (\tau \xrightarrow{k_1} \tau_1)_{r,d} [FV(e_1)]) \quad r \notin rP(\Gamma[FV(e_1) \cup FV(e_2)])}{\Gamma \vdash \text{letrec}_r f x = e_1 \text{ in } e_2 : \tau_2 ! k_2 \cup \{r\}} \\
\\
\text{Move} \frac{\Gamma \vdash y : \sigma_{l_2,d} ! \emptyset \quad \Gamma, x : \sigma_{l_1,d} \vdash e : \tau ! k \quad k_s \cap sP(\Gamma[FV(e)]) = \emptyset \quad k_r; k_s \text{ cmv } \sigma_{l_1,d}; \sigma_{l_2,d} \quad k_r \cap rP(\Gamma[FV(e)]) = \emptyset}{\Gamma \vdash \text{move}_{l_1 \leftarrow l_2} x = y \text{ in } e : \tau ! k \cup k_r} \\
\\
\text{Save} \frac{\Gamma \vdash e : \tau ! k \quad r \notin rP(\tau)}{\Gamma \vdash \text{save}_r e : \tau ! k \setminus \{r\}} \quad \text{Sub} \frac{\Gamma \vdash e : \tau ! k \quad \tau \preceq \tau_1 \quad k \subseteq k_1}{\Gamma \vdash e : \tau_1 ! k_1}
\end{array}$$

Fig. 5. Type rules.

later on in the computation. These premises correspond to the conflicts generated at each program point in a conventional three-address-code setting, where an assigned variable is said to be in conflict (or interfere) with all variables live at that point [5].

To type operator application, we call for the operator signature function $\llbracket - \rrbracket$: to be defined, mapping every primitive operator to a signature of the form $(\tau_1, \dots, \tau_n) \xrightarrow{k} \tau$. Since primitive operators can take multiple arguments and are part of the semantic machinery, i.e. they are not loaded into any register, we don't assign location-annotated types to them. The signature of an operator must of course match its semantics if the system is to be consistent. More on this in section 3.

In the Lam rule, the effects of the abstraction body are encapsulated in the type of the function. Loading of the closure for the abstraction is not allowed to overwrite any of the registers holding the free variables of its body. The register holding the result value of a function will (except for the identity function, which

$$\frac{}{\emptyset; \emptyset \text{ cmv } \sigma_{r,d}; \sigma_{r,d}} \quad \frac{r \notin d \quad r \neq l}{\{r\}; \emptyset \text{ cmv } \sigma_{r,d}; \sigma_{l,d}} \quad \frac{}{\emptyset; \{s\} \text{ cmv } \sigma_{s,d}; \sigma_{r,d}}$$

Fig. 6. Consistent-move relation.

$$\text{Refl}_{\preceq} \frac{}{\tau \preceq \tau} \quad \text{Fun}_{\preceq} \frac{k \subseteq k' \quad \tau'_1 \preceq \tau_1 \quad \tau_2 \preceq \tau'_2 \quad d \subseteq d'}{(\tau_1 \xrightarrow{k} \tau_2)_{l,d} \preceq (\tau'_1 \xrightarrow{k'} \tau'_2)_{l,d'}}$$

Fig. 7. Subtype relation.

has both an empty displacement and kill set) always occur in the functions kill effect and/or displacement. The only way the result register can avoid occurring in the kill effect is when a value computed outside the lambda abstraction is returned by it:

```

let x = 42R3 in      x : IntR3
let f = λR1y.x in    f : (IntR8 → IntR3)R1  (for example)
...

```

Latent effects are unleashed upon application. Type rule App makes sure that the closure will be sought after in the correct register and that the argument is loaded into the register where the function expects it.

The rule for typing **let**-expressions makes sure that evaluation of the first term won't damage values needed when evaluating the second term. The term

let $x = 1_{R1}$ **in** **let** $y = 2_{R1}$ **in** x

is not typeable since the disjointness condition of the **let**-expression binding y is not satisfied. The Let rule allows reuse of registers not containing live data:

$$\emptyset \vdash \text{let } x = 1_{R1} \text{ in let } y = 2_{R1} \text{ in } y : \text{Int}_{R1} ! \{R1\}$$

In this example, the disjointness condition of the innermost **let** is satisfied:

$$\{R1\} \cap rP(\{x : \text{Int}_{R1}\}[\text{FV}(y)]) = \emptyset$$

The Letrec rule is a combination of the rules Let and Lam. The type rule Move makes use of the *consistent move* relation, defined in figure 6, to reject stack-to-stack annotations that lack semantics and to get hold of the registers and stack-slots overwritten by the move. If $k_r; k_s \text{ cmv } \tau; \tau'$, then a **move** of a value of type τ' to a value of type τ has semantics and will overwrite registers k_r and stackslots k_s .

The kill effects are decreased with the **save** construct. Since the saved register will be written back to the register file after the body e of the **save** has been evaluated, the saved register is not allowed to be one of the registers containing the value computed by e .

A function type can be “casted” in a non-shallow way by wrapping the function inside lambdas, **moves** and **saves**. Suppose that f and g have types as follows:

$$\begin{aligned} f &: (\text{Int}_{\text{R5}} \xrightarrow{\{\text{R4}, \text{R7}\}} \text{Int}_{\text{R7}})_{\text{R8}} \\ g &: ((\text{Int}_{\text{R3}} \xrightarrow{\{\text{R4}\}} \text{Int}_{\text{R4}})_{\text{R5}} \xrightarrow{\{\text{R2}, \text{R3}, \text{R4}\}} \text{Int}_{\text{R2}})_{\text{R9}} \end{aligned}$$

then just applying g to f would be type incorrect. However, a wrapper, f' , for f could be constructed so that g could be applied to f' instead:

```

let  $f' = \text{move}_{\text{S0} \leftarrow \text{R8}} f0 = f$  in
   $\lambda_{\text{R5}} x. \text{save}_{\text{R7}}$ 
     $\text{save}_{\text{R5}}$ 
     $\text{move}_{\text{R5} \leftarrow \text{R3}} x1 = x$  in
     $\text{move}_{\text{R7} \leftarrow \text{S0}} f1 = f0$  in
    let  $y = f1 @_{\text{R7}} x1$  in
     $\text{move}_{\text{R4} \leftarrow \text{R7}} y1 = y$  in
     $y1$ 
in ...

 $f' : (\text{Int}_{\text{R3}} \xrightarrow{\{\text{R4}\}} \text{Int}_{\text{R4}})_{\text{R5}}$ 

```

Types are partially ordered by the sub-typing relation given in figure 7. If $\tau \preceq \tau'$ then a value v with type τ can safely be said to have type τ' . Since larger types have larger displacements and larger latent kill effects this will only result in unnecessary extra precautions when v is used. With the Sub rule, a term can be assigned a larger type and a larger kill set. Without subtypes, typing of application and if-expressions would not be possible for many intuitively correct terms. This subsumption rule differs from those in [6, 7] in that there is no *observation criterion* for the effects of a term to be decreased. The obvious reason for this is that our system lacks local effects. If a register is modified in a sub term, that will always be visible to the surrounding term.

3 Consistency

As the first step of showing that our system is correct we show that the type system is consistent w.r.t. the stack-register semantics. In other words the types and effects obtained in a type derivation match the semantic derivation. The consistency theorem we formulate is a variant of *subject reduction*, extended to deal also with effects and bears strong similarities with that in [7]. Informally the theorem states that “if e is well typed with type τ and effect k , then the result of evaluating e (if any) has type τ and actual effects not greater than k ”.

$$\begin{array}{c}
\text{Base} \frac{\mathcal{S}|\mathcal{R}(l) \in A}{\mathcal{S}|\mathcal{R} \models A_l} \qquad \text{Fun} \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 ! k \quad rP(\Gamma) \subseteq d \quad \mathcal{S}|\mathcal{R}(l) = \langle x, e, \mathcal{S}_1 \rangle \quad \mathcal{S}_1|\mathcal{R} \models \Gamma}{\mathcal{S}|\mathcal{R} \models (\tau_1 \xrightarrow{k} \tau_2)_{l,d}} \\
\\
\text{Rec} \frac{\begin{array}{l} \Gamma, f : (\tau'_1 \xrightarrow{k'} \tau'_2)_{r,d'}, x : \tau'_1 \vdash e : \tau'_2 ! k' \\ rP(\Gamma, f : (\tau'_1 \xrightarrow{k'} \tau'_2)_{r,d'}) \subseteq d' \subseteq d \quad \mathcal{S}_1|\mathcal{R} \models \Gamma \\ \Gamma, f : (\tau'_1 \xrightarrow{k'} \tau'_2)_{r,d'}, x : \tau_1 \vdash e : \tau_2 ! k \quad \mathcal{R}(r) = \langle f, x, e, \mathcal{S}_1 \rangle \\ \mathcal{S}|\mathcal{R}(l) = \langle f, x, e, \mathcal{S}_1 \rangle \end{array}}{\mathcal{S}|\mathcal{R} \models (\tau_1 \xrightarrow{k} \tau_2)_{l,d}}
\end{array}$$

Fig. 8. Consistent types.

To state this formally we must first discuss what we mean by “the result has type”.

In a conventional type system, types are given semantics as sets of values. This approach must be modified somewhat for our annotated types. It would not be sensible to ask which values that have type Int_{R4} without involving a register file since only integer values residing in register 4 can be said to have that type. In figure 8 we give semantics to types in the form of a relation between types and pairs of a stack and a register file. We write $\mathcal{S}|\mathcal{R} \models \tau$ and say that τ is consistent w.r.t. \mathcal{S} and \mathcal{R} . The notation $\mathcal{S}|\mathcal{R} \models \Gamma$ is used as a shorthand for $\mathcal{S}|\mathcal{R} \models \Gamma(x)$ for all x in the domain of Γ .

For the type system to be consistent with respect to the stack register semantics we must of course require that the signature given to an operator matches its semantic behaviour.

Definition 1

(Consistent operator signatures)

$\llbracket - \rrbracket$: is consistent w.r.t. $\llbracket - \rrbracket_R$ if

- (i) $\text{Dom}(\llbracket - \rrbracket) = \text{Dom}(\llbracket - \rrbracket_R)$
- (ii) If $\llbracket o \rrbracket = (\tau_1, \dots, \tau_n) \xrightarrow{k} \tau$ and $\emptyset|\mathcal{R} \models_{i=1}^n \tau_i$
then $\mathcal{R} \setminus k = \mathcal{R}_1 \setminus k$ and $\emptyset|\mathcal{R}_1 \models \tau$, where $\mathcal{R}_1 = \llbracket o \rrbracket_R R$

Thus the signature given to an operator must really reflect that operator’s register behaviour. More, we know that $\llbracket o \rrbracket_R R$ is defined when R supports all types τ_i in the signature of o .

We can now state the consistency theorem formally:

Theorem 1

(Consistency of the type system)

If $\llbracket - \rrbracket$: is consistent w.r.t. $\llbracket - \rrbracket_R$, $\Gamma \vdash e : \tau ! k$, $\mathcal{S}|\mathcal{R} \models \Gamma$ and $\mathcal{S}|\mathcal{R} \vdash e \Downarrow \mathcal{R}_1$
then $\emptyset|\mathcal{R}_1 \models \tau$ and $\mathcal{R} \setminus k = \mathcal{R}_1 \setminus k$.

To prove this theorem we need the two following simple lemmas:

Lemma 1

(Consistent subtypes)

If $\mathcal{S}|\mathcal{R} \models \tau$ and $\tau \preceq \tau'$ then $\mathcal{S}|\mathcal{R} \models \tau'$.

Lemma 2

(Sub can be pushed)

If D is a derivation of $\Gamma \vdash e : \tau ! k$, where e is not a variable-, literal-, operator- or lambda-term, there exists a derivation D' of $\Gamma \vdash e : \tau ! k$ that does not use Sub as its outermost rule.

The proof of Theorem 1 is fairly straightforward by induction on the length of evaluation. We sketch the case for application:

Case App We assume $\Gamma \vdash f @_r x : \tau_2 ! k$, $\mathcal{S}|\mathcal{R} \models \Gamma$ and $\mathcal{S}|\mathcal{R} \vdash f @_r x \Downarrow \mathcal{R}_1$.

From the first assumption, lemma 2 and type rule App we have:

$$\Gamma \vdash x : \tau_1 ! \emptyset \quad (1)$$

$$\Gamma \vdash f : (\tau_1 \xrightarrow{k} \tau_2)_{r,d} ! \emptyset \quad (2)$$

The third assumption and semantic rule App gives us:

$$\mathcal{R}(r) = \langle y, e, \mathcal{S}_1 \rangle \quad (3)$$

$$\mathcal{S}_1|\mathcal{R} \vdash e \Downarrow \mathcal{R}_1 \quad (4)$$

Since the only rules used to type (2) and (1) are Var and Sub we know from the second assumption, the definition of \models and lemma 1 that $\emptyset|\mathcal{R} \models (\tau_1 \xrightarrow{k} \tau_2)_{r,d}$ and $\emptyset|\mathcal{R} \models \tau_1$. From this, (3) and the Fun rule of the consistent types relation we now get:

$$\Gamma', y : \tau_1 \vdash e : \tau_2 ! k \quad (5)$$

$$\mathcal{S}_1|\mathcal{R} \models \Gamma', y : \tau_1 \quad (6)$$

Now we can use the induction hypothesis together with (5), (6) and (4) to get $\emptyset|R_1 \models \tau_2$ and $\mathcal{R} \setminus k = \mathcal{R}_1 \setminus k$.

4 Correctness

From theorem 1 we know that well typed terms that evaluate, produce values of the expected type. We don't know however, that the value computed is the correct one. To assure that the correct value is computed we compare the stack register semantics to a simple standard natural semantics. For well typed terms we want that they compute to the "same" value in both semantics. Our formulation and proof of this theorem follows the same pattern as that of theorem 1.

$$\begin{array}{c}
\text{Lit} \frac{}{\mathcal{E} \vdash v \Downarrow_S v} \quad \text{Var} \frac{\mathcal{E}(x) = v}{\mathcal{E} \vdash x \Downarrow_S v} \\
\\
\text{Op} \frac{\mathcal{E} \vdash_{i=1}^n x_i \Downarrow_S v_i \quad v = \llbracket o \rrbracket_V v_1 \dots v_n}{\mathcal{E} \vdash o x_1 \dots x_n \Downarrow_S v} \quad \text{Lam} \frac{}{\mathcal{E} \vdash \lambda x. \bar{e} \Downarrow_S \langle x, \bar{e}, \mathcal{E} \rangle} \\
\\
\text{App} \frac{\mathcal{E} \vdash x \Downarrow_S v_1 \quad \mathcal{E} \vdash f \Downarrow_S \langle y, \bar{e}, \mathcal{E}_1 \rangle \quad \mathcal{E}_1, y := v_1 \vdash \bar{e} \Downarrow_S v}{\mathcal{E} \vdash f @ x \Downarrow_S v} \quad \text{AppRec} \frac{\mathcal{E} \vdash f \Downarrow_S \langle g, y, \bar{e}, \mathcal{E}_1 \rangle \quad \mathcal{E} \vdash x \Downarrow_S v_1 \quad \mathcal{E}_1, f := \langle g, y, \bar{e}, \mathcal{E}_1 \rangle, y := v_1 \vdash \bar{e} \Downarrow_S v}{\mathcal{E} \vdash f @ x \Downarrow_S v} \\
\\
\text{If}_T \frac{\mathcal{E} \vdash x \Downarrow_S \text{True} \quad \mathcal{E} \vdash \bar{e}_1 \Downarrow_S v}{\mathcal{E} \vdash \text{if } x \text{ then } \bar{e}_1 \text{ else } \bar{e}_2 \Downarrow_S v} \quad \text{Let} \frac{\mathcal{E} \vdash \bar{e}_1 \Downarrow_S v_1 \quad \mathcal{E}, x := v_1 \vdash \bar{e}_2 \Downarrow_S v_2}{\mathcal{E} \vdash \text{let } x = \bar{e}_1 \text{ in } \bar{e}_2 \Downarrow_S v_2} \\
\\
\text{If}_F \frac{\mathcal{E} \vdash x \Downarrow_S \text{False} \quad \mathcal{E} \vdash \bar{e}_2 \Downarrow_S v}{\mathcal{E} \vdash \text{if } x \text{ then } \bar{e}_1 \text{ else } \bar{e}_2 \Downarrow_S v} \quad \text{Letrec} \frac{\mathcal{E}_1 = \mathcal{E}, f := \langle f, x, \bar{e}_1, \mathcal{E}_1 \rangle \quad \mathcal{E}_1 \vdash \bar{e}_2 \Downarrow_S v}{\mathcal{E} \vdash \text{letrec } f x = \bar{e}_1 \text{ in } \bar{e}_2 \Downarrow_S v} \\
\\
\text{Move} \frac{\mathcal{E} \vdash y \Downarrow_S v \quad \mathcal{E}, x := v \vdash \bar{e} \Downarrow_S v}{\mathcal{E} \vdash \text{move } x = y \text{ in } \bar{e} \Downarrow_S v} \quad \text{Save} \frac{\mathcal{E} \vdash \bar{e} \Downarrow_S v}{\mathcal{E} \vdash \text{save } \bar{e} \Downarrow_S v}
\end{array}$$

Fig. 9. The standard semantics.

We write \bar{e} for the term e with all annotations removed. Judgements in the standard semantics are of the form $\mathcal{E} \vdash \bar{e} \Downarrow_S v$, where \mathcal{E} maps variables to values. Values are ranged over by v . Semantics rules, given in figure 9, are defined in the standard way, with **moves** treated as **lets** and **saves** as identity. Values are similar to those in the stack register semantics but now closures contain an \mathcal{E} environment with bindings for all free variables in the closure expression.

Our notion of when values in the two semantics are equal is formalised by the relation \sim given in figure 10. A triple of a stack, a register file and a type is used to specify the stack register value. We write $\mathcal{S} | \mathcal{R}; \Gamma \sim \mathcal{E}$ meaning $\mathcal{S} | \mathcal{R}; \Gamma(x) \sim \mathcal{E}(x)$ for all x in the domain of Γ .

Just as in the stack register semantics we make use of an auxiliary function, $\llbracket - \rrbracket_V$, to give operators their denotation and just as in the formulation of theorem 1 we require that the semantics and signatures of operators are related.

Definition 2

(Consistent operator denotation)

$\llbracket - \rrbracket_V$ is consistent w.r.t. $\llbracket - \rrbracket_R$ and $\llbracket - \rrbracket$: if

- (i) $\text{Dom}(\llbracket - \rrbracket) = \text{Dom}(\llbracket - \rrbracket_R) = \text{Dom}(\llbracket - \rrbracket_V)$
- (ii) If $\llbracket o \rrbracket = (\tau_1, \dots, \tau_n) \xrightarrow{k} \tau$, $v = \llbracket o \rrbracket_V v_1 \dots v_n$ and $\emptyset | \mathcal{R}; \tau_i \stackrel{n}{\sim}_{i=1} v_i$ then $\emptyset | \llbracket o \rrbracket_R \mathcal{R}; \tau \sim v$.

$$\begin{array}{c}
\text{Base} \frac{\mathcal{S}|\mathcal{R}(l) = n \quad n \in A}{\mathcal{S}|\mathcal{R}; A_l \sim n} \qquad \text{Fun} \frac{\Gamma, x: \tau_1 \vdash e: \tau_2 ! k \quad rP(\Gamma) \subseteq d \quad \mathcal{S}|\mathcal{R}(l) = \langle x, e, \mathcal{S}_1 \rangle \quad S_1|R; \Gamma \sim \mathcal{E}}{\mathcal{S}|\mathcal{R}; (\tau_1 \xrightarrow{k} \tau_2)_{l,d} \sim \langle x, e, \mathcal{E} \rangle} \\
\\
\text{Rec} \frac{\begin{array}{l} \Gamma, f: (\tau'_1 \xrightarrow{k'} \tau'_2)_{r,d'}, x: \tau'_1 \vdash e: \tau'_2 ! k' \\ rP(\Gamma, f: (\tau'_1 \xrightarrow{k'} \tau'_2)_{r,d'}) \subseteq d' \subseteq d \quad S_1|R; \Gamma \sim \mathcal{E} \\ \Gamma, f: (\tau'_1 \xrightarrow{k'} \tau'_2)_{r,d'}, x: \tau_1 \vdash e: \tau_2 ! k \quad \mathcal{R}(r) = \langle f, x, e, \mathcal{S}_1 \rangle \\ \mathcal{S}|\mathcal{R}(l) = \langle f, x, e, \mathcal{S}_1 \rangle \end{array}}{\mathcal{S}|\mathcal{R}; (\tau_1 \xrightarrow{k} \tau_2)_{l,d} \sim \langle f, x, e, \mathcal{E} \rangle}
\end{array}$$

Fig. 10. Cross-semantic value equality.

The correctness theorem is now stated as follows:

Theorem 2

(Correctness w.r.t. the standard semantics)

If $\llbracket _ \rrbracket_v$ is consistent w.r.t. $\llbracket _ \rrbracket_R$ and $\llbracket _ \rrbracket_\cdot, \llbracket _ \rrbracket_\cdot$ is consistent w.r.t. $\llbracket _ \rrbracket_R$, $\Gamma \vdash e: \tau ! k$, $\mathcal{E} \vdash \bar{e} \Downarrow_{\mathcal{S}} v$ and $\mathcal{S}|\mathcal{R}; \Gamma \sim \mathcal{E}$ then $\mathcal{S}|\mathcal{R} \vdash e \Downarrow \mathcal{R}_1$ and $\emptyset|\mathcal{R}_1; \tau \sim v$.

The proof is by induction of the length of evaluation in the standard semantics and makes use of theorem 1.

5 Related Work

Typed intermediate languages have recently been used by several modern compilers [8, 9, 10]. These compilers maintain and use type information in various transformations and optimisations. None of them, however, keep type information until the register allocation phase. Recent work by Morriset et al. describe a typed assembly language (TAL) [11] and how to compile System F terms into it. TAL is a register based assembly language based on a generic RISC instruction set. The main intended purpose of TAL is to enable a practical system for safely executing untrusted code. Since TAL-programs are typed with a normal type system and not a type and effect system, function closures in TAL cannot have free variables residing in registers. Furthermore, all live variables must be saved to memory locations around call sites.

5.1 Type and Effect Systems

Early work on effect systems by Lucassen and Gifford [2] identifies read, write and allocation operations on memory regions as effects. They perform their analysis on a language with imperative memory operations. The effect information gathered could then be used to determine if two expressions can have any interfering memory operations. If not, they could be scheduled for parallel execution.

For side-effect free languages, type and effect systems can be used as the base for analyses if the semantics of the language is given from a low-level enough point of view. With the view of a language implementor, all languages can be given side-effect semantics. When the compiled code for an expression is executed, things go on that are not part of the result, e.g. memory is allocated in the heap and register contents are altered. In Tofte/Talpins region inference [6], a type and effect system is used as a base in the translation of normal lambda terms to terms with explicit allocation/deallocation of memory regions. By using effects to keep track of memory usage they can implement the lambda calculus without a heap.

Our system bears some similarities with the region inference. The registers we use can be interpreted as fixed-size regions containing only one value. Our kill effects correspond to **put**-effects and our displacement sets correspond to the latent **get**-effects of a function. One important difference is that Tofte/Talpin's region annotated terms must be typed with regions of unbounded size and/or an unbounded number of regions. As discussed, our system types terms with only a finite number of registers available. Register allocation for a region annotated term is made first after it has been compiled to the (untyped) Kit Abstract Machine [12].

5.2 Conventional Register Allocation

A requirement of any register allocation method is that no values should be allowed to be written into registers containing live data. In a conventional 3-address code setting, graph colouring is one of the more well known techniques of performing register assignments to temporary variables [5, 13]. In this setting, an interference graph is computed where nodes in the graph correspond to temporary variables and edges to interferences between temporary variables. Two temporary variables interfere if one is assigned at a program point where the other is live.

Our type and effect system can be seen as a formalisation of this for the lambda calculus. The disjointness conditions present in the type rules of the syntactic constructs that are implemented as a sequencing of computations and/or operations correspond to the interference between assigned and live temporary variables in the 3-address code setting.

To our knowledge, we are the first to develop a type system capable of checking the correctness of register assignments made to lambda calculus terms.

6 Discussion and Future Work

As presented here, our language has all essential constructs to perform simple computations. It has recursion and if-expressions for conditional computation. The language is complete in the sense that any simply typed lambda calculus

term can be translated into a well typed annotated term in our language, requiring only two registers in the target architecture. Such a translation is just a straight forward compilation schema that **move** every value to the stack as soon as it is computed and only **move** the value back to the register file when it should take part in an application or primitive operation.

To be practically usable as a back end language in an optimising compiler, the language must however be extended.

6.1 Stronger Language and Types

First of all, algebraic data types and case-expressions must be included in such a way that they can be manipulated efficiently. The type system must be able to express both data nodes that are in-lined in the register file as well as allocated on the heap (and then pointed to from some register). It would then be possible for functions to use several registers to cheaply return large values in a type safe manner. Also, tuples with all components stored in registers would allow efficient calling conventions for multiple argument functions.

Since we aim at inter procedural register allocation across module boundaries, our system must be extended to include polymorphic types. We have normal separate compilation in mind where each module is compiled only once and without knowledge of the context in which the functions it defines will be used. In this setting, a monomorphic type system like the current one makes it impossible to separately compile modules with reusable functions such as *map*.

Polymorphic types in our system could be introduced by generalising over effects, displacements, locations and type-constructors. The type system would then have to propagate constraints on register- and effect-variables in a way resembling the qualified types of Jones [14]. These constraints would be more or less the same as the premises in the current system and they would flow in and out of type schemes at generalisation and instantiation. Polymorphic effects and displacements are most likely very useful since they make the types of higher order functions much more precise. For example, consider the following definition of *apply*:

$$\lambda_{R2} f. \lambda_{R1} x. f @_{R3} x$$

The polymorphic type for this term could look something like:

$$\forall_{\alpha, \beta, k, d, l, l'} . ((\alpha_l \xrightarrow{k} \beta_{l'})_{R3, d} \xrightarrow{\{R1\}} (\alpha_l \xrightarrow{k} \beta_{l'})_{R1, \{R3\} \cup d})_{R2}, \text{ where } R1 \notin d$$

With the monomorphic types in the current system, all effects and types must be fixed. Not only is this much more restrictive in terms of how the function can be used, it can also lead to loss of accuracy in the types. Fixing the type of *apply* to, say:

$$\begin{aligned} & ((\text{Int}_{R4} \xrightarrow{\{R3, R4, R5, R6\}} \text{Int}_{R5})_{R3, \{R7, R8\}} \xrightarrow{R1} \\ & (\text{Int}_{R4} \xrightarrow{\{R3, R4, R5, R6\}} \text{Int}_{R5})_{R1, \{R3, R7, R8\}})_{R2} \end{aligned}$$

would mean that all partial applications of `apply` would get the same type, namely:

$$(\text{Int}_{R4} \xrightarrow{\{R3,R4,R5,R6\}} \text{Int}_{R5})_{R1,\{R3,R7,R8\}}$$

So if a function with a smaller kill effect than $\{R3, R4, R5, R6\}$, say $\{R5\}$, was passed as an argument to `apply`, the type of the partial application would grossly overestimate the actual effects. With the polymorphic system, such information losses would not occur. So by using polymorphic effects, we hope to produce efficient inter-procedural register allocation across module boundaries.

6.2 Inference

The inference of register and location annotations for a term is the actual process of register allocation. Type and effect inference can be done by collecting and solving constraints corresponding to the premises in the type rules [7, 3]. In our case however, the constraints generated from a term naively translated from ordinary lambda calculus can very well be unsolvable. Consider the lambda term `let f = λx.x + 1 in f 1 + f 2` and suppose it was translated to:

```
let f = λr0x. let y = 1r1 in
      ADDr2,r3,r4 x y
in let z1 = 1r5 in
   let z2 = f @r6 z1 in
   let z3 = 2r7 in
   let z4 = f @r8 z3 in
   ADDr9,r10,r11 z2 z4
```

Here we assume that primitive operators for addition are available just like instructions in a RISC architecture, i.e. one for every combination of operand registers. Which ones to use will then be determined by the inference. Among the constraints gathered for this example would be $r_4 = r_9$ and $r_4 = r_{10}$ from the two applications of f and $r_9 \notin \{r_1, r_4\}$ from the last `let`-expression. Thus there is no assignment of registers to the register variables that satisfy the constraints. Such situations require the register allocator to insert `move` and/or `save` constructs until the constraints can be solved. This approach is just like the build-color-spill cycle in a conventional register allocator based on graph colouring [4, 15]. An example of how this term could be rewritten and allocated is:

```
let f = λR1x. let y = 1R2 in
      add_R3_R2_R3 x y
in let z1 = 1R3 in
   let z2 = f @R1 z1 in
   moveR4←R3 z5 = z2 in
   let z3 = 2R3 in
   let z4 = f @R1 z3 in
   add_R4_R3_R1 z5 z4
```

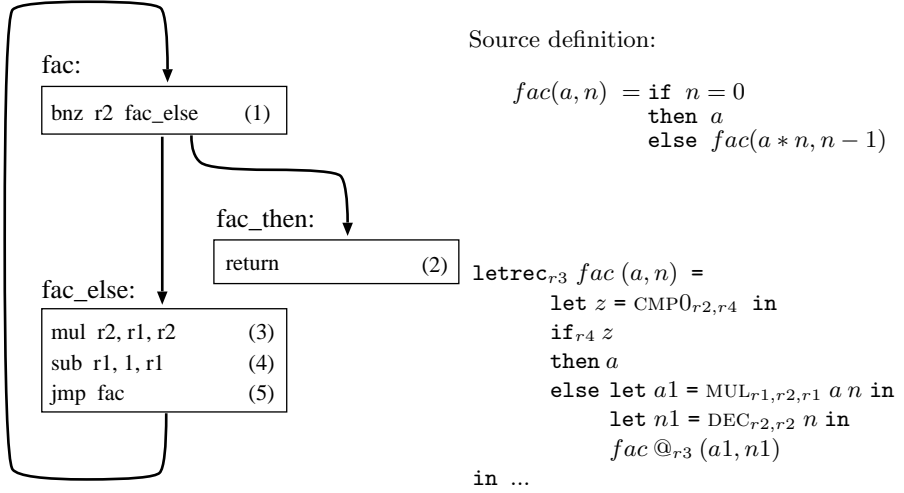


Fig. 11. The tail recursive factorial function compiled to 3-address code and annotated term.

If we recall for a moment how normal type inference is conducted, it gives us a clue about how to perform register allocation in our system. A type is inferred for a function only when types have been assigned to all functions that it references. Adapting this idea to register allocation in our setting seems to lead to a variant of the bottom-up register allocation discussed in [4]. But instead of allocating registers bottom-up in the call graph of the program, we would allocate registers bottom-up in the function name reference graph.

6.3 Expressiveness

Our language and type system is expressive enough to encode the register assignments resulting from a conventional graph colouring based register allocation. Figure 11 shows the tail recursive variant of the factorial function compiled to 3-address code and also translated to our register annotated language. We have here taken us the liberty of informally extending our language to handle multiple arguments. In the 3-address code, a is stored in virtual register $r1$ and n in $r2$. Performing live-variable analysis on this code, building an inference graph and colouring it would result in different registers for $r1$ and $r2$, since they conflict with each other from the operations performed on lines (3) and (4). For our annotated term, a type based constraint gathering followed by building and colouring an inference graph from the constraints, would also result in different register holding the values of a and n . The constraints enforcing this come from the disjointness condition of the let rule, applied at the bindings of $a1$ and $n1$.

The 3-address code in figure 11 shows two results of compilation that cannot be expressed by our current system. Firstly, the test $n = 0$ and the branching

is compiled to one **bnz** instruction. In our language, this must be expressed as a computation of a boolean and a branching on that boolean even though the implementation of the term would never compute any boolean. This mismatch with the implementation of our semantics can be dealt with by adding an **if0**-construct that selects its then branch if the test register contains 0.

Secondly, the recursive tail call has been compiled to a jump in the 3-address code but to a normal function call in the annotated term. Recursive calls in our language can still be implemented with jumps but since our system is unable to distinguish between normal and recursive function calls, a register is unnecessarily needed to store the recursive closure. The only way to work around this problem is by modifying the semantics and type system to keep track of recursive applications. We have avoided doing so in order to keep our system simple, thus trading some expressiveness for simplicity. Another slight problem with our formulation of recursive functions is that their types always have a non-empty displacement containing the register of the recursive closure. To avoid cluttering the register file with these displacement registers, entire recursive definitions can be wrapped inside lambdas:

$$\begin{aligned} \text{let } fac &= \lambda_{R3} n. \text{save}_{R3} \\ &\quad \text{letrec}_{R3} fac' \ n' = \dots \\ &\quad \text{in } fac' @_{R3} n \\ \text{in} \dots \end{aligned}$$

7 Conclusions

We have presented a variant of the lambda calculus and a type and effect system capable of expressing register utilisation. The system has been proved consistent and correct with respect to a standard semantics. Having argued that our system is expressive enough to encode the register assignments resulting from a conventional graph colouring base register allocation, we feel that the first step towards type based register allocation is taken. It is clear, however that the system must be extended with polymorphism and algebraic data-types before any serious attempts on type based register allocation can be made.

References

- [1] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers, Palo Alto, 1990.
- [2] John M. Lucassen and David K. Gifford. Polymorphic Effect Systems. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, San Diego, California, January 1988. ACM Press.
- [3] Pierre Jouvelot and David K. Gifford. Algebraic Reconstruction of Types and Effects. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, pages 303–310, Orlando, Florida, January 1991. ACM Press.
- [4] Peter A. Steenkiste. Advanced Register Allocation. In Peter Lee, editor, *Topics in Advanced Language Implementation*. MIT Press, 1991.

- [5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, Tools*. Addison-Wesley Publishing Company, Reading, Mass., 1986.
- [6] Mads Tofte and Jean-Pierre Talpin. Implementation of the Typed Call-by-Value λ -calculus using a Stack of Regions. In *21th Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, January 1994. ACM Press.
- [7] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 162–173, Santa Cruz, California, June 1992. IEEE Press.
- [8] Simon L. Peyton Jones. Compiling haskell by program transformation: a report from the trenches. In *Proceedings of the European Symposium on Programming*, Linköping, April 1996.
- [9] Zhong Shao and Andrew W. Appel. A Type-Based Compiler for Standard ML. In *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'95)*, pages 116–129, La Jolla, June 1995. ACM.
- [10] D. Tarditi, G. Morriset, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'96)*, pages 181–192, Philadelphia, May 1996. ACM.
- [11] Greg Morriset, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, San Diego, California, January 1998. ACM.
- [12] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From Region Inference to von Neumann Machines via Region Representation Inference. In *23rd ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 171–183, St. Petersburg Beach, Florida, January 1996. ACM.
- [13] F. Chow and J. Hennessy. Register Allocation by Priority-based Coloring. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 222–232, Montreal, 1984.
- [14] Mark P. Jones. A Theory of Qualified Types. In *ESOP'92: European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, Rennes, France, February 1992. Springer-Verlag.
- [15] Urban Boquist. Interprocedural Register Allocation for Lazy Functional Languages. Licentiate Thesis, Chalmers University of Technology, Mars 1995.
URL: <http://www.cs.chalmers.se/~boquist/lic.ps>.

Types for 0, 1 or Many Uses

Torben Æ. Mogensen

DIKU

Universitetsparken 1
DK-2100 Copenhagen O
Denmark

email: `torbenm@diku.dk`

phone: (+45) 35321404

fax: (+45) 35321401

Abstract. This paper will present an analysis for detecting single uses of values in functional programs during call-by-need reduction. There are several reasons why such information can be useful. The Clean language uses a uniqueness type system for detecting single-threaded uses which allow destructive updating. Single-use information has also been proposed for compile-time garbage collection. Turner, Wadler and Mossin have presented a single-use analysis which is intended to detect cases where call-by-need can safely be replaced by call-by-name.

This paper will focus on this last application of single-use analysis and present a type-based analysis which overcomes some limitations present in the abovementioned analysis.

Keywords: Program analysis, storage management, functional programming, types, constraints.

1 Introduction

This paper will present an analysis for detecting single uses of values in functional programs during call-by-need reduction. There are several reasons why such information can be useful. The Clean language [6] uses a uniqueness type system [1] for detecting single-threaded uses which allow destructive updating. In [3], single-use information is proposed for compile-time garbage collection using the observation that the any use of a single-use value will be the last use. In [10], Turner, Wadler and Mossin present an analysis based on linear types which is intended to detect cases where call-by-need can safely be replaced by call-by-name. The present paper will focus on this last application of single-use analysis and present a type-bases analysis which overcomes some limitations present in the analysis shown in [10].

The analysis is (like Turner *et al.*'s) based on a type system. We use the type rules to generate a set of constraints that are solved off-line using a linear-time constraint solver.

We do not attempt to prove our analysis correct, but it is expected that an approach similar to that used in [10] can be used.

1.1 Limitations in Turner *et al.*'s Analysis

Turner *et al.*'s analysis can handle simple cases of zero usage by use of structural rules similar to those found in linear logic. Since this method involves adding or deleting entire variables in the environment, it cannot handle zero usage of parts of variables, only whole variables.

Additionally, there is a restriction on usage-annotated types that means that whenever the spine of a list can be used repeatedly, it is assumed that the same is true for the elements. As an example, the function

$$\text{mean } l = \text{sum } l / \text{length } l$$

traverses the spine of l twice, but accesses the lements of l only once. The analysis cannot detect this, as it assumes each spine-traversal implies access to the elements.

In this paper, we introduce a type-based analysis that will be able to detect single use in such cases by removing the restriction on structured types and by introducing the ability to detect zero uses of parts of values by means of an explicit 0 usage annotation on types.

2 The Language

We use a simple functional language with integers, pairs and recursive datatypes and function definitions. This language is slightly larger than the language presented in [10], but this is mainly to show how general recursive data structures (*i.e.* more than lists) are handled.

2.1 Types

We assume that the programs are simply typed using the following types

$$t ::= \text{Int} \mid t_1 \times t_2 \mid \mu\alpha.c_1 t_1 + \dots + c_n t_n \mid \alpha \mid t_1 \rightarrow t_2$$

where α is a type variable used for recursive types. We will later discuss possible extension to polymorphic types.

2.2 Expressions

The abstract syntax of the language is given below. n is any natural number, the x_i are variables and the c_i are constructors.

$$\begin{aligned} e ::= & x \mid n \mid e_1 + e_2 \mid (e_1, e_2) \mid \text{split } e_1 \text{ to } (x_1, x_2) \text{ in } e_2 \\ & \mid c_i e \mid \text{case } e_0 \text{ of } c_1 x_1 \rightarrow e_1; \dots; c_n x_n \rightarrow e_n \\ & \mid e_1 e_2 \mid \lambda x : t. e \mid \text{let } x : t = e_1 \text{ in } e_2 \\ & \mid \text{letrec } x_1 : t_1 = e_1, \dots, x_n : t_n = e_n \text{ in } e_0 \end{aligned}$$

Note the *split* expression which breaks down a pair into its components. We use this instead of *fst* and *snd* because these would require two uses of a pair to get at the components, where *split* uses only one. Hence, uses of *split* instead of *fst* and *snd* will give more cases of single use.

Pattern matching is naturally translated into uses of *case* and *split* (more so than with *fst* and *snd*) and it is easy to define *fst* and *snd* in terms of *split*. Hence, it is no serious limitation to use *split*.

The *case* expressions are assumed to be exhaustive (though this doesn't really matter for the analysis).

3 Adding Usage Annotations

We will now annotate types and expressions with usage counts. The count indicates an upper bound on the usage of values of the annotated expression or type. Usage counts can be any of 0, 1 or ∞ . A count of 0 indicates that the value is certain never to be used. While this isn't likely to be the case for a value at the place of its construction, zero-counts are used to express that a variable (or part thereof) is unused in a subexpression. 1 means *at most* one use. ∞ indicates that no upper bound on the use has been detected.

3.1 Annotated Types

A count k on a type constructor in a well-typed program means that any value of that type is used at most k times during execution of the program.

When we use a type variable with a superscript count or count variable, e.g. t^1 or t^k we mean that t ranges over all types with that count on the topmost type constructor. Count annotated types are written using the following notation

$$t ::= Int^k \mid t_1 \times^k t_2 \mid \mu\alpha.^k c_1 t_1 + \dots + c_n t_n \mid \alpha \mid t_1 \rightarrow^k t_2$$

where k is a count (0, 1 or ∞).

3.2 Annotated Expressions

The aim of the analysis is to annotate value introducing expressions in the program by an usage count, that gives an upper limit to the number of uses of the value created by the expression. The syntax of annotated expressions is

$$\begin{aligned} e ::= & x \mid n^k \mid e_1 +^k e_2 \mid (e_1,^k e_2) \mid split\ e_1\ to\ (x_1, x_2)\ in\ e_2 \\ & \mid c_i^k\ e \mid case\ e_0\ of\ c_1\ x_1 \rightarrow e_1; \dots; c_n\ x_n \rightarrow e_n \\ & \mid e_1\ e_2 \mid \lambda^k x : t. e \mid let\ x : t = e_1\ in\ e_2 \\ & \mid letrec\ x_1 : t_1 = e_1, \dots, x_n : t_n = e_n\ in\ e_0 \end{aligned}$$

where k is a count (0, 1 or ∞). Note that the types used in abstractions and *let(rec)* expressions are now annotated types.

The annotations on value-producing expressions (*i.e.* constants, additions, pairs, injected values and lambda-abstractions) indicate upper bounds of the usage of the values produced by these expressions. Annotated types on lambda-bound and let(rec)-bound variables indicate how many times values bound to these variables are used *inside their scope*. Since the same value may be bound to several variables, a single-use annotation on one variable does not guarantee that the value bound to it is single-threaded. Only the annotation at the creation point of a value will reflect the total usage of that value throughout its lifetime.

For some kinds of optimization it is sufficient to know the number of uses of a value at the creation point of that value, but for other optimizations it is required at a usage point to know if this is the single use. The latter information is not present in annotations as shown, but it is a simple matter to propagate this information to the uses of the value. In this respect (as well as in our use of a 0 count and *split*-expressions), our analysis is similar to the analysis by Jensen and Mogensen from ESOP'90 [3].

3.3 Counts

Apart from the obvious ordering ($0 \leq 1 \leq \infty$) we also need some operations on counts:

$k_1 \sqcup k_2$				$k_1 + k_2$				$k_1 \cdot k_2$				$k_1 \triangleright k_2$			
$k_1 \setminus k_2$	0	1	∞	$k_1 \setminus k_2$	0	1	∞	$k_1 \setminus k_2$	0	1	∞	$k_1 \setminus k_2$	0	1	∞
0	0	1	∞	0	0	1	∞	0	0	0	0	0	0	0	0
1	1	1	∞	1	1	∞	∞	1	0	1	∞	1	0	1	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	0	∞	∞	∞	0	1	∞

\sqcup is least upper bound. $+$ and \cdot are addition and multiplication over counts, respectively. \triangleright is used as a guard to indicate that uses in an expression should not count if the result of the expression is never used. Hence, we use this operator to simulate laziness of evaluation: An expression is not evaluated if its result is never used.

We extend the ordering and the operations to work on annotated types and type environments as well as counts. The ordering extends pointwise: For a type t_1 to be less than another t_2 , the must have the same underlying type and the counts in t_1 must be less than the counts on the corresponding positions in t_2 . Note that this makes the ordering covariant on function types. Ordering on type environments E_1 and E_2 implies the same underlying structure (*i.e.* the same variable is bound to types in E_1 and E_2 that are identical except for annotations) and $E_1 \leq E_2$ indicates that for all x , if E_1 binds x to t_1 and E_2 binds x to t_2 then $t_1 \leq t_2$.

\sqcup (least upper bound) operator likewise works pointwisely on counts in equivalent positions in types and environments.

We will always use \cdot with a single count on the left-hand side, but may use either of count, type and environment on the right-hand side. If a type or environment occurs on the right-hand side, all count annotation in the type or

environment are multiplied by the left-hand count, e.g. $\infty \cdot Int^1 \rightarrow^1 Int^0 \times^\infty Int^1 = Int^\infty \rightarrow^\infty Int^0 \times^\infty Int^\infty$.

The same rules apply to \triangleright .

Addition is a bit more complex, as addition on function types uses least upper bound on the argument and result types. This models that, even though a function is used repeatedly, the argument and result of the function may still only be used once at each application. Since the argument and result may be constructed anew at each application, these may still be single-threaded even if the function itself is not. Hence, we define

$$\begin{array}{lll}
 int^{k_1} & + int^{k_2} & = int^{(k_1+k_2)} \\
 t_1 \times^{k_1} t_2 & + t_3 \times^{k_2} t_4 & = (t_1 + t_3) \times^{(k_1+k_2)} (t_2 + t_4) \\
 \mu\alpha.^{k_1} c_1 s_1 + \dots + c_n s_n + \mu\alpha.^{k_2} c_1 t_1 + \dots + c_n t_n = & & \\
 & \mu\alpha.^{(k_1+k_2)} c_1 (s_1 + t_1) + \dots + c_n (s_n + t_n) & \\
 t_1 \rightarrow^{k_1} t_2 & + t_3 \rightarrow^{k_2} t_4 & = (t_1 \sqcup t_3) \rightarrow^{(k_1+k_2)} (t_2 \sqcup t_4)
 \end{array}$$

Like the other operators, addition extends to type environments, and here it indicates point-wise addition of the annotated types.

We use the following precedence order on the operators, from tightest to weakest binding: $\triangleright, \cdot, +, \sqcup, \leq$.

4 Type Rules

We now present well-typedness rules for our language. The intention is that in any well-annotated program, the values created at an expression with annotation k will be used at most k times during evaluation under call-by-need. Unlike in [10], we allow the same variable to have different annotations on its type in different parts of the program. The annotations describe the usage in the given subexpression. It is hence the usage count in the type given at the creation time of a value that determines the total number of (potential) uses of the value.

The judgements are of the form

$$E \vdash e : t$$

where E is an environment mapping variables to annotated types, e is an annotated expression and t is an annotated type. The intuitive meaning is that if e is to be evaluated in a context given by t , then E describes the uses of the free variables of e .

If we assume that the result of the entire computation is printed, the natural context for the goal expression has usage 1 on all components of the result type.

In the rules below, we assume that the result of the entire expression is required. For non-strict uses of values, we use the \triangleright guard on the type environments of subexpressions that may not always be evaluated. In this way, laziness is modeled. In the explanation about how the information is passed around, we state an intuitive order of information flow. The actual analysis will not use this order of evaluation but generate a set of constraints that are solved off-line.

Hence, the flow-description is only intended to give an intuitive understanding of the type rules.

We will in general use inequality constraints between environments even in cases where equality would seem natural. The reason is that equality constraints are not in general efficiently solvable and a unique best solution may not exist. This point is explained in section 5.

The variable rule below simply indicates that if a variable is used in context t , then the environment reflects that use.

$$E[x : t] \vdash x : t$$

A constant expression indicates the number of uses of the constant.

$$E \vdash n^k : Int^k$$

If we add two numbers, these are used once each regardless of how many times we use the result.

$$\frac{E_1 \vdash e_1 : Int^1 \quad E_2 \vdash e_2 : Int^1 \quad E_1 + E_2 \leq E}{E \vdash e_1 +^k e_2 : Int^k}$$

The annotation on a pair expression must match the number of uses indicated by the type. The subexpressions are evaluated only if the corresponding parts of the pair are later accessed (indicated by a non-zero count on the types of the components). Hence, we guard the environments of the subexpressions by their types before adding them.

$$\frac{E_1 \vdash e_1 : t_1^{k_1} \quad E_2 \vdash e_2 : t_2^{k_2} \quad k_1 \triangleright E_1 + k_2 \triangleright E_2 \leq E}{E \vdash (e_1,^k e_2) : t_1^{k_1} \times^k t_2^{k_2}}$$

When analysing a *split* expression, we find the contexts of the components by analysing the body in the context of the entire expression. We then combine these to a product type with usage 1 (the usage implied by the *split*) and analyse the splitted expression. Since *split* is strict in both subexpressions, we use no guards.

$$\frac{E_1[x_1 : t_1, x_2 : t_2] \vdash e_2 : t \quad E_2 \vdash e_1 : t_1 \times^1 t_2 \quad E_1 + E_2 \leq E}{E \vdash \text{split } e_1 \text{ to } (x_1, x_2) \text{ in } e_2 : t}$$

When injecting a value into a sum-type, we annotate the constructor by the usage count indicated by the type. Since injection is lazy, we guard the environment for the injected expression by its usage.

$$\frac{E_1 \vdash e : t_i \quad t_i \triangleright E_1 \leq E}{E \vdash c_i^k e : \mu\alpha.{}^k c_1 t_1 + \dots + c_i t_i + \dots + c_n t_n}$$

When analysing a *case*-expression, we analyse each of the branches using the context of the entire expression. The yielded contexts for each of the pattern variables are used to construct a recursive type for the sum-type, constraining

each of the injected types to be conservative approximations of the pattern-variable contexts and adding the constraint that a value of that type may be used at least once (namely, by this *case*-expression). This type is used as the context for the inspected expression. The environment for the entire expression is found as the combined worst-case of the environments of the branches plus the environment of the inspected expression. The i subscript ranges from 1 to n .

$$\frac{E_i[x_i : s_i] \vdash e_i : t \quad s_i \leq t_i[\alpha \setminus (\mu \alpha.^k c_1 t_1 + \dots + c_n t_n)] \quad 1 \leq k}{E_0 \vdash e_0 : \mu \alpha.^k c_1 t_1 + \dots + c_n t_n \quad E_0 + E_i \leq E} \\ E \vdash \text{case } e_0 \text{ of } c_1 x_1 \rightarrow e_1; \dots; c_n x_n \rightarrow e_n : t$$

When analysing a function application, we find the type of the function to get a context for the argument. We also constrain the result type of the function to be at least as “bad” as the context of the application. Since function application is lazy, we guard the environment of the argument expression.

$$\frac{E_1 \vdash e_1 : t_1^k \rightarrow^1 t_2 \quad E_2 \vdash e_2 : t_1^k \quad t_3 \leq t_2 \quad E_1 + k \triangleright E_2 \leq E}{E \vdash e_1 e_2 : t_3}$$

An abstraction is annotated both with the number of uses of the function and by the type of the argument. The body is analysed to find the uses of variables in a single evaluation of the body. Since the body is evaluated at each application of the function, we multiply the environment for the body by the number of uses of the function.

$$\frac{E_1[x : t_1] \vdash e : t_2 \quad k \cdot E_1 \leq E}{E \vdash \lambda^k x : t_1. e : t_1 \rightarrow^k t_2}$$

We analyse the body of a *let*-expression to find the context for the bound variable. This is used as the context for the bound expression and to annotate the variable in the expression. Since *let*-expressions are lazy, we guard the environment of the bound expression by its type.

$$\frac{E_2[x : t_1^k] \vdash e_2 : t_2 \quad E_1 \vdash e : t_1^k \quad k \triangleright E_1 + E_2 \leq E}{E \vdash \text{let } x : t_1^k = e_1 \text{ in } e_2 : t_2}$$

We assume that strongly connected components of a *letrec* are found in an earlier stage of compilation and used to split complex *letrecs* into nested *lets* and *letrecs*, as described in [4]. This allows us to use a fairly simple conservative approximation and assume that the variables bound in a *letrec*-expression can be used arbitrarily often. Hence, we constrain the types of the bound expressions to have infinite counts if they are used at all. The subscript i ranges from 0 to n and j ranges from 1 to n .

$$\frac{E_i[x_1 : t_1, \dots, x_n : t_n] \vdash e_i : s_i \quad \infty \cdot t_j \leq s_j \quad E_0 + \dots + E_n \leq E}{E \vdash \text{letrec } x_1 : t_1 = e_1, \dots, x_n : t_n = e_n \text{ in } e_0 : s_0}$$

5 Usage Analysis

The type rules can be used to check whether a given annotation is valid, but it is more interesting to use the rules to find a minimal valid annotation of a program.

The approach we have chosen is to write a type derivation of the program where all counts are replaced by different variables. This is possible, as type derivations for different annotations of the same program only differ in the values of the counts used in types, environments and annotations. The type rules can then be seen as generating a set of constraints between counts (constraints between types and environments are expanded to collections of constraints over counts). Each constraint is in one of the forms $k_1 = k_2$, $1 \leq k_1$, $k_1 \leq k_2$, $k_1 + k_2 \leq k_3$, $\infty \cdot k_1 \leq k_2$, $k_1 \cdot k_2 \leq k_3$, $k_1 \sqcup k_2 \leq k_3$ or $k_1 \supset k_2 \leq k_3$, where the k_i are variables that range over counts. Each solution to a set of constraints generated by the type rules corresponds to a type derivation for a well-annotated program.

We want minimal annotations, so we must find a minimal solution to the constraint set. We have carefully chosen the constraints to be meet-closed: Whenever we have two solutions, then the meet (greatest lower bound) of these is also a solution. This is the motivation for using $k_1 + k_2 \leq k_3$ where $k_1 + k_2 = k_3$ might seem more natural *etc.*, as the latter is not meet-closed. Hence, a unique minimal solution can be found by taking the meet of all solutions. That a solution must exist is not terribly difficult to see either: All occurrences of constants in the constraints will occur on the left of an inequality, so setting all count variables to ∞ will yield a solution.

Each type rule generates a constant number of constraints between types and type environments. As we have chosen a language where variables are typed explicitly in the program text, the size (measured as number of nodes in the syntax tree) of a type is bounded by the size of the program. Furthermore, the sum of the sizes of types in a type environment is also bounded by the size of the program, as the type environment lists a subset of the variables from the program with their types. Hence, each constraint between types or environments expands to a number of primitive constraints that is bounded linearly by the size of the program. Since the number of inferences in a type derivation is proportional to the size of the program, the total number of primitive constraints is bounded by the square of the size of the typed program. As we will see that the constraints can be solved in time linear in the size of the constraint set, the analysis can be done in quadratic time.

5.1 Solving the Constraints

Rehof and Mogensen [7] show two different ways of solving meet-closed inequality constraints over finite lattices. One method solves the constraints directly, using a variant of Kildall's algorithm [5]. The other translates the constraints into Horn-clauses, which can be solved using the HORNSAT algorithm of Dowling and Gallier [2]. Both methods find the least solution in time linear in the size of

the constraint set. We will show the latter method here and refer readers to [7] for the first.

We represent each element in the count lattice by two boolean values:

count	representation
0	00
1	01
∞	11

Hence, we replace each constraint variable k_i by two variables l_i and r_i over the binary domain, with the constraint $l_i \leq r_i$ to reflect that the pair 10 is unused.

We translate each constraint over counts to a set of constraints over the boolean lattice. The translation below uses the the general scheme shown in [7] with local reduction in the cases where constants occur in the constraints:

constraint	translation
$k_1 = k_2$	$l_1 \leq l_2, r_1 \leq r_2, l_2 \leq l_1, r_2 \leq r_1$
$1 \leq k$	$1 \leq r$
$k_1 \leq k_2$	$l_1 \leq l_2, r_1 \leq r_2$
$k_1 + k_2 \leq k_3$	$l_1 \leq l_3, r_1 \leq r_3, l_2 \leq l_3, r_2 \leq r_3, r_1 \wedge r_2 \leq l_3$
$k_1 \sqcup k_2 \leq k_3$	$l_1 \leq l_3, r_1 \leq r_3, l_2 \leq l_3, r_2 \leq r_3$
$\infty \cdot k_1 \leq k_2$	$r_1 \leq l_2$
$k_1 \cdot k_2 \leq k_3$	$r_1 \wedge r_2 \leq r_3, l_1 \wedge r_2 \leq l_3, l_2 \wedge r_1 \leq l_3$
$k_1 \triangleright k_2 \leq k_3$	$r_1 \wedge l_2 \leq l_3, r_1 \wedge r_2 \leq r_3$

Note that we have reduced the number of different constraints to just ordering (\leq is the usual ordering on the binary domain) and $x \wedge y \leq z$, where $x \wedge y$ is 0 if any of x or y is 0, and 1 otherwise.

It is shown in [7] that, given the extra constraints $l_i \leq r_i$ for all variables k_i , the translation has a solution if and only if the original constraint has. Furthermore, any solution to the translation maps into a solution of the original constraint set using the inverse of the representation function.

6 Conclusion

We can solve these constraints in linear time using a variety of methods. The number of constraints is linear in the size of the type derivation for the program we analyse, i.e. in the worst case quadratic in the size of the typed program. This is somewhat worse than Turner *et al.*'s analysis [10] which is linear in the size of the typed program. Our analysis is, however, more precise: Whenever Turner *et al.*'s analysis detects single use, so will our, and there are cases (e.g. the *mean* example) where our analysis detects single use where Turner *et al.*'s analysis do not.

The presented version of the analysis finds usage counts at value creation times. This is fine if the update information is put into the values themselves in the form of self-updating closures, as is the case for most tagless implementations

of lazy functional languages. If usage counts are required at value destruction time, the presented analysis must be supplemented by a simple forwards analysis. This approach was also used in [3]. Note that this will actually reduce the precision of the analysis, as values with different usages may float to the same destruction point. If annotations are given at creation time, the destruction can treat these differently, but any annotation at the destruction point must treat the values the same way.

The analysis can detect that a partial application is single-use. In addition to optimizing call-by-need to call-by-name, that information can be used to avoid full-laziness transformation of the function, which can save both space and time.

Initial experiments with an implementation of the constraint solver from [7] indicate that solving the constraints directly takes approximately the same time as translating the constraints to Horn-clauses and then using the first of the linear-time HORNSAT algorithms from [2]. However, the latter method uses somewhat more space. These experiments use randomly generated constraints, not constraints generated by an actual usage count analysis, so it is too early to tell which will be best for an actual implementation.

Even though the worst case analysis time is quadratic, we believe this is rarely encountered in real programs, as types and environments typically grow less than linearly in program size. There is, however, little doubt that the analysis of [10] is faster, and it is not clear to what extent the added precision of the proposed analysis is worth the extra cost.

In addition to using the analysis to avoid updates during graph-reduction, it can be used for a limited kind of compile-time garbage collection. This was, indeed, the stated intention of the analysis presented in [3]. In a system with a good garbage collection, this kind of compile-time GC is probably not worth the effort. However, if the typing rules are modified to handle a call-by-value language (essentially eliminating the uses of \triangleright), the analysis can be used in conjunction with region inference [9]. Experience with the ML-kit with regions [8] shows that a common space leak is when a function takes apart a constructor and builds a new constructor of the same type in the same region, causing this region to grow. If the analysis can detect that the original constructor is not used again, the same space can be reused for the new constructor, avoiding the growth.

Another related type system is the uniqueness typing of Clean [1]. This too has the restriction that if the spine of a list is used repeatedly, then it is assumed that this is also the case for the elements. However, the uniqueness type analysis takes evaluation order into account and can hence in some cases detect the last use of value even if there are multiple uses within the life time of the value, which our analysis can not. Furthermore, the uniqueness type system can handle polymorphism, which the analysis presented here does not.

A possible way of handling polymorphism is by abstracting over annotated types and include in each type schema a set of constraints over the type variables (both bound and free). These constraints will be added to the current set of constraints when a type schema is instantiated. When type variables are instantiated to actual types, constraints over the type variables are expanded

out into more primitive constraints. In the end (assuming the goal function of the program has a monomorphic type), all constraints are expanded fully into constraints over atomic counts which can be solved as described above. This approach is likely to have a high worst-case complexity, but may be O.K in practice. It may be worthwhile to reduce the constraints in type schemas at the time they are constructed rather than wait until they are fully instantiated. This will, however, require a different solution mechanism, as the current constraint solver is inherently off-line.

References

- [1] Erik Barendsen and Sjaak Smetsers. Uniqueness type inference. In *PLILP'95, Utrecht, The Netherlands (Lecture Notes in Computer Science, vol. 982)*, pages 189–206. Springer-Verlag LNCS 982, 1995.
- [2] W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.
- [3] Thomas P. Jensen and Torben Æ. Mogensen. A backwards analysis for compile-time garbage collection. In *ESOP '90, Copenhagen, Denmark (Lecture Notes in Computer Science, vol. 432)*, pages 227–239. Springer-Verlag LNCS 432, 1990.
- [4] Simon L. Peyton Jones and David Lester. *Implementing Functional Languages*. Prentice Hall Series in Computer Science. Prentice Hall, New York, London, Toronto, Syney, Tokyo, Singapore, 1 edition, 1992.
- [5] G. Kildall. A unified approach to global program optimization. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 194–206. ACM Press, January 1993.
- [6] M.J. Plasmeijer and M.C.J.D. van Eekelen. Concurrent clean 1.0 language report. Technical report, Computing Science Institute, University of Nijmegen, 1995.
- [7] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semi-lattices. In R. Cousot and D.A. Schmidt, editors, *Third International Static Analysis Symposium (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 285–301. Springer, September 1996.
- [8] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeldt Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit. Technical report, Dept. of Computer Science, University of Copenhagen, 1997.
- [9] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, January 1994.
- [10] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *7th International Conference on Functional Programming and Computer Architecture*, pages 1–11, La Jolla, California, June 1995. ACM Press.

Fully Persistent Graphs – Which One To Choose?

Martin Erwig

FernUniversität Hagen, Praktische Informatik IV
D-58084 Hagen, Germany
erwig@fernuni-hagen.de

Abstract. Functional programs, by nature, operate on functional, or persistent, data structures. Therefore, persistent graphs are a prerequisite to express functional graph algorithms. In this paper we describe two implementations of persistent graphs and compare their running times on different graph problems. Both data structures essentially represent graphs as adjacency lists. The first uses the version tree implementation of functional arrays to make adjacency lists persistent. An array cache of the newest graph version together with a time stamping technique for speeding up deletions makes it asymptotically optimal for a class of graph algorithms that use graphs in a single-threaded way. The second approach uses balanced search trees to store adjacency lists. For both structures we also consider several variations, for example, ignoring edge labels or predecessor information.

1 Introduction

A data structure is called *persistent* if it is possible to access old versions after updates. It is called *partially persistent* if old versions can only be read, and it is called *fully persistent* if old versions can also be changed [4]. There is a growing interest in persistent data structures, for a recent overview, see [9]. However, persistent graphs have almost been ignored. In [6] we have sketched an implementation of unlabeled, fixed-size persistent graphs by functional arrays. The purpose of that paper was, however, to demonstrate an inductive view of graphs and a corresponding functional style of writing graph algorithms, mainly based on graph fold operations, and to show how this style facilitates reasoning about and optimization of graph algorithms.

In this paper we explain the implementation by functional arrays in more detail, and we extend it in several ways. First, we are now able to work with labeled graphs. Second, the implementation can also be used in semi-dynamic situations, that is, new nodes can be efficiently allocated, and graphs can grow to a limited degree. This has become possible through an extension of the graph representation by node partitions which are realized in a way similar to the implementation of sparse sets described in [3]. Third, we consider several specialized implementations: one for unlabeled graphs, one keeping only successor information, and one for a combination of both. Moreover, the underlying functional array implementation has been improved.

Besides the explanation of the persistent graph implementation based on functional arrays, the main goal of this paper is to find a good “standard representation” suitable for most application scenarios. We have therefore implemented persistent graphs also on the basis of balanced binary search trees. We present some example programs and report running times of the different graph implementations. We also pay attention to the question of whether specialized implementations (unlabeled, keeping only successors) are worthwhile. The results of this paper have helped us in the design of a *functional graph library*, which has been implemented in Standard ML and which can be downloaded from

<http://www.fernuni-hagen.de/inf/pi4/erwig/fgl>

All examples of this paper are contained in the distribution. The main contributions of this work are:

- (1) The world’s first functional graph library
- (2) Several implementations of persistent graphs
- (3) Empirical results about the performance of different persistent graph structures

Section 2 introduces a compact graph data type whose operations are briefly demonstrated with examples taken from graph reduction. These examples are used in Section 3 to explain the different implementations of persistent graphs. In Section 4 we describe a set of test programs and report running times of the different graph implementations. Conclusions follow in Section 5.

2 A Data Type for Graphs

In the following we consider directed node- and edge-labeled multi-graphs. This is a sufficiently general model of graphs, and many other graph types can be obtained as special cases: for instance, undirected graphs can be represented by symmetric directed graphs where “symmetric” means that presence of edge (v, w) implies the existence of edge (w, v) . Unlabeled graphs have node and/or edge label type `unit`, and graphs embedded in the Euclidean plane can be modeled by having `real * real` node type.

Typical operations on graphs are the creation of an empty graph, adding, retrieving, and deleting nodes and edges, and retrieving and changing node and edge labels. We can cover all these functions by a simple interface consisting of just three operations. We have a type for nodes, which we assume for simplicity to be `int`, and a type for graphs whose type parameters `'a` and `'b` denote the type of node and edge labels, respectively.

```
type node = int
type ('a,'b) graph
```

Additionally, we use the following type abbreviations that make the typings of some operations more concise.

```
type      'b adj      = ('b * node) list
type ('a,'b) context = 'b adj * node * 'a * 'b adj
```

Concerning operations, we have a constant¹ `empty` representing the empty graph, an operation `embed` that extends a graph by a new node together with incoming and outgoing edges, and an operation `match` that retrieves and at the same time removes a node with all its incident edges from the graph.

```
val empty : ('a,'b) graph
val embed : ('a,'b) context * ('a,'b) graph -> ('a,'b) graph
val match : node * ('a,'b) graph -> ('a,'b)context * ('a,'b)graph
```

Since graphs are not freely constructed by `empty` and `embed`, that is, since there are different term representations denoting the same graph, matching is, in general, not uniquely defined. The additional node parameter allows us to specify which representation is to be selected (namely that one with the given node inserted last) and thus makes `match` a function again; `match` is actually an example of an *active pattern* as described in [5]. If the node to be matched is not in the graph, a `Match` exception is raised.²

Consider, for example, the graph `g` in Figure 1 that represents the combinator expression $(sqr\ 3) + (sqr\ 3)$. The node numbers are given for later reference.

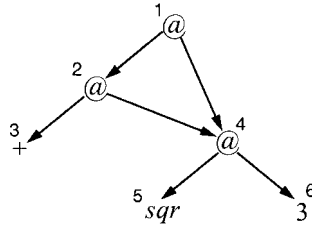


Fig. 1. Example Graph `g`

The type of `g` is `(combinator,direction) graph` with

```
datatype value = INT of int | BOOL of bool
datatype direction = L | R
```

¹ Note that in the array implementation `empty` takes an integer argument specifying the maximum graph size.

² The reader might wonder why the argument node of `match` is also returned as a result (as part of the type `context`). It is for consistency with the operation `matchAny` (see Section 4.2) that does not take a node argument and therefore reports the node actually matched.

```

datatype combinator =
  APP
| COND
| VAL of value
| OP of (value * value -> value)
| COMB of string

```

We have omitted the edge labels from the picture since the values are implied by the spatial embedding of the edges. The following expression constructs g in a bottom-up manner.

```

val g = foldr embed empty
  [ ([],1,APP,[ (L,2),(R,4)]), ([],2,APP,[ (L,3),(R,4)]),
    ([],3,OP plus,[ ]), ([],4,APP,[ (L,5),(R,6)]),
    ([],5,OP sqr,[ ]), ([],6,VAL (INT 3),[ ]) ]

```

As already indicated this is indeed not the only way to build the graph g . Actually, we can insert the nodes in any order. For a precise definition of the semantics of graph constructors, see [6].

Suppose now we are to reduce g . We first have to reduce the subgraph rooted at node 4, and replace it with the result of the reduction. Thus, we first match node 4, and we get the context

$$(([(R,1),(R,2)],4,APP,[(L,5),(R,6)]), g')$$

where g' is any representation of the graph:

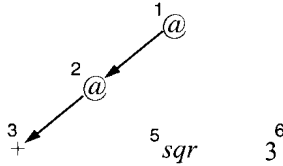


Fig. 2. Decomposed Graph g'

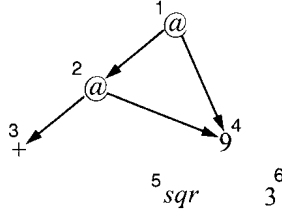
In this case we have to apply a δ -rule for computing $sqr\ 3$, and we re-insert node 4 with the result as the new node label, with the old predecessors, and with no successors, that is,

```

val r = embed (([(R,1),(R,2)],4,VAL (INT 9),[ ]),g')

```

Thus we obtain the graph shown in Figure 3 in which nodes 5 and 6 have become garbage. Note that after the update g is still bound to the original graph; graph reduction as described here happens purely functionally, that is, node 4 is not destructively overwritten.

**Fig. 3.** Reduced Graph r

Summarizing we see that `embed` implements node and edge insertion and that `match` comprises the operations of finding and deleting nodes and edges. A `match` followed by `embed` can be used to realize update functions. It is not difficult to derive specific graph operations from these general ones. For instance, the insertion of a single labeled edge can be simply done by:

```

fun insEdge (v,w,l) g =
  let val ((p,_,vl,s),g') = match (v,g)
  in
    embed (p,v,vl,(l,w)::s,g')
  end

```

We will see the need for some more predefined graph operations when considering example programs in Section 4. Mostly on the grounds of efficiency we also provide the following functions.

```

val matchFwd : ('a,'b) graph -> 'a * 'b out
val matchAny : ('a,'b) graph -> ('a,'b) context * ('a,'b) graph
val isEmpty : ('a,'b) graph -> bool
val newNodes : int -> ('a,'b) graph -> node list

```

The functions `matchFwd` and `matchAny` are just special versions of `match`: `matchFwd` selects only the node label and the successors, and `matchAny` matches an arbitrary node context. This means that `matchAny` is non-deterministic and can assume any valid term representation of its argument graph to return the outermost node context of this representation. The function `isEmpty` tests whether a graph is the empty graph, and `newNodes i g` generates a sequence of i nodes that are not contained in the graph g .

3 Persistent Graph Structures

The representation of a graph by (an array of) adjacency lists is often favored over the incidence matrix representation since its space requirement is linear in the graph size whereas an incidence matrix always needs $\Omega(n^2)$ space which is very wasteful for sparse graphs. Moreover, adjacency lists offer $O(k)$ access time to the k successors of an arbitrary node where we have to spend $\Omega(n)$ time to find the successors by

scanning a complete row in an incidence matrix. We therefore focus on two alternatives for making adjacency lists persistent: the first representation uses a variant of the version tree implementation of functional arrays, and the second stores successor (and predecessor) lists in a balanced binary search tree. The functional array structure is more difficult to understand because it employs an additional cache array and a further array carrying a kind of time stamps for nodes. Moreover, to support some of the specialized operations efficiently, this structure is extended by a two-array implementation of node partitions to keep track of inserted and deleted nodes. We therefore explain this structure in some detail and only sketch the rather obvious binary tree implementation.

3.1 Implementation by Functional Arrays

In a first attempt we can make an imperative graph persistent by simply using a functional array for storing adjacency lists and node labels. However, with this representation the deletion of a node v from the graph (which is performed with every `match` operation) is quite complex, since we have to remove v from each of its predecessor's successor lists and from each of its successor's predecessor lists. This means, on the average, quadratic effort in the size of node contexts. To avoid this we therefore store with each node a positive integer when the node is present in the graph and a negative integer when the node is deleted. We also carry over positive node stamps into successor/predecessor lists. This has the following effect: When, for example, node v is deleted, we can simply set its stamp $s(v)$ to $-s(v)$; we need not remove v from all referencing successor and predecessor lists because when, say, a successor list l of w containing v is accessed, we can filter out all elements that have non-matching stamps, and by this v will not be returned as a successor. When v is re-inserted into the graph later, we set $s(v)$ to $|s(v)|+1$, and take this new stamp over to all newly added predecessors and successors. Now if w is not among the new predecessors, the old entry in l with stamp $s(v)$ is still correctly ignored when l is accessed.

In the version tree implementation of functional arrays as described in [2] changes to the original array are recorded in an inward directed tree of *(index, value)* pairs that has the original array at its root. Each different array version is represented by a pointer to a node in the version tree, and the nodes along the path to the root mask older definitions in the original array (and the tree). Adding a new node to the version tree can be done in constant time, but index access might take up to u steps where u denotes the number of updates to the array. By adding an imperative "cache" array to the leftmost node of the version tree the array represented by that node is actually duplicated. Since index access in this array is possible in constant time, algorithms that use the functional array in a single-threaded way have the same complexity as in the imperative case, since the version tree degenerates to a "left spine" path offering $O(1)$ access all the time during the algorithm run.

There is a subtlety in this implementation having just *one* cache array: if a functional array is used a second time, the cache has already been consumed for the previous computation and cannot be used again. This gives a surprising time

behavior: the user executes a program on a functional array, and it runs quite fast. However, running the same program again results, in general, in much larger execution times since all access now goes via the version tree. Therefore, we create in our implementation a new cache for each new version derived from the original array.

An initial functional array representation of \mathcal{g} is shown in Figure 4.

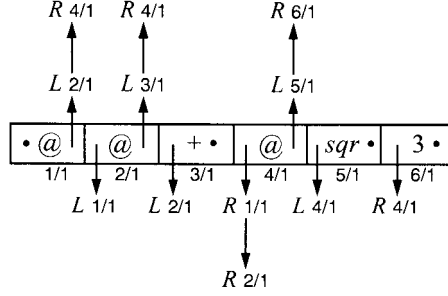


Fig. 4. Functional Array Representation of Graph \mathcal{g}

Each array entry consists of a list of predecessors (extending downwards), a node label, and a list of successors (extending upwards). Empty lists are represented by \bullet . Note that initially all node stamps are set to 1.

Now consider the result of the expression `match (4, \mathcal{g})`. First, the predecessor list, the node and its label, and the successor list are retrieved, and then the decomposed graph \mathcal{g}' is constructed. Here, it suffices to add a single node to the version tree indicating that node 4 is deleted. If we had not the node stamps available, we would have to add four (!) more nodes to the version tree specifying the change of the successor lists (for nodes 1 and 2), respectively, predecessor lists (for nodes 5 and 6). The updated graph structure is shown in Figure 5.³

If we now access, say, the successors of node 2, we obtain just the list $[(L, 3)]$ – node 4 is omitted, since the node stamp found in the successor list does not match the stamp currently stored with the node.

Let us finally consider what happens if we re-insert node 4 to obtain the reduced graph \mathcal{r} from Figure 3. First, the stamp of node 4 changes to +2, and a modification node with empty successor list and the old predecessors is added to the version tree. Unfortunately, we are not finished yet: we have to add node 4 to the successor lists of each of its predecessors, that is, we have to create corresponding nodes in the version tree. Note that in both these successor lists node 4 appears twice, but only the entry with the new stamp is relevant. The resulting graph representation is shown in Figure 6.

³ Note that the actual implementation differs in a minor point: instead of one array storing four-tuples we are working with four individual arrays for stamps, labels, predecessors, and successors. This simplifies the implementation at some points. On the other hand, the structure is easier to explain using only one array.

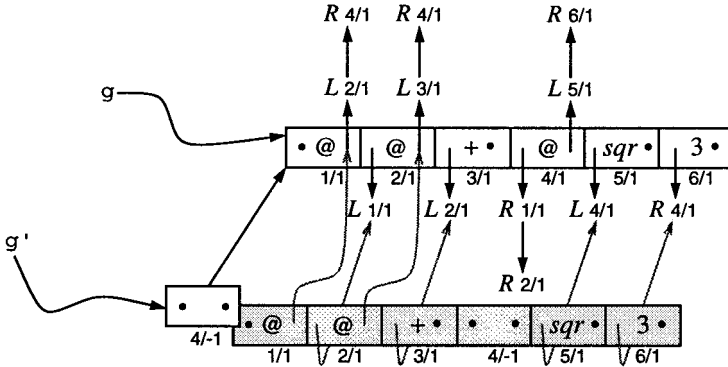


Fig. 5. Functional Array Representation of Graph g'

In principle, we had to update the predecessor lists for all successors, too, but this does not apply in this particular case, since the successor list is empty. The described updates actually use the array in a single-threaded way so that the version tree degenerates to a path with the cache at its end. If we now perform a further update to the original graph g , this is reflected in the version tree by a new sibling node of g' . Since this version was created from the original version g , the node is also equipped with its own cache array to speed up single threaded operations on this version, too. This new cache is needed, since the other cache is still in use.

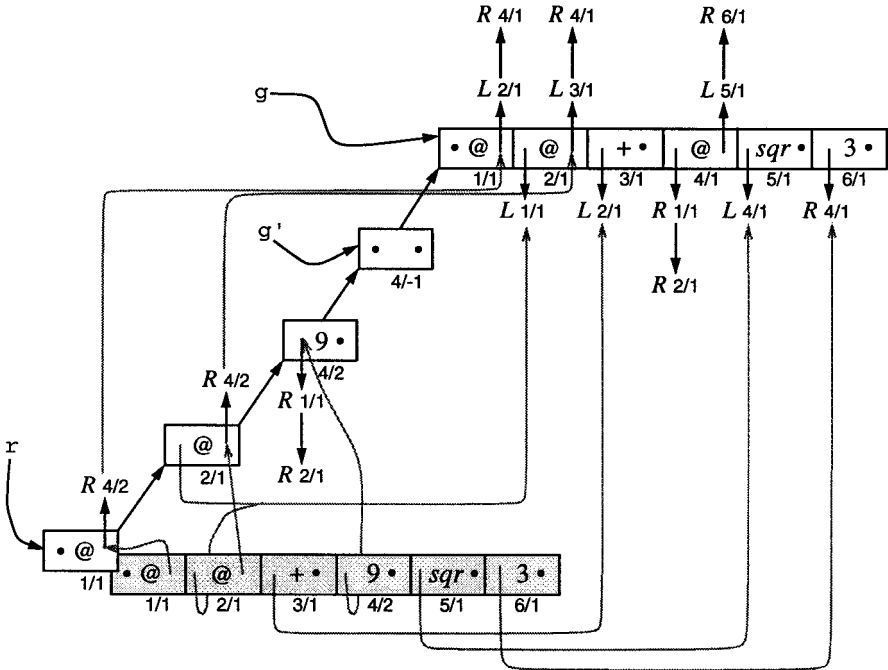


Fig. 6. Functional Array Representation of the Reduced Graph r

The array representation is expected to perform quite well in situations where graphs are used in a single-threaded way. However, in cases when the cache cannot be used the need for traversing possibly long paths in the version tree can slow down the structure significantly. Actually, few applications use graphs in a persistent manner. Maybe this is due to the fact that graph algorithms are traditionally expressed in an imperative style? In any case, a trivial, yet important, persistent use of graphs is given by executing more than one algorithm on a graph. By spending a new cache for each “primary” version, the functional array implementation is well prepared for these situations, too.

The functional array implementation is fine for the (minimal) interface described in Section 2. However, some algorithms require operations that cannot be easily reduced to the three basic ones and should therefore also be provided by a library. For instance, a simple algorithm for reversing the edges in a graph (see Section 4.2) uses the functions `matchAny` and `isEmpty`. Now in the array representation an implementation of `matchAny` is quite inefficient, since, in general, we have to scan the whole stamp array to find a valid, that is, non-deleted, node. The same is true for `isEmpty` and also for `newNodes`. Here the simple array implementation also requires, in general, linear time by scanning the whole array.

Thus to reasonably use the array implementation in those cases we have to extend the implementation to support these two operations in constant time.

We therefore keep for each graph a partition of inserted nodes (that is, nodes existent in the graph) and deleted nodes: when a node is deleted (decomposed), it is moved from the inserted-set into the deleted-set, when a node is inserted into the graph, it is moved the other way. The node partition is realized by two arrays, *index* and *elem*, and an integer *k* giving the number of existent nodes, or, equivalently, pointing to the last existing node. The array *elem* stores all existent nodes in its left part and all deleted nodes in its right part, and *index* gives for each node its position in the *elem* array. A node *v* is existent if $index[v] \leq k$, likewise, it is deleted if $index[v] > k$. Inserting a new node *v* means to move it from the deleted-set into the inserted-set. This is done by exchanging *v*’s position in *elem* with the node stored at *elem*[*k*+1] (that is, the first deleted node) followed by increasing *k* by 1. The entries in *index* must be updated accordingly. To delete node *v*, first swap *v* and *elem*[*k*], and then decrease *k* by 1. All this is possible in constant time.

Now all the above mentioned graph operations can be implemented to work in constant time: `matchAny` can be realized by calling `match` with *elem*[1], `isEmpty` is true if *k*=0, and `newNodes` i can simply return a list of nodes [*elem*[*k*+1], ..., *elem*[*k*+i]]. Some other useful graph operations are efficiently supported by the node partition: *k* gives the number of nodes in the graph, and all nodes can be reported in time $O(k)$ which might be much less than the size of the array. The described implementation of node partitions is an extension of the sparse set technique proposed in [3].

Although offering some nice features, the described extension has its drawbacks and limits: Keeping the partition information requires additional space and causes some overhead. Moreover, arrays do not become truly dynamic since they (at least in the current implementation) can neither grow nor shrink. In the following we will

report test results for both the static array implementation (*Array*) and also for the semi-dynamic version (*DArray*) whenever this is possible.

The attentive reader might wonder whether the garbage nodes in successor and predecessor lists (that is, invalid and unused references to deleted nodes) are a source of inefficiency. Although we have not studied the problem in detail, it seems that in practice, this is not a problem. For example, in the case of graph reduction, where graphs are heavily updated, only 25-30% of nodes in successor and predecessor lists are filtered out due to invalid stamps.

3.2 Implementation by Balanced Binary Search Trees

We use the implementation of balanced binary search trees as provided by the SML/NJ library. Actually, this is the implementation described in [1]. For convenience, we have added two functions `update` and `findSome`. Although `update` can be realized by `remove` and `insert`, the directly implemented version saves multiple traversals of the same path.

Since a binary search tree can be used as a functional array implementation, we obtain an immediate realization of functional graphs, that is, a graph is represented by a pair (t, n) where t is a tree of pairs $(node, (predecessors, label, successors))$ and n is the largest node occurring in t . Note that n is used to support the operation `newNodes`. Keeping the largest node value used in the graph, this is possible in $O(1)$ time.

3.3 Tuned Implementations

There are two principal sources for improving efficiency of any graph implementation: first, carrying edge labels in unlabeled graphs wastes unnecessarily space and causes some overhead. Second, keeping predecessor information is not necessary in algorithms that only access successors.

So in addition to the labeled/full-context array and tree implementations we have implemented unlabeled/full-context, labeled/forward, and unlabeled/forward versions to see which speed-ups can be achieved. It should be clear that the `match` operation is not available in the forward implementations. Instead an operation `matchFwd` giving only node label and successors is provided. Note that even `matchFwd` cannot be (efficiently) realized in a forward-tree implementation, since removing a node from a graph means to remove it from all referencing successor lists, but without the predecessor information, which provides direct access to all referencing nodes, all successor lists of the graph had to be scanned, which is not acceptable.

More extensions and improvements are conceivable, for example, combining the stamping technique with the tree implementation or making functional arrays fully dynamic. These will be examined in the near future.

4 Test Programs and Running Times

We have selected the following example programs to get a picture of the graph implementations' performances in different situations: depth-first search (`dfs`,

Section 4.1), reversing the edges of a graph (`rev`, Section 4.2), generating cliques (`clique`, Section 4.3), combinator graph reduction (`red`, Section 4.4), and computing maximal independent sets of nodes (`indep`, Section 4.5). We cannot use the benchmarking kit Auburn [8] since (i) it is restricted to unary type constructors (labeled graphs have two type parameters), and (ii) argument and result types must not contain applications of other type constructors (`embed` and `match` take/yield *lists* of nodes (and labels)).

All programs have been run with SML of New Jersey 109.29 under Solaris 2 on a SUN SPARCstation 20 with 64MB main memory. Reported are the average user times (average taken over ten⁴ runs). The benchmarks are intended to serve two goals: first, we would like to find out which graph implementation performs best in which scenario, and second, whether it is worthwhile to use specialized implementations in situations where only part of the functionality is needed.

With regard to the latter we consider three aspects: (1) *Kind of context*. This addresses the fact that there are graph algorithms (for example, *dfs*) that move only in forward direction through the graph. For all these, a graph representation that does not explicitly store predecessors is more space efficient and is expected to be faster because of less overhead. (2) *Edge Labels*. Many graph algorithms (for example, *dfs*, *rev*) do not need edge labels, that is, they also work on graphs without edge labels, and the question is whether they perform faster on a specialized graph type for unlabeled graphs. (3) *Graph Size*. For algorithms that do not change or at least do not increase the size of the graph the static array representation (without node partitions) is expected to run faster than semi-dynamic arrays.

For the comparison of the two main implementations we distinguish graph algorithms by the needed *operations on graphs*. Many algorithms use graphs in a “read-only” fashion, that is, they only decompose the graph by means of the `match` operation. On the other hand, there are algorithms that build or change graphs, and the performance picture might be different here. Algorithms for this case can be distinguished further according to whether they are *dynamic*, that is, generate new nodes and truly extend a graph, or *static*, that is, just perform operations on existing nodes and edges. Finally, although most algorithms seem to use graphs in a single-threaded way, there are examples of persistent graph uses, and it is interesting to know which structures are preferable in those situations.

The test programs provide examples for each kind of graph use, operation, and specialization:

⁴ Some implementations crashed due to memory shortage on large graphs. In those cases averages over three or five runs have been taken.

<i>Graph Use</i>	<i>Operation</i>	<i>Specialization</i>	<i>Program</i>
<i>single-threaded</i>	match	<i>forward, unlab</i>	dfs
	embed (<i>static</i>)	<i>unlab</i>	rev
	embed (<i>dynamic</i>)	<i>unlab</i>	clique
			red
<i>persistent</i>	embed	<i>unlab</i>	indep

The test results provide some advice of the kind “If you have a graph problem of type *X*, then it is best to use graph implementation *Y*.”

4.1 Depth First Search

One of the most important general graph algorithm is certainly depth first search. We consider the task of computing a depth first spanning forest for a graph. We choose forests storing only nodes without labels; they are represented by:

```
datatype tree = BRANCH of node * tree list
```

Now a functional version of depth first search is defined by the two mutual recursive functions `dfs1` and `dfs` which work as follows: First, `dfs1` decomposes `v` from `g` getting the list of successors (and node labels) `s` and the reduced graph `g1`. Then for each element of `s` (that is, for each second component) a depth first spanning tree is computed recursively by calling `dfs`. Note that by using `g1` as the graph argument of `dfs` it is ensured that `v` cannot be encountered again in a recursive call to `dfs1`. Finally, the constructor `BRANCH` is applied to `v` and to the resulting forest `f` yielding the spanning tree of `v`. In addition, the part of the graph not visited by the recursion is returned. Now `dfs` does nothing more than calling `dfs1` for each node of its argument list, passing around decomposed graphs.

```
fun dfs1 (v,g) =
  let val ((_,_,s),g1) = match (v,g)
      val (f,g2) = dfs (map (fn (_,w)=>w) s,g1)
  in
    (BRANCH (v,f),g2)
  end
and dfs ([],g) = ([],g)
  | dfs (v::l,g) =
    let val (t,g1) = dfs1 (v,g)
        val (f,g2) = dfs (l,g1)
    in
      (t::f,g2)
    end
handle Match => dfs ([],g)
```

Since the graph might consist of unconnected components, the depth-first spanning forest is obtained by applying `dfs` to all nodes of the graph:

```
fun dfs g = dfsn (nodes g,g)
```

The running times for `dfs` (user time in seconds, including garbage collection) are given in Table 1 for sparse graphs (of average degree 8) depending on the number of nodes in the graph and the chosen graph implementation.

	1 000	5 000	10 000	50 000	100 000	<i>Ratios</i>
<i>Array</i>	0.08	0.53	1.12	9.75	21.42	1
<i>DArray</i>	0.14	0.77	1.79	16.68	36.98	1.45 .. 1.75
<i>Tree</i>	0.25	1.63	3.62	29.61	67.01	3.04 .. 3.23

Table 1. Running Times for `dfs`

We see that both array implementations are always faster than the tree implementation and that *DArray* tends to take 70% more time than *Array*. It is interesting that *Tree* performs quite well with only a factor of about 3 slower than *Array*.

In Table 2 we show running times for unlabeled (*u*) and forward only (*f*) specialized implementations. We observe that omitting labels yields a speed-up of at least 7% (*Array*) or 12% (*Tree*) up to 20% (for both *Tree* and *Array*), and just keeping forward links (which is only possible with the array implementation) is about 3 to 18% faster. The unlabeled/forward implementation gives, as expected, the best performance and is 13 to 25% faster.

	1 000	5 000	10 000	50 000	100 000	<i>Ratios</i>
<i>Array</i>	0.10	0.53	1.12	9.75	21.42	1
<i>Array</i> ^{<i>u</i>}	0.07	0.48	0.88	7.73	20.00	0.79 .. 0.93
<i>Array</i> ^{<i>f</i>}	0.07	0.51	0.99	9.46	20.66	0.88 .. 0.97
<i>Array</i> ^{<i>uf</i>}	0.06	0.41	0.84	7.53	18.70	0.75 .. 0.87
<i>Tree</i>	0.25	1.63	3.62	29.61	67.01	1
<i>Tree</i> ^{<i>u</i>}	0.20	1.31	2.94	26.01	55.41	0.80 .. 0.88

Table 2. Speed-ups gained by Unlabeled and Forward Graphs

The improvements in running time obtained by the specialized implementations are similar for the other examples programs, so we shall not give any additional tables in the following.

4.2 Reversing Graphs

The second graph problem considered is to reverse the edges of a graph. The following simple algorithm can be used:

```
fun rev g =
  if isEmpty g then g else
  let val ((p,v,l,s),g') = matchAny g
  in
    embed ((s,v,l,p),rev g')
  end
```

We have already noticed that `isEmpty` and `matchAny` are efficiently supported only by the dynamic array implementation. Thus we omit the running times for the static version, since this would make `rev` actually a quadratic algorithm. The graphs used are the same as for the `dfs` tests.

	1 000	5 000	10 000	50 000	100 000	<i>Ratios</i>
<i>DArray</i>	0.27	2.12	4.35	47.27	120.59	1.04 .. 2.55
<i>Tree</i>	0.26	1.68	3.33	21.24	47.38	1

Table 3. Running Times for `rev`

It is striking that *Tree* outperforms the array implementation. This is certainly due to the fact that `matchAny` is realized by just taking the tree's root, which is very fast and, in particular, creates little garbage. It is therefore interesting to consider in this case the running times without garbage collection. Here the figure changes, and the tree implementation takes up to 50% more time, see Table 4.

	1 000	5 000	10 000	50 000	100 000	<i>Ratios</i>
<i>DArray</i>	0.23	1.18	2.37	12.88	24.32	1
<i>Tree</i>	0.24	1.42	2.88	16.79	36.36	1.04 .. 1.50

Table 4. Command Times for `rev` (without GC)

At this point it is important to mention that the test program for iterating the *DArray* implementation exhausted heap memory already on graphs with 50000 nodes, and we were forced to take the average only over three runs. All in all this shows that worst-case running times given in big-Oh notation are one aspect and that actual running times as well as practicability and reliability of implementations are often a different matter.

Note that we can efficiently reverse graphs with the static array implementation if we rewrite the algorithm as follows.


```

fun revl ([],g)   = g
|  revl (v::l,g) =
    let val ((p,_,lab,s),g') = match (v,g)
    in
        embed ((s,v,lab,p),revl (l,g'))
    end
    handle Match => g

fun revd g = revl (nodes g,g)

```

By avoiding the functions `matchAny` and `isEmpty` we obtain a linear algorithm. Deterministic matching causes *Tree* to spend more time on searching node contexts, and the static array implementation is again ahead, see Table 5.

	1 000	5 000	10 000	50 000	100 000	<i>Ratios</i>
<i>Array</i>	0.21	1.32	3.27	33.66	73.57	1
<i>Tree</i>	0.35	2.43	6.55	39.19	83.89	1.14 .. 2.00

Table 5. Running Times for `revd`

So `rev` is only of limited use in judging the implementations' efficiency for graph construction.

4.3 Generating Cliques

For a better comparison of the implementations of `embed`, we therefore use the following program⁵ for generating cliques of a specified size (`map (fn x=>((),x))` just extends the list of nodes by a unit edge label):

```

fun clique 0 = empty
|  clique n =
    let val g   = clique (n-1)
        val l   = map (fn x=>(( ),x)) (nodes g)
        val [v] = newNodes 1 g
    in
        embed ((l,v,(),l),g)
    end

```

The results in Table 6 show that the tree implementation performs better in constructing graphs.

⁵ For testing *DArray* we need a slightly different program taking into account that `empty` needs `n` as a size parameter.

	50	100	500	1 000	<i>Ratios</i>
<i>DArray</i>	0.05	0.22	18.15	85.67	1.14 .. 2.00
<i>Tree</i>	0.03	0.17	8.45	46.69	1

Table 6. Running Times for clique

4.4 Combinator Graph Reduction

The graph type to be used for graph reduction was already given in Section 2. For lack of space we do not give here the complete code of the graph reducer.⁶ We have implemented combinator graph reduction as described in Chapter 12 of [7]. The main components are the functions `unwind` and `red`: unwinding the graph's spine yields the combinator to be reduced together with all argument nodes and the root node to be replaced. The function `red` actually contains the reduction rules for all combinators. Consider the case for the *K* combinator: first, the argument nodes and the root node as given by `unwind` are bound to the variables `x`, `y`, and `r`. Then the root node `r` is matched in the graph `e` which is to be reduced. This is just to decompose `r` from `e` and to remember the references to it in the list `rp`. After that the label (`xl`) and the successors (`xs`) of `x` are determined by applying the function `fwd`. Note that `match` need not be used here, since `x` is not to be changed. Finally, node `r` is re-inserted, now with label and successors of `x`, but with its own predecessors (`rp`). This realizes, in a functional way, the “overwriting” of `r`. The situation is similar for the *S* combinator: bindings are created, the root is decomposed, and the new graph is built by node insertion. Note, however, that two new nodes (`n` and `m`) have to be generated. Performing this node generation *before* matching of `r` takes place ensures a single-threaded use of the graph, and we could, in principle, use an imperative graph implementation for the graph reducer. In contrast, if new nodes were generated in an imperative graph after matching `r`, `r` could be returned (“recycled”) as one of the new nodes which would definitely lead to wrong results.

```

fun red (v,e) =
  let val (comb,args) = unwind (v,[],e)
  in
    case comb of
      COMB "K" =>
        let val [x,y,r] = args
          val ((rp,_,_),e') = match (r,e)
          val (xl,xs) = fwd (x,e')
        in
          embed ((rp,r,xl,xs),e')
        end
      | COMB "S" =>

```

⁶ The code for all examples can be found in the FGL distribution.

```

let val [f,g,x,r] = args
    val [n,m] = newNodes 2 e
    val ((rp,_,_,_),e') = match (r,e)
in
    embed ((rp,r,APP,[ (L,n),(R,m) ]),
    embed ([],n,APP,[ (L,f),(R,x) ]),
    embed ([],m,APP,[ (L,g),(R,x) ]),e'))
end
...
end

```

We give the running times for the reduction of graphs that represent applications of the fibonacci function on different arguments. We know that fibonacci has been criticized as a benchmark for graph reduction, but we are testing the underlying graph implementation, not the graph reducer, and for that fibonacci is sufficient by causing reductions and allocations. Unfortunately, *DArray* exhausts heap memory even for a single run when computing `fib 20`. For the smaller sizes, the tree implementation is clearly faster.

	fib 10	fib 15	fib 20	<i>Ratios</i>
<i>DArray</i>	0.52	11.91	—	1.33 .. 1.79
<i>Tree</i>	0.39	6.66	289.93	1
<i>Reductions</i>	1 031	11 576	128 521	
<i>Allocations</i>	814	9 139	101 464	

Table 7. Running Times for `red`

4.5 Maximal Independent Node Sets

The function `indep` for computing sets of non-adjacent nodes of maximal size is an example of an algorithm that uses graphs in a truly persistent way. Although the algorithm has exponential complexity (the problem is NP-hard) it is much faster than blindly trying all possible node subsets: in the second line a node `n` with maximum degree is determined. Then two node sets are computed, namely, the independent set of `g` simply without `n`, and `n` plus the independent set of `g` without `n` and all its neighbors. The larger of the two sets is the result.

```

fun indep g =
  if isEmpty g then [] else
  let val n = any (max,deg g) (nodes g)
      val ((p,_,_,l),g1) = match (n,g)
      val i1 = indep g1
      val i2 = n::indep (delNodes (l@p,g1))
  in
    if length i1 > length i2 then i1 else i2
  end
end

```

Looking at the results in Table 8 it is striking that *Array* behaves very poorly on larger graphs. The reason is that `nodes` is linear in the size of the representation (that is, the size of the starting graph) and not linear in the number of nodes of the (usually much smaller) graph in the current recursion.

	15	20	50	<i>Ratios</i>
<i>Array</i>	0.02	4.00	2264.89	2.00 .. 8.68
<i>DArray</i>	0.02	1.85	555.34	1.97 .. 2.13
<i>Tree</i>	0.01	0.94	261.02	1

Table 8. Running Times for `indep`

5 Conclusions

The test results indicate that the tree implementation should be used as the primary representation of functional graphs. Although it is in some cases slower than the functional array implementation, it is more versatile and more reliable on large graphs. The tree implementation can also be easily translated into Haskell making the graph library accessible to a broader range of users, although the running times might be quite different due to the effects of lazy evaluation.

References

1. Adams, S.: Efficient Sets – A Balancing Act. *Journal of Functional Programming* **3** (1993) 553–561
2. Aasa, A., Holström, S., Nilsson, C.: An Efficiency Comparison of Some Representations of Purely Functional Arrays, *BIT* **28** (1988) 490–503
3. Briggs, P., Torczon, L.: An Efficient Representation for Sparse Sets, *ACM Letters on Programming Languages* **2** (1993) 59–69
4. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making Data Structures Persistent. *Journal of Computer and System Sciences* **38** (1989) 86–124
5. Erwig, M.: Active Patterns. *8th Int. Workshop on Implementation of Functional Languages*, LNCS 1268 (1996) 21–40
6. Erwig, M.: Functional Programming with Graphs. *2nd ACM SIGPLAN Int. Conf. on Functional Programming* (1997) 52–65
7. Field, A.J., Harrison, P.G.: *Functional Programming*. Addison-Wesley, Wokingham, England (1988)
8. Moss, G.E., Runciman, C.: Auburn: A Kit for Benchmarking Functional Data Structures. *9th Int. Workshop on Implementation of Functional Languages*, this LNCS volume (1997)
9. Okasaki, C.: Functional Data Structures, *Advanced Functional Programming*, LNCS 1129 (1996) 131–158

Auburn: A Kit for Benchmarking Functional Data Structures

Graeme E. Moss and Colin Runciman

Department of Computer Science, University of York, UK
`{gem,colin}@cs.york.ac.uk`

Abstract. Benchmarking competing implementations of a data structure can be both tricky and time consuming. The efficiency of an implementation may depend critically on how it is used. This problem is compounded by persistence. All purely functional data structures are persistent. We present a kit that can generate benchmarks for a given data structure. A benchmark is made from a description of how it should use an implementation of the data structure. The kit will improve the speed, ease and power of the process of benchmarking functional data structures.

1 Motivation

Purely functional data structures differ from their imperative counterparts as the original value of a data structure may continue to exist after an update. Data structures that allow this are termed *persistent*. A sequence of operations that never uses this property is called *single-threaded*: after a data structure has been updated, its previous value is never reused.

When analysing the performance of competing implementations of an abstract datatype (ADT), it is common to choose a handful of applications, or benchmarks, and time each application using each implementation of the ADT. However, the choice of application can significantly affect the performance of each implementation. Moreover, the reasons why an implementation suits one application better than another is often not clear. Persistence compounds this problem: the performance of implementations can differ substantially between applications that take advantage of persistence to different degrees.

To illustrate this, consider the queue ADT whose signature is given in Fig.1. Listed below are three simple implementations of this ADT (code is given in Appendix A):

Naïve queues are implemented as ordinary lists. This provides $O(1)$ access to the front of the queue and $O(n)$ access to the rear, where n is the length of the queue.

Batched queues use a pair of lists to represent the front and rear of the queue [4]. This provides $O(1)$ amortized access to the front and rear of the queue, provided that the queue is used in a single-threaded manner, otherwise the complexity degenerates to $O(n)$. The name is taken from [6].

```

module Queue (Queue,empty,snoc,tail,head,isEmpty) where
empty :: Queue a
snoc  :: Queue a -> a -> Queue a
tail  :: Queue a -> Queue a
head  :: Queue a -> a
isEmpty :: Queue a -> Bool

```

Fig. 1. A signature for a queue ADT.

Banker's queues use the same implementation as batched queues but ensure that the rear list is never bigger than the front list [5]. This provides $O(1)$ amortized access to the front and rear of the queue regardless of the way in which the queue is used.

To see how these implementations behave under different uses of queues, the artificial simple applications shown in Fig.2 are run using each of the three implementations. The results are shown in Table 1.

These three applications were chosen to reveal how differently the queue implementations can behave according to how they are used. The huge values for **app2** using naïve queues and **app3** using batched queues are extreme examples of this. We might be able to guess at why this is so, especially where the implementations and applications are simple, as is the case here. However, this is by no means an easy task, and we would need to write and test several more applications to confirm any hypothesis.

To illustrate that these artificial applications are not unusual in how they distinguish between implementations, we also considered an application that implemented Shell's sort [9] using queues. Shell's sort depends on using insertion sort on successively larger sublists. The partitioning into sublists depends on a set of increments which may be adjusted to obtain greater efficiency. Using two different sets of increments and each implementation of queues, the same list of 1500 pseudo-random integers was sorted. The running times are given in Table 2. Code implementing Shell's sort using queues is given in Appendix A.

Again we see that a change in how the queue is used can change the efficiency of the queue drastically. Naïve queues perform well on the small set of increments, but very badly on the large set. But why? Given a knowledge of how this application uses the queues, one might guess that it is because the queues are much larger when using the larger set of increments. However, we cannot draw this conclusion from just looking at the times provided by Shell's sort.

To address this problem, we describe a kit named *Auburn* that generates applications which use an ADT in a specified manner. We may then measure the efficiency of competing implementations of the ADT when used by the generated applications. This will allow us to draw conclusions about the efficiency of an implementation of an ADT according to how it is used. For example, the results of this paper back up the suspicion that large naïve queues are rather inefficient.

```

-- 'apply n f q' applies 'f' to 'q' 'n' times,
-- ie. 'f (f .. (f q) ..)'
apply :: Int -> (a -> a) -> a -> a
apply 0 _ q = q
apply n f q = apply (n-1) f (f q)

-- 'snocTrue q' snoc's 'True' onto 'q'.
snocTrue :: Queue Bool -> Queue Bool
snocTrue q = snoc q True

-- 'app1 n' calculates the 'and' of 'n' _separate_ copies of:
-- '(head . tail . tail . snocTrue . snocTrue . snocTrue) empty'
app1 :: Int -> Bool
app1 n = (and . map getTrue . take n . repeat) ()
    where getTrue () =
        (head . apply 2 tail . apply 3 snocTrue) empty

-- 'app2 n' performs 'n' snoc's followed by 'n-1' tails,
-- finishing with a 'head'.
app2 :: Int -> Bool
app2 n = (head . apply (n-1) tail . apply n snocTrue) empty

-- 'app3 n' performs 'n' snoc's to give queue 'q'.
-- Then it performs tail then head on 'n' _shared_ copies of 'q'.
app3 :: Int -> Bool
app3 n = (and . map (head . tail) . take n . repeat) q
    where q = apply n snocTrue empty

```

Fig. 2. Three artificial simple applications of queues: `app1`, `app2` and `app3`.

Table 1. Average times in seconds over three runs of each application using each implementation of a queue. Standard deviation was less than 1% of the mean in every case.

Application	Naïve	Batched	Banker's
app1 1000000	4.66	4.84	4.95
app2 100000	16375.87	0.73	1.32
app3 100000	1.51	11032.00	1.53

Table 2. Average times in seconds over three runs of Shell's sort using each set of increments and each implementation of a queue on the same list of 1500 pseudo-random integers. Standard deviation was less than 1% of the mean in every case.

Increments	Naïve	Batched	Banker's
1, 3, 7, 21, 48, 112, 336, 861	12.89	0.53	0.58
1, 7, 48	0.55	0.40	0.41

Section 2 describes the underlying framework of the kit: labelled graphs called *dugs* that capture how an ADT is used by an application. Section 3 imposes restrictions on ADTs. Section 4 imposes restrictions on dugs. Section 5 describes how a dug may be condensed into a *profile*. Section 6 describes a method for avoiding ill-defined applications of randomly-chosen operations. Section 7 gives an example of the kit in action on the queue implementations of this section: naïve, batched, and banker’s. Section 8 concludes and outlines some future work. Appendix A gives code for the three implementations of queues.

2 Datatype Usage Graph

A major obstacle to overcome is the ambiguity of the phrase “how an ADT is used”. Without an exact definition of this property, we would find it hard to talk about the efficiency or otherwise of an implementation of an ADT according to how it is used, or indeed about how a particular application uses an ADT. Consider the three applications of queues in Fig.2. Inspecting the code for each application allows us to see what operations are being performed, in what order and how the result of one operation may rely on the result of another. But the task is by no means straightforward. With more complicated applications, the task would become extremely difficult. We need a simple record of how an ADT is used by an application.

We use a labelled directed graph. See Fig.3 for examples that describe how the queue ADT is used by the three applications of Fig.2. The nodes are labelled with partially applied operations of the ADT—each operation has all arguments that are *not* ADT values already supplied. There is an arc from u to v if the result of the operation at u is taken as an argument by the operation at v . If v has more than one incoming arc, these arcs must be labelled with the order in which they are taken by the operation at v . The nodes are numbered according to the order of evaluation. Such a graph is a *datatype usage graph*, or *dug*. Dugs are very similar to *execution traces* [6] and *version graphs* [2].

During the run of an application, many different instances of an ADT will exist. For example, whilst running queue application **app1** there will exist at some time an empty queue, a queue containing just **True**, a queue containing two copies of **True**, and so on. Each of these particular instances of the ADT is called a *version* [6]. A node of a dug is called a *version node* if it is labelled with an operation that results in a version. The subgraph of a dug containing just the version nodes is called the *version graph*. This is consistent with the definition of a version graph given by Driscoll et al. [2]. As each operation returns only a single value, we may associate each node with the value it produces. The nodes of the version graph can be thought of as representing different versions of the ADT formed by either generating a fresh version or by mutating one or more previous versions. In the latter case, the arcs specify which versions to give to the mutating operation and represent the flow of data *within* the privacy of the ADT framework. The nodes outside of the version graph and the arcs connecting them represent the flow of data *out* of the privacy of the ADT framework.

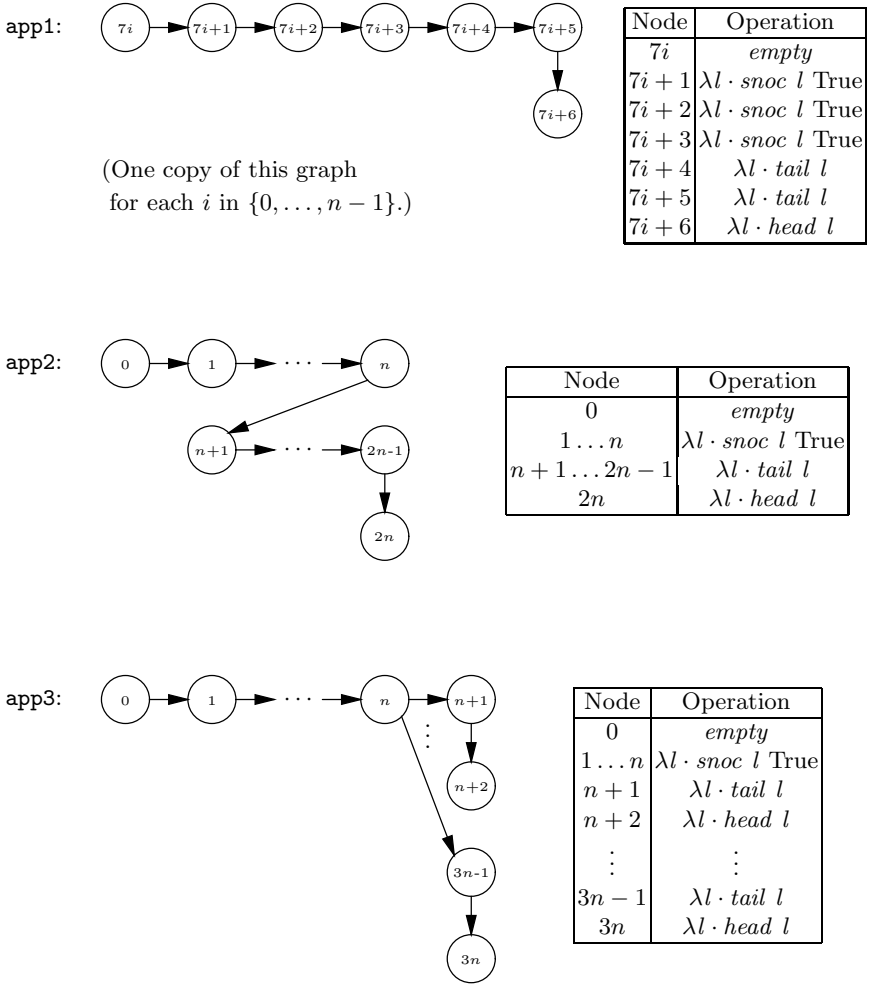


Fig. 3. Dugs showing how the queue ADT is used by the different applications given in Fig.2.

3 Simple Abstract Datatype

To simplify early releases of Auburn, we restrict the ADTs considered. A *simple* ADT satisfies the following:

- The ADT defines one type constructor T of arity one.
- The ADT only defines *simple operations*. An operation is simple if
 - its type contains at most one type variable, say a ,
 - every argument is of type $T a$, a or Int ,
 - the result is of type $T a$, a , Int or $Bool$, and
 - at least one argument or the result is of type $T a$.

Every simple operation may be classified as a

generator if no argument is of type $T\ a$, or as a
mutator if at least one argument is of type $T\ a$ and so is the result, or as an
observer if at least one argument is of type $T\ a$ but the result is not.

This classification is called the *role* of an operation. Any simple operation has a single role.

For example, the queue ADT whose signature is given in Fig.1 is simple and defines one generator: *empty*, two mutators: *snoc* and *tail*, and two observers: *head* and *isEmpty*.

4 Simple Dug

The following requirements ensure we have a well-defined dug:

- The number of incoming arcs to a node must match the arity of the operation it is labelled with (where the arity of an operation is the number of arguments of type $T\ a$).
- The operations must be type consistent. There must be at least one type to which we are able to instantiate the type variable “ a ” in every operation label.

As with the restrictions placed on ADTs, we shall impose the following restrictions on dugs to simplify early releases of Auburn:

- The dug must be defined over a simple ADT.
- The ordering given to the nodes must not conflict with the ordering given by the arcs. If node u is the predecessor of node v , u must be ordered before v . Note that this also implies that the dug must be acyclic.
- There must be no outgoing arcs from a node labelled with an observer operation, that is, no dependency must be made on the results of observations.

We discuss in Sect. 8 how these simplifications may be removed.

5 Dug Profile

A dug captures information relevant to how an ADT is used by an application. However, it is rather hard to talk about a large graph in detail, so we shall condense the most important information into a *profile*.

How often one operation is used relative to another is one of the most obvious candidates for a profile. We define the *weight* of an operation f to be the number of arcs leading to nodes labelled with f . For the purpose of this definition, consider all generators as operating on a unique void version.

Another important characteristic to capture is the degree of persistence. We split this into persistent mutations and persistent observations. We define a mutation/observation of a version v to be an arc from v to a mutator/observer

node. A persistent mutation/observation is any mutation/observation made on a version that has already been mutated (recall that the versions are ordered according to evaluation order). Therefore a persistent mutation is any mutation but the first. A mutation/observation that is not persistent is *ephemeral*.

It is useful to split a profile into *phases* covering different stages of an ADT's life. We define the *age* of a node to be the number of operations that have contributed to its creation. We define a *phase* to be a continuous age range. As the weights of generators are fixed at the start of an ADT's life, this property is separated out of a phased profile. Note that any given dug has many phased profiles, one for each choice of phases.

The following properties therefore make up a phased profile:

Generation Weight Ratio the ratio of the weights of each generator;

Phased Properties for a given phase P , the following four properties are recorded over the version nodes N of ages included by P :

Mutation/Observation Weight Ratio the ratio of the weights of mutators and observers, considering only operations on nodes in N ;

Mortality the fraction of nodes in N that are not mutated;

Persistent Mutation Factor (PMF) the fraction of mutations of nodes in N that are persistent;

Persistent Observation Factor (POF) the fraction of observations of nodes in N that are persistent;

For example, consider the dug shown in Fig. 3. As there is only one generator in the queue ADT, the generation weight ratio is redundant.

Consider the profile of **app1** over only one phase covering all ages. The mutation/observation weight ratio is: $snoc : tail : head = 3 : 2 : 1$. The mortality is $1/6$. The PMF and POF are both zero, indicating that the application is single-threaded.

Consider the profile of **app2** with respect to three phases: ages $\{1 \dots n\}$, $\{n + 1 \dots 2n - 1\}$ and $\{2n\}$. The age of node n is $n + 1$ so nodes $0 \dots n - 1$ are in the first phase, nodes $n \dots 2n - 2$ are in the second and node $2n - 1$ is in the third. The mutation/observation weight ratio $snoc : tail : head$ is $1 : 0 : 0$ in the first phase, $0 : 1 : 0$ in the second and $0 : 0 : 1$ in the third. Note that we count the application of an operation from the point of view of the node being operated on, not the node that results. Using the latter point of view would make phased profiles and persistence measures untenable at worst and messy at best. The mortality is $1/2n$ and the PMF and POF are both zero, again illustrating the single-threaded nature of the application.

Finally consider the profile of **app3** with respect to a single phase covering all ages. The mutation/observation weight ratio is: $snoc : tail : head = 1 : 1 : 1$. The mortality is $1/2$. The PMF is $(n - 1)/2n$ as all n applications of *snoc* are ephemeral, one application of *tail* is ephemeral and the remaining $n - 1$ applications of *tail* are persistent. The POF is zero as no observations of a node v occur after a mutation of v .

6 Guarding Data Structure

Recall that the purpose of the kit is to generate an application that uses an implementation of an ADT in a specified manner. We shall do this by first generating a dug with a given profile and then by running an evaluator over the operations of the dug. This evaluator will therefore use the ADT in a manner specified by the dug profile.

The generated dug only define operations on ADTs carrying elements of type *Int*. This simplifies the generation procedure (every argument to an operation is now either of type *Int* or *T Int*) and still allows the ordering or equality of elements to affect the efficiency of an implementation of an ADT. We discuss in Sect. 8 how this simplification may be removed.

Auburn generates a dug by making random choices that probabilistically result in a dug of the given profile. It builds a dug one node at a time. A major problem to overcome is the possibility of undefined results of partial operations. For example, the application *head empty* will not be defined in most specifications of a queue ADT and will probably result in an error if evaluated. We therefore need to guard against such applications. One way to do this is to provide *guarding operations*, or *guards*, that given an application will return true or false according to whether the application is well-defined or not.

For example, in the case of the *head* operation of queues, we may define an operation *head_Guard* by:

$$\begin{aligned} \text{head_Guard} &:: \text{Queue } a \rightarrow \text{Bool} \\ \text{head_Guard } q &= \text{not } (\text{isEmpty } q) \end{aligned}$$

which defines the application *head q* to be well-defined if and only if *q* is not empty. However, this requires a dug generator to have access to an implementation of the ADT. It is simpler and more efficient to maintain a *shadow* of every ADT version. The shadow of a version *v* will contain all of the information required to determine if an application of an operation involving *v* is well-defined. We will shadow queues simply by their length. The definition of *head_Guard* becomes:

$$\begin{aligned} \text{head_Guard} &:: \text{Shadow} \rightarrow \text{Bool} \\ \text{head_Guard } s &= (s > 0) \end{aligned}$$

where the type *Shadow* is defined to be *Int*.

The shadow of a data structure is maintained by providing a shadow operation for every generator and mutator (observers do not produce ADT versions). Given an operation of type *t*, the shadow operation is of type *shadow(t)*:

$$\begin{aligned} \text{shadow}(t_1 \rightarrow t_2) &= \text{shadow}(t_1) \rightarrow \text{shadow}(t_2) \\ \text{shadow}(T \text{ Int}) &= \text{Shadow} \\ \text{shadow}(\text{Int}) &= \text{Int} \end{aligned}$$

Shadow operations that shadow a queue by its length are given in Fig.4.

This approach has a drawback: every argument of an operation must be chosen *before* passing these arguments to the relevant guard. However, with

```

type Shadow = Int

empty_Shadow :: Shadow
empty_Shadow = 0
snoc_Shadow :: Shadow -> Int -> Shadow
snoc_Shadow s _ = s+1
tail_Shadow :: Shadow -> Shadow
tail_Shadow s = s-1

```

Fig. 4. Haskell code defining shadow operations for queues.

an application such as *lookup l i* on lists which returns the i^{th} element of l , this means guessing which indices are available for *lookup* before testing the validity of the application. A better scheme passes the guard *only the version arguments* of an operation. The valid ranges of remaining arguments are returned as the result. One argument is then chosen from each range with the resulting application guaranteed to be valid. As we have ensured that every argument to an operation that is not a version is of type *Int*, a guard may return a range using the type *IntSubset*:

$$\begin{aligned}
 \text{data IntSubset} = & \text{All} \\
 & | \text{Int} \dots \text{Int} \\
 & | \text{FiniteSet (Set Int)} \\
 & | \text{None}
 \end{aligned}$$

Given an operation of type t taking n arguments, v of which are versions, the type of its guard is given by $\text{guard}(v, n - v)$:

$$\text{guard}(v, i) = \begin{cases} \text{Shadow} \rightarrow \text{guard}(v - 1, i) & \text{if } v > 0 \\ [\text{IntSubset}]_i & \text{if } v = 0 \wedge i > 0 \\ \text{Bool} & \text{if } v = 0 \wedge i = 0 \end{cases}$$

where $[a]_i$ is the type of lists of i elements of type a .¹

For example, shadowing lists by their length, a suitable guard of *lookup* on lists could be given by:

$$\begin{aligned}
 \text{lookup_Guard} & \quad :: \text{Shadow} \rightarrow [\text{IntSubset}]_1 \\
 \text{lookup_Guard } s & = [0 \dots (s - 1)]
 \end{aligned}$$

which defines an application *lookup l i* to be well-defined if and only if i is greater than or equal to zero and less than the length of l . Guards of queue operations are given in Fig.5. Note that Haskell does not provide a type for lists of given length, and so general lists must be used.

A suitable definition of a shadow type, shadow operations and guard operations for an ADT \mathcal{A} is called a *guarding data structure* of \mathcal{A} . Note that the

¹ Haskell does not support functions over tuples of arbitrary size so we are forced to use lists, here annotated informally with their intended length.

```

empty_Guard :: Bool
empty_Guard = True
snoc_Guard :: Shadow -> [IntSubset]
snoc_Guard s = [All]
tail_Guard :: Shadow -> Bool
tail_Guard s = (s>0)
head_Guard :: Shadow -> Bool
head_Guard s = (s>0)
isEmpty_Guard :: Shadow -> Bool
isEmpty_Guard s = True

```

Fig. 5. Haskell code defining guards of queue operations.

guarding data structure is only used to aid the generation of dugs and is not involved in the evaluation of dugs or any other use of an implementation of the ADT.

7 An Example: Benchmarking Queues

There are five ingredients in any experiment using Auburn:

1. a signature of an ADT,
2. implementations of the ADT,
3. a guarding data structure,
4. a dug generator, and
5. dug evaluators, one for each ADT implementation.

The user must supply 1 and 2. Auburn provides a template for 3 which the user must complete, and provides 4 and 5 in full. A signature of an ADT is simply a Haskell module exporting the type constructor and operations of the ADT *with* type signatures but *without* code.

We illustrate use of the kit on the queue implementations given in Sect.1: naïve, batched and banker's. The signature of queues that we shall use for our experiment is given in Fig.1. The module is named `Queue` and is stored in the file `Queue.sig`. We start our experiment with the signature file and the three implementations:

```

$ ls
BankersQueue.hs BatchedQueue.hs NaiveQueue.hs Queue.sig

```

7.1 Making the Dug Generator and the Dug Evaluators

Given a signature, Auburn produces type signatures of the required operations for a guarding data structure:

```
$ auburn -d Queue
Wrote 'Queue_GDS.hs'.
```

Auburn outputs these in a module called `Queue_GDS` based on the name of the signature module. The user must then add definitions of the required operations. Fig.4 and Fig.5 together define the guarding data structure of queues that we shall use.

Given a signature, Auburn creates a generator of dugs:

```
$ auburn -g Queue
Wrote 'Queue_Gen.hs'.
```

and one evaluator of dugs for each implementation to be tested:

```
$ auburn -e Queue NaiveQueue BatchedQueue BankersQueue
Wrote 'Queue_Eval_NaiveQueue.hs'.
Wrote 'Queue_Eval_BatchedQueue.hs'.
Wrote 'Queue_Eval_BankersQueue.hs'.
```

As we have three implementations of queues, we now have one generator of dugs and three evaluators of dugs. A dug is stored in a file as a sequence of operations. The dug generator produces such a file given a profile and a few other controlling parameters (such as a random seed). A dug evaluator reads and performs the operations listed, evaluating the dug using a particular implementation of the ADT. The evaluator returns a *checksum* made from the sum of the observations. This checksum may be used to check the correctness of a new implementation of an ADT by comparing it with the checksum produced by a trusted implementation.

Auburn can also produce a *null implementation* of the ADT:

```
$ auburn -n Queue
Wrote 'Queue_Null.hs'.
```

This performs almost no work and produces incorrect results, but it does provide a type constructor and operations of the required type. The purpose of the null implementation is to allow an evaluator to be built:

```
$ auburn -e Queue Queue_Null
Wrote 'Queue_Eval_Queue_Null.hs'.
```

which will spend almost all of its time doing the work required for reading a dug file such as input/output, summing the checksum, etc. Subtracting this time from the time spent by another evaluator yields an estimate of the actual time spent performing ADT operations.

7.2 Choosing the Profiles

For our experiment, we shall build dugs with two phases: a build phase and a fall phase. The phases shall be of equal length and provide a convenient framework for adjusting the following parameters:

1. the mean size of the queues involved in operations,
2. the speed of construction/de-construction, and
3. the degree of persistence used.

By varying each parameter across two values, a simple Haskell script produces eight profiles:

```
$ makeProfiles
```

in the files `dug- n .profile` for $1 \leq n \leq 8$, and a description of each profile in the file `dugs.profiles`. For example, setting the ratio of *snoc* : *tail* to 10 : 1 in a build phase lasting 122 operations, and to 1 : 10 in a fall phase lasting 122 operations, produces dugs with a mean queue size of 50 and a speed of construction/de-construction of 9/11. Together with zero persistence, this profile is stored in the file `dug-7.profile`.

Parameter 1 was chosen to indicate the expected problem naïve queues have in handling large queues. Parameter 2 is an attempt to capture the difference between the batched and banker’s implementations. The batched implementation delays work that may be *repeated* later by persistent operations. This work may be forced by an application of *tail*. This is the root of the problem when the batched implementation performs poorly on application `app3` of Sect. 1. The banker’s implementation avoids this by periodically packaging delayed work to be *shared* later. Therefore, building and taking apart a queue quickly with persistence may disadvantage the batched implementation. Parameter 3 is of interest primarily because little is known about the effect of persistence on the efficiency of data structures.

7.3 Generating the Dugs

As Auburn builds a dug, there is a part of the graph that is “live” and waiting to be attached to new nodes, and a part that is “dead” that will play no further role in the generation of the dug. The live section is called the *frontier* of the graph. The size of the frontier grows exponentially in the degree of persistence. To counter this growth a maximum bound may be placed on the size of the frontier—this is an additional argument to the dug generator. For ADTs supporting non-unity mutators (operations that combine versions, for example catenation) it is also useful to be able to set the minimum size of the frontier. For our experiment, we set the frontier to have minimum size one and maximum size ten.

In all, a dug generator takes seven arguments:

1. the profile to aim for—the `makeProfiles` program will generate the profiles for our experiment;
2. the size of the pool to draw integer arguments from—this is not relevant in our experiment as queues do not look at the elements they carry, so we set it to ten to allow the checksum to vary reasonably;
3. the number of nodes in the dug—our experiment generates dugs of ten thousand nodes;

4. the minimum size of the frontier—set to one for our experiment;
5. the maximum size of the frontier—set to ten for our experiment;
6. a seed for pseudo-random number generation—for each profile, we shall generate five dugs from five different seeds; this allows variation of the dug within the given profile;
7. the mode to run in—to generate the dug file, **Compile** mode is used, and to analyse the actual profile of the resulting dug we use **Analyse** mode; other modes include **Draw** for producing output from which **dotty** [3] may draw the dug, and **Raw** for producing a raw text description of the dug.

Auburn is distributed with three simple Perl scripts that partially automate any benchmarking experiment. The first of these scripts generates dug files from the profiles already generated:

```
$ makeDugs
```

in files of the form **dug-*n-seed*.dc**. For example, the profile stored in **dug-7.profile** given above causes a dug to be built as follows:

```
Queue_Gen "Profile [1.0] [Phase 122 [1.81818, 0.181818, 2.0, 0.0]
          0.0 0.0 0.0, Phase 122 [0.181818, 1.81818, 2.0, 0.0]
          0.0 0.0 0.0]" 10 10000 1 10 1899102810 Compile
>dug-7-1899102810.dc
```

and stored in the file **dug-7-1899102810.dc**.

The **Analyse** mode returns the actual profiles of the dugs generated. These were stored in files of the form **dug-*n-seed*.profile** and checked for being close to the profiles we aimed for. The mean size of a queue can be displayed if we supply additional operations in the module exporting the guarding data structure. These maintain information on the shadows of ADT versions passed to operations. As the shadow of a queue is its size, we simply sum the shadows and then calculate the mean.

7.4 Evaluating the Dugs

A dug evaluator takes two arguments:

1. the dug file, and
2. the number of times to evaluate the dug.

The latter argument is used when the evaluation time is too small and increasing the size of the dug file is impractical. Our experiment repeats each evaluation ten times. The call to the evaluator is also repeated three times to gain a better estimate of the evaluation time.

The second Perl script distributed with Auburn calls the dug evaluators:

```
$ evalDugs
```

and records the total times in files of the form `dug-n-implementation.time`. These times in seconds are given in Table 3. Here is how the dug stored in `dug-7-1899102810.dc` is evaluated using banker's queues:

```
Queue_Eval_BankersQueue dug-7-1899102810.dc 10
```

The third Perl script subtracts the null implementation times, and calculates the ratios of the resulting times:

```
$ processTimes
```

and stores the result in the file `dugs.times` using the descriptions of each profile given in `dugs.profiles`. These ratios are given in Table 4.

7.5 Analysis

The first thing to note is that the batched implementation is the clear winner in all categories considered by this experiment (see Table 4). The absolute times in Table 3 also show little variation according to the profile used. A large (de)construction speed has none of the intended ill effect on its performance. This is surprising. The following profile:

```
Profile [1.0] [Phase 50 [2.0, 0.0, 2.0, 0.0] 0.0 0.0 0.0,
                  Phase 1 [0.0, 2.0, 0.0, 0.0] 0.0 0.9 0.9,
                  Phase 1 [0.0, 0.0, 1.0, 0.0] 1.0 0.0 0.0]
```

simulates the key elements of the application `app3` of Sect.1 that caused the batched implementation so much trouble. Evaluating dug with this profile does indeed produce poor results for the batched implementation: it takes twice as long as the other two implementations. However, we were not able to find a less obscure profile that also produced poor performance for the batched implementation. This suggests that the batched implementation only performs poorly on a rather small isolated group of dug.

The banker's implementation also performs evenly across all profiles considered, but lags behind the simple pair implementation by just over a factor of two. The banker's implementation performs much of the same work as the batched implementation but uses more bookkeeping to ensure a constant complexity in all circumstances. In all profiles considered here, this extra work is not necessary.

The naïve implementation performs very poorly on the large queues, and reasonably well on the small queues, as expected. One notable feature is the increase in efficiency in moving from ephemeral to persistent use. This is true to some extent of all the implementations across all profiles. This can be explained by considering that the same number of operations will share more work with persistence than without. The degree of change is still surprising however.

It seems from this experiment that the batched implementation of queues performs the best amongst those considered in most circumstances. The banker's implementation only wins when queues are used in a particular, rather uncommon manner. It is worth replacing the naïve implementation with the batched implementation, even for reasonably small queues.

Table 3. Total times in seconds of the evaluation of dugs of each profile using each implementation. These were obtained with the York release of the nhc compiler [10] using C interface extensions to Haskell to improve the accuracy of the results.

Profile Parameters			Implementation			
Mean Size	(De-)construction Speed	Persistence	Null	Naïve	Batched	Banker's
5	1/3	0	14.480	28.870	22.690	28.900
5	1/3	0.1	14.580	25.450	22.510	26.600
5	9/11	0	14.400	26.170	22.270	27.780
5	9/11	0.1	14.890	22.580	21.460	25.790
50	1/3	0	14.610	116.670	22.480	29.260
50	1/3	0.1	14.670	42.620	21.940	24.900
50	9/11	0	15.070	111.340	22.800	28.220
50	9/11	0.1	15.330	41.340	21.130	24.470

Table 4. Ratios of the total times given in Table 3 after subtracting the null implementation times.

Profile Parameters			Implementation		
Mean Size	(De-)construction Speed	Persistence	Naïve	Batched	Banker's
5	1/3	0	1.753	1.000	1.756
5	1/3	0.1	1.371	1.000	1.516
5	9/11	0	1.496	1.000	1.700
5	9/11	0.1	1.170	1.000	1.659
50	1/3	0	12.968	1.000	1.861
50	1/3	0.1	3.845	1.000	1.407
50	9/11	0	12.454	1.000	1.701
50	9/11	0.1	4.484	1.000	1.576

8 Conclusion, Present Status, and Future Work

Auburn provides a convenient framework on which to build benchmarking experiments. Little is known about the empirical performance of functional data structures, but the kit should help us address this issue. Compared to building benchmarks by hand, the kit is very quick and easy to use. Compared to choosing a fixed set of benchmarks whose use of an ADT may be unclear, the kit allows a wide range of uses of an ADT to be defined easily and clearly. The kit may also be used to test the correctness of a new implementation of an ADT by comparing its results against a trusted implementation (see the discussion of checksums in Sect. 7.1). Auburn should enable functional programmers to make a more informed choice of efficient data structures.

The current version of Auburn may be downloaded from the Auburn web page [1]. Note that this paper is based on Auburn version 1.0, which is also

available for downloading at the Auburn web page. A tutorial based on the example of Sect. 7 is included in any version of Auburn, and has been adapted to suit that version.

Listed below are the main areas for improvement.

Dug Extraction. A major addition to the kit would be a *dug extractor*. Given an application that uses an ADT, the extractor runs the application and produces a dug of how it uses the ADT. This allows a user to match their application to the most suitable implementation of an ADT.

Current work on Auburn version 2.0 provides an extractor via the Green Card language extension [8]. The extractor can also handle unevaluated portions of a dug, something not discussed in this paper but present in many applications. However, the extractor cannot currently record any non-version arguments.

Generalisation. The kit only handles simple ADTs covering most but not all operations. For example, higher-order operations such as *fold* and *map* are not simple. Operations involving more than one data structure such as *toList* and *fromList* are also not simple. Such operations must be excluded from the ADT signature.

To include higher-order operations would involve choosing a random function as an argument whilst generating a dug. It is not clear how this is best done, though one possibility might be to let the user supply a collection of functions from which to choose one at random.

To include operations over lists as well as over the ADT, we would also have to choose a random list. This is more straightforward than choosing a random function, but would still require a reasonable amount of input from the user; for example, how long should the list be?

Generating dugs with arguments other than versions or integers is problematic for similar reasons: how do we choose one at random?

Extending the dug model to include dependencies on observations is very tedious for dug generation, and very difficult for dug extraction. The value of observation dependencies does not appear to warrant this effort.

Evaluation order is a very confusing and difficult issue to deal with properly. In lazy functional languages, one cannot tell in general from a simple trace of events if the underlying algorithm is non-single-threaded or single-threaded, as lazy evaluation may re-order a single-threaded computation into an apparently non-single-threaded route. However, if any persistent mutation is observed, the algorithm must be non-single-threaded.

Automation. Much of the Auburn user's work could be automated, especially the running of dug evaluators and the collection and processing of times. Automating scripts and makefiles are distributed with the kit, but more could be done. Although it is reasonably straightforward to give the code for a guarding data structure, help in the form of suggestions or guidelines would be useful. Default shadows and guards for total operations would be useful too.

Current work on Auburn version 2.0 provides default shadows and guards for two standard shadow data structures: one that does nothing but provides a convenient template, and one based on the size of the ADT versions.

Acknowledgements

Thanks go to Martyn Pearce for his many comments on a draft of this paper, and to Nick Merriam for his implementation of Shell's sort.

References

- [1] The Auburn Home Page. <http://www.cs.york.ac.uk/~gem/auburn/>.
- [2] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.
- [3] Graphviz: Tools for viewing and interacting with graph diagrams. <http://www.research.att.com/sw/tools/graphviz/>.
- [4] Robert Hood and Robert Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13(2):50–54, November 1981.
- [5] Chris Okasaki. Simple and efficient purely functional queues and dequeues. *Journal of Functional Programming*, 5(4):583–592, October 1995.
- [6] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, September 1996.
- [7] Chris Okasaki. The role of lazy evaluation in amortized data structures. In *Proceedings of the International Conference on Functional Programming*, pages 62–72. ACM Press, May 1996.
- [8] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. Green Card: A foreign-language interface for Haskell. In *Haskell Workshop*, Amsterdam, June 1997. Published by Oregon Graduate Institute of Science & Technology.
- [9] D. L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959.
- [10] York Functional Programming Group. <http://www.cs.york.ac.uk/fp/>.

A Code

Figures 6, 7 and 8 give Haskell code for the naïve, batched and bankers queue implementations respectively. Figure 9 gives Haskell code implementing Shell's sort using queues.

```

newtype Queue a = Queue [a]
empty = Queue []
snoc (Queue xs) x = Queue (xs ++ [x])
tail (Queue (_:xs)) = Queue xs
head (Queue (x:_)) = x
isEmpty (Queue []) = True
isEmpty _ = False

```

Fig. 6. Haskell code implementing naïve queues. Naïve queues use ordinary lists.

```

data Queue a = Queue [a] [a]
empty = Queue [] []
snoc (Queue f r) x = queue f (x:r)
tail (Queue (_:f) r) = queue f r
head (Queue (x:_)) = x
isEmpty (Queue [] _) = True
isEmpty _ = False
queue :: [a] -> [a] -> Queue a
queue [] r = Queue (reverse r) []
queue f r = Queue f r

```

Fig. 7. Haskell code implementing batched queues. Batched queues are based on an implementation due to Hood and Melville [4]. A queue is represented by a pair of lists (f, r) — f giving the front portion of the queue and r the rear portion of the queue but *reversed*. A batched queue (f, r) may be converted into a naïve queue by applying: *append f (reverse r)*. When the front list becomes empty we make a new front list from the reversal of the rear list. The new rear list is empty.

```

data Queue a = Queue !Int [a] !Int [a]
empty = Queue 0 [] 0 []
snoc (Queue lenf f lenr r) x = queue lenf f (lenr+1) (x:r)
tail (Queue lenf (_:f) lenr r) = queue (lenf-1) f lenr r
head (Queue _ (x:_)) = x
isEmpty (Queue _ [] _ _) = True
isEmpty _ = False
queue :: Int -> [a] -> Int -> [a] -> Queue a
queue lenf f lenr r
  | lenr <= lenf = Queue lenf f lenr r
  | otherwise    = Queue (lenf+lenr) (f++reverse r) 0 []

```

Fig. 8. Haskell code implementing banker’s queues. Banker’s queues are given by Okasaki in [5] and shown to have $O(1)$ amortized bounds in [7] and [6]. The implementation is similar to the batched queues except that the rear list is never allowed to grow larger than (a constant times) the front list—we use a constant of one here. When this invariant is about to be violated, we append the reversal of the rear list onto the back of the front list.

```

-- 'fromList' and 'mapQ' would probably be more efficient if
-- imported from an implementation (but Auburn does not support
-- these operations).
fromList :: [a] -> Queue a
fromList xs = foldl snoc empty xs
mapQ :: (a -> b) -> Queue a -> Queue b
mapQ = mapQ' empty
mapQ' q' f q | isEmpty q = q'
              | otherwise = mapQ' (snoc q' (f (head q))) f (tail q)

-- Insertion sort.
isort :: (a -> a -> Bool) -> [a] -> [a]
isort before xs = foldr insert [] xs
  where insert x [] = [x]
        insert x zs@(y:ys) | x 'before' y = x : zs
                           | otherwise    = y : insert x ys

-- Merges from a queue of m lists to a queue of n lists.
mergeToN :: Int -> Queue [a] -> Queue [a]
mergeToN n ys = mergeToN' ys (fromList (take n (repeat [])))
mergeToN' ins outs | isEmpty ins = outs
                   | otherwise    =
  case head ins of
    [] -> mergeToN' (tail ins) outs
    (y:ys) -> mergeToN' (snoc (tail ins) ys)
                     (snoc (tail outs) (y : head outs))

-- We would normally generate the increments from the length of the
-- list to be sorted (all increments should be less than this length).
-- However, we know that the list to be sorted has length 1500.
incs :: Int -> Int -> [Int]
incs 0 len = [1, 3, 7, 21, 48, 112, 336, 861]
incs 1 len = [1, 7, 48]

-- As we merge from one queue of lists to another, each list is
-- reversed, and so the ordering we sort with must also be reversed.
incsOrders :: Ord b => Int -> Int -> [(Int, b -> b -> Bool)]
incsOrders incsSel len = zip (incs incsSel len) (cycle [(<=), (>=)])

-- Shell's sort is an alternating sequence of merges and
-- insertion sorts piped together.
shellSort :: Ord a => Int -> [a] -> [a]
shellSort incsSel [] = []
shellSort incsSel xs =
  head (foldr1 (.) (map makePipe (incsOrders incsSel (length xs)))
        (snoc empty xs))
  where makePipe (inc,order) = mapQ (isort order) . mergeToN inc

```

Fig. 9. An implementation of Shell's sort using queues.

Complete and Partial Redex Trails of Functional Computations

Jan Sparud and Colin Runciman

Department of Computer Science, University of York, UK
(e-mail: {sparud,colin}@cs.york.ac.uk)

Abstract. Redex trails are histories of functional computations by graph reduction; their main application is fault-tracing. A prototype implementation of a tracer based on redex trails [8] demonstrated the promise of the technique, but was limited in two respects: (1) trails did not record every reduction, only those constructing a new value; (2) even so computing trails was very expensive, particularly in terms of the memory space they occupied. In this paper, we address both problems: *complete* redex trails provide a full computational record; *partial* versions of these trails exclude all but selected details, greatly reducing memory costs. We include results of experiments tracing several applications, including a compiler.

1 Introduction

Programs do not always work first time. When they fail, programmers cannot always see the reason immediately. Implementors must therefore provide some way of *tracing* computations. By examining traces, programmers can see not only final results or failures, but also how these have come about.

In many programming languages, traces can be based on sequences of events. But in a lazy functional language, the sequence of events is not so easily presented, nor is it what the programmer really needs. A more appropriate form of trace links a final value, or failure, to its *parent redex*, and each part of that redex in turn to *its* parent, and so on until one reaches input data or initial expressions present in the program. We call such traces *redex trails*. In a previous paper [8] we described a method for transforming functional programs to self-tracing equivalents, computing not only the result as usual but also a redex trail showing how the result was obtained. In addition, we described a *trace browser* used to examine traces in close connection with programs and their results. The scheme was promising, but its scope of usefulness was limited in two respects.

First, not every reduction was recorded in a redex trail, only those with the construction of a new value as the outcome. For some applications we found this was enough to provide highly informative traces. But in others, for example those with a strong search component, large parts of the computation are concerned with conditions for selecting between given values; if details of these parts are not recorded in trails, the programmer examining traces lacks important information.

Secondly, and despite the restricted class of reductions recorded, it was very expensive to compute redex trails. For a range of small applications we found that a trace-constructing computation could take 30 times longer than a normal one. The demands for memory space were even more prohibitive, with each recorded reduction taking up to 100 bytes. Tracing was just not feasible for computations beyond about a million reductions — sufficient for most textbook exercises, but not for serious applications.

In this paper we describe how we have tackled both problems. Redex trails have been extended to include a representation for every reduction that occurs, providing complete information about all parts of a computation. Desirable as this may be, it makes trails even larger, and more expensive. We have implemented two techniques for controlling the construction and retention of redex trails, so that stored trace structures are simpler and smaller: it is indeed useful to confine trails to a limited class of reductions, but now this class is *selected* by the programmer in terms of the program components they wish to observed at work; if demands on memory are still too high, we *prune* redex trails using harsh garbage-collection policies.

We give results for a range of applications, including two much larger than those studied in the previous paper.

Section 2 briefly explains the main elements of redex trails, as implemented in our prototype tracing system. Section 3 identifies the class of reductions not recorded by that system, and describes the extension to *complete* redex trails. Section 4 is concerned with *partial trails*; it deals first with *selective* trails and then with the complementary technique of trail *pruning*. Section 5 gives results, concentrating on the space requirements of the enhanced tracing system. Section 6 discusses related work. Section 7 draws conclusions and suggests future work.

2 A Redex-Trail Tracing System

Suppose the function `length` is defined by the equations

```
length [] = 0
length (x:xs) = 1 + length xs
```

and we compute 3 as the result of `length [1,2,3]`. Figure 1 shows a trace of that computation in the form of a *redex trail*. The *parent redex* of the final 3 is 1+2. Considering subexpressions in that redex, everything but the 2 was constructed as an immediate result of reducing the original expression `length [1,2,3]`, but the 2 came about by reduction of 1+1. And so on.

As described in a previous paper [8], we have modified the Haskell compiler `nhc` [5, 6] to provide traces based on redex trails. On request, our modified compiler applies a program transformation causing each value in the compiled program to be computed along with a redex trail recording the value's origin. A special browser, illustrated in Figure 2, allows the user to examine redex trails, starting from program output, or a run-time fault, or some point at which evaluation was interrupted.

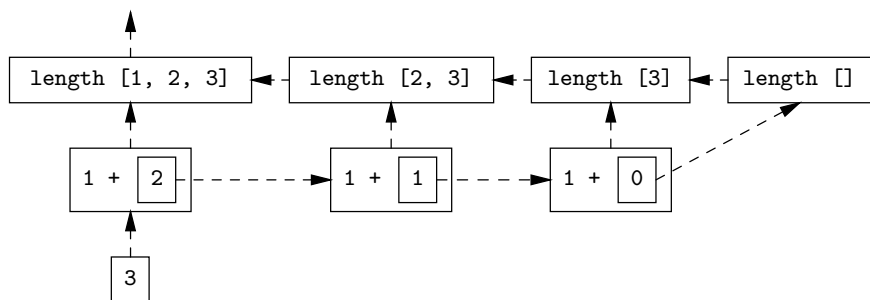


Fig. 1. The full redex trail for the computation of 3 as the result of `length [1,2,3]`.

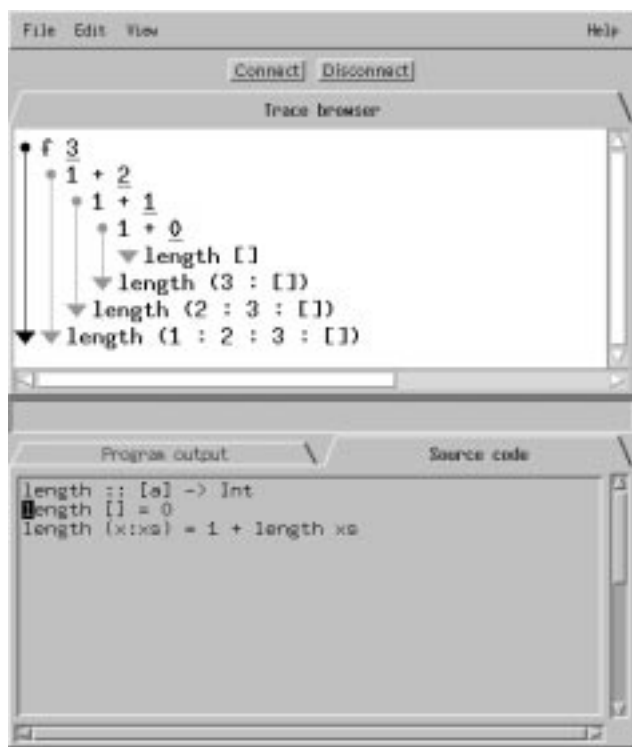


Fig. 2. How redex trails are displayed by the trace browser, shown here being used to examine the `length` trail of Figure 1. Each redex on display has been requested by the user in a point-and-click style. Available links to the source program include the points of definition and use for each variable occurring in the trail, and also the site of each function application.

Redex trails are themselves implemented as Haskell data structures of type `Trace`. A simplified definition of the `Trace` datatype is:

```
data Trace = Appl Trace [Trace] SourceRef
           | Name Trace NameInfo SourceRef
           | Root
```

Each construction `Appl t_c [$t_f, t_{a_1}, \dots, t_{a_n}$] sr` records a function application. The trail recording the origin of the function is t_f , and the trails t_{a_i} are for its arguments. The trail t_c , which we term the *application context* records how the application of this function to these arguments came about. (In a first-order language t_c and t_f would always be the same, but in a higher-order language it is important to distinguish between them.) The source reference sr allows immediate access to the corresponding function application in the source program.

Each construction `Name t i sr` represents an identifier, constant or literal. Its redex trail is t . The information field i records a textual representation of the name, and the module and position of its *definition*. The source reference sr is linked to the relevant applied occurrence or *use* of the name.

The `Root` constructor is used as a trail terminator: for example, as the trail of each name with a top-level definition.

As all values in a traced computation, including functions, have trails attached to them, traced function applications are translated by the compiler to applications of special combinators that extend the redex trails appropriately.

3 Complete Trails

In the previous section we had a good reason for choosing the `length` function as an example: our prototype system builds *complete* redex trails for `length` computations, including a record of every reduction. But now consider another simple function over lists:

```
last [x] = x
last (x:xs) = last xs
```

Suppose we again compute 3, but this time as the result of `last [1,2,3]`. In contrast to Figure 1, the trail constructed for this computation by our prototype tracer is not very informative (see Figure 3).



Fig. 3. A rather uninformative redex trail for the result of `last [1,2,3]`.

A programmer examining this trail to answer the question ‘Where does the result 3 come from?’ finds only the answer that 3 was given in the original

expression; there is no mention of the `last` function or its workings. Why such a dramatic difference between the trails for `length` and `last` computations? Because a trail attached to a value by our prototype system records only how that value itself came to be computed. Shared values have identical trails. If the result of a function is (a component of) one of its arguments the trail for the result is exactly the trail already recorded in the argument.

Results that are not fresh values also pose a problem for graph-reducing interpreters or compilers: the application subgraph must somehow be updated to become, in effect, a piece of graph already constructed elsewhere. A commonly adopted solution is to introduce an indirection. We also use indirections to record such reductions in extended redex trails, giving `Trace` a new construction:.

```
data Trace = ... | Indi Trace Trace
```

A trace of the form `Indi t_{res} t_{app}` is constructed for the result of an indirecting application trace t_{app} , the result itself having trace t_{res} . The revised trail for the `last [1,2,3]` example is shown in Figure 4.

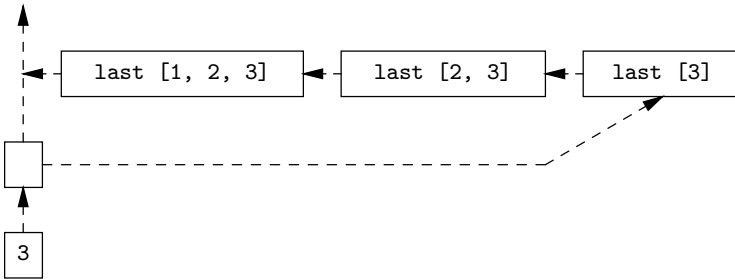


Fig. 4. A complete redex trail for `last [1,2,3]`. The empty box represents an indirection giving access to the chain of recursive calls selecting 3 as the final list element as well as to the trace of 3 itself.

With the addition of these indirection structures, redex trails become complete. Every reduction occurring in the computation can be observed in the trail.

4 Partial Trails

Our prototype tracer needed tens of megabytes to store the trails of computations involving a few hundred thousand reductions. Plenty of functional programs take many millions of reductions to compute their results. And with redex trails extended to include all reductions, not just the value-producing ones, the demand for memory will be even greater. There is a clear need for techniques to obtain simplified, smaller variants of a full redex trail.

4.1 Selective Trails

Not even the most inquiring programmer will be able (or willing!) to look at every single reduction in a large computation. So why record all of them? The problem is how to know which reductions to record and which to ignore. One solution is to let the programmer select the parts to be traced when they run a program.

Suppose each function defined at the top level of a program is either *suspected* or *trusted* by the programmer, and somehow marked as such. (Even in a freshly written program, obvious candidates for trusted functions include any from the **Prelude**.) Broadly speaking, an application of a suspected function should always be recorded in the redex trail, but for trusted functions this may be unnecessary.

For example, when taking the length of a list, we are rarely interested in how **Prelude.length** works, so recording a full redex trail for the internal computation of **length** would be excessive, both from the programmer's point of view and in terms of memory space. If we trust **length** (and primitive addition), then instead of the full redex trail in Figure 1 we might prefer the simpler *partial trail* shown in Figure 5. Note that the top-level call to **length** is still in the

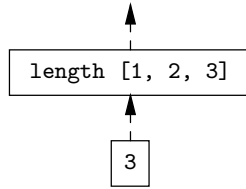


Fig. 5. A selective trail for **length** [1, 2, 3] when **length** is trusted.

trace. Though we trust the internal workings of **length**, we still want to see the *original application* that reduced to 3.

The tracing machinery has to decide for each function application whether or not to record it in the redex trail. A simple rule would be: *Do not record applications in a trusted context*. This would produce the desired result for the **length** function, as all the intermediate applications of **length** and **+** occur in the context of **length** itself.

Higher-Order Functions However, matters get more complicated when we decide to trust a higher-order function. Figure 6 shows a full redex trail from the result of **map** **dbl** [1, 2, 3], where **dbl** is a function that doubles its argument. Suppose we trust **map** but not **dbl**. Should we then trust the result of **map** **dbl** [1, 2, 3]? Clearly not. The problem is that the trusted function **map** builds applications of the suspected function **dbl**. If we follow our simple rule, we will not record applications of the suspected function **dbl**, as figure 7 shows.

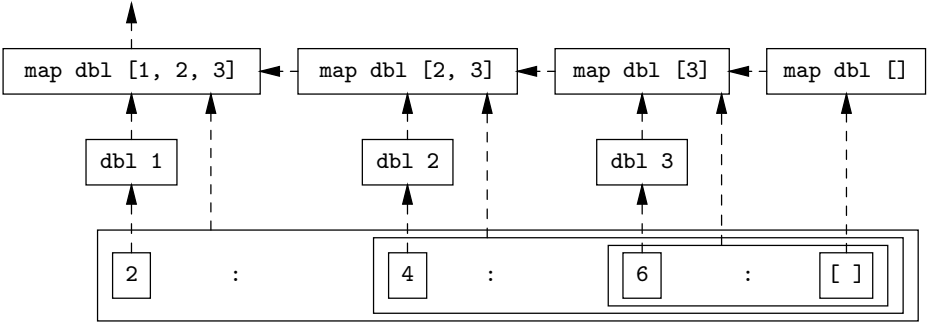


Fig. 6. The complete trail for `map dbl [1, 2, 3]`.

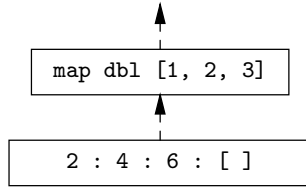


Fig. 7. A selective trail for `map dbl [1, 2, 3]` when `map` is trusted but `dbl` is suspected, if context alone determines which reductions are recorded.

To solve this problem we revise the rule: *Do not record applications of trusted functions in trusted contexts*. Figure 8 shows the effect of this revised rule for the `map dbl` example: there is no record of applications within `map`, except for the calls to `dbl`.

Implementing Selective Trails To implement selective trails, we need to be able to find out if the context and the applied function are trusted when a function is applied. We can find out the name of the function in the current context and the name of the function to be applied by inspecting their accompanying trace structures. But as this operation will be performed every time a function is applied, it is important to make it cheap. The combinators for traced function application are written in Haskell, but to use a Haskell data structure to record whether a function is trusted or not is inappropriate for two reasons: (1) it would be hard to create the structure in the context of separate compilation; (2) it would be expensive to lookup the information in it.

Recall that wherever a name occurs in a redex trail, there is a `Name` construction in the `Trace`, including a record of all needed information about the name. The record for each name is unique, shared across all `Name` occurrences. For each module, the compiler constructs a table of records for all the names the module defines. Also, for each module there is a table of imported modules; so from the main module it is possible to reach the name table for all modules in the program.

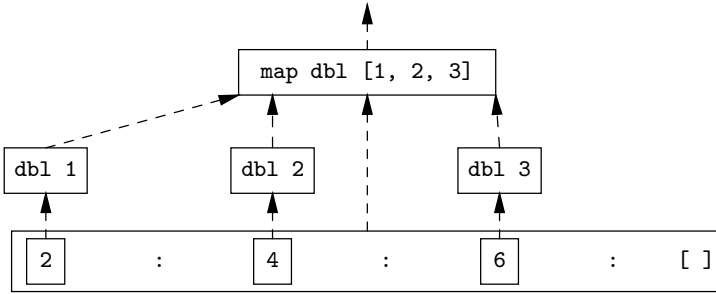


Fig. 8. The trace of `map dbl [1, 2, 3]` when `map` is trusted and `dbl` is suspected, using both the context and the applied function to determine recorded reductions.

So we simply add a ‘trusted’ flag to every name record. Whenever a function is applied, the application operator checks if the trusted flag is set for *both* the context of the application *and* the function being applied. The redex trail is only extended if this test fails.

The programmer can decide at compile-time whether or not to trust all the functions in a module by default. But run-time flags allow the default to be overridden. The implementation is a straightforward search through import tables, starting from the main module, to find the name table of the specified module. There the trusted flags are set or reset as needed. The search is performed once only, before starting the functional computation.

4.2 Trail Pruning

Selective trails should go some way to solving the problem of excessive memory demands. But even with reductions within the *Prelude* and other low-level modules excluded, a redex trail for something like a Haskell compilation may be so large as to be impractical. If tracing is to be feasible even for large computations, we need something more.

Another way of limiting the size of redex trails is to apply some kind of *pruning* when memory starts to run out. This idea prompts two questions: (1) when to prune? and (2) what to prune? A reasonable answer to the first question is to prune redex trails during each garbage collection. The garbage collector is called when available heap memory is exhausted; it already scans the heap including both normal program data and trails. Pruning can be implemented as a harsh garbage collection policy. The second question is harder. Our approach is simple: limit the *length* of all trails accessible from live program data or output. Trails beyond a certain length k are pruned, by overwriting them with a new `Trace` constructor `Pruned`.

When exploring a trace that has been pruned to length k , the user is guaranteed to be able to follow each trail at least k steps. When reaching a pruned part of the trace, the display program indicates that the information is no longer

Table 1. A comparison of memory costs for incomplete redex trails without indirecting applications and complete trails with them.

	Incomplete	Complete
<code>cichelli</code>	13 Mb	14 Mb
<code>clausify</code>	28 Mb	28 Mb
<code>primes</code>	54 Mb	85 Mb

available. For the common error that no clause in a function definition matches the actual argument (eg. `head []`), setting $k = 1$ is enough to be able to find out which function called the non-matching function (and the source position of the call). With $k = 2$, the top level of the argument will still be accessible.

5 Experimental Results

What are the practical consequences of extending the redex-trail tracer in the ways described in previous sections? To find out we first looked again at the three benchmark computations studied in our earlier paper: `cichelli` uses a brute-force search to construct a perfect hash function for a set of 16 keywords; `clausify` transforms a proposition to an equivalent in clausal form; `primes` computes the first 2,500 primes using a wheel sieve.

5.1 Extra Cost of Complete Trails

How much larger are the complete trails, including indirecting applications, than trails without them? The comparative memory costs are shown in Table 1. Recall our experience with the prototype: we felt a lack of information about some applications but not others. For `primes`, completing the trail means increasing its size by over 50%, but for `cichelli` and `clausify` the trail sizes hardly change.

5.2 Sizes of Selective Trails

Our first remark about the practical consequences of selective trails is a subjective one. We immediately appreciated the benefit of suppressing detail within the `Prelude` when examining traces.

More objectively, how much smaller are selective trails than full ones? As Table 2 shows, when `Prelude` functions are trusted, between 40% and 80% of reductions recorded in the full trails of these programs are omitted in a selective one. There is a corresponding saving in memory, but not necessarily in proportion. Some function applications have far larger representations than others depending, for example, on the number and size of arguments and whether these are shared. For `cichelli` and `clausify` the memory saving is only 20%–30% — rather less than the reduction figures might lead one to expect. For `primes`,

Table 2. The effect of excluding trace information within a trusted `Prelude`.

	Full Trail #reductions	Selective Trail #reductions	memory	No Trail memory
<code>cichelli</code>	355 k	213 k (60%)	10.0 Mb	34 kb
<code>clausify</code>	1406 k	291 k (21%)	23.2 Mb	7 kb
<code>primes</code>	981 k	322 k (33%)	0.8 Mb	60 kb

on the other hand, the fraction of memory saved far exceeds such expectations: about a third of all reductions are still recorded, yet the amount of memory needed shrinks by almost two orders of magnitude!

However, in all cases the final column of Table 2 provides a salutary reminder of how large the trails still are compared with the structures in memory for the normal untraced computation. Even selective redex trails can increase heap-memory requirements by a huge factor. Such increases may be infeasible for larger applications, confirming the need for a more drastic space-saving technique such as pruning.

5.3 How Effective Is Pruning?

To test the effectiveness of pruning we again constructed traces for the three benchmark computations `cichelli`, `clausify` and `primes`, but with trails beyond some length k pruned away at each garbage collection. Figures 9 and 10 confirm the expected result that pruning with small values of k can save a great deal of space. What the figures do *not* show is that in practice even for very small values of k one finds a good deal of information in traces — after all, the last garbage collection may have been some time ago. Even setting $k = 0$ is a worthwhile option, and the memory overhead in that case is only a small constant multiple of normal memory use.

Another observation from these graphs is that the fraction of memory that is saved for a given pruning length k varies widely between applications. Compare the effect of pruning the `clausify` trail (upper part of Figure 10) with the effect of pruning the `primes` trail (lower part of Figure 10). For `clausify`, even ‘light pruning’ with $k = 16$ is enough to remove about 95% of the trace: we have had to omit the plot for $k = \infty$, to make comparisons between other k values visible! For `primes`, however, the memory saved by pruning with $k = 16$ is only 1%–2%, barely discernible in the graph, and much harsher pruning is needed to make appreciable memory savings.

Any pruning scheme based on a fixed numeric limit has the drawback that the user may be unsure what limiting value is appropriate. An advantage of directly limiting path-length is that the user knows what losses to expect when it comes to browsing the trace. But what the user may *not* know is whether the trace will fit in available heap memory. As an alternative to fixing the value of k , programmers could specify only an overall bound on the size of heap mem-

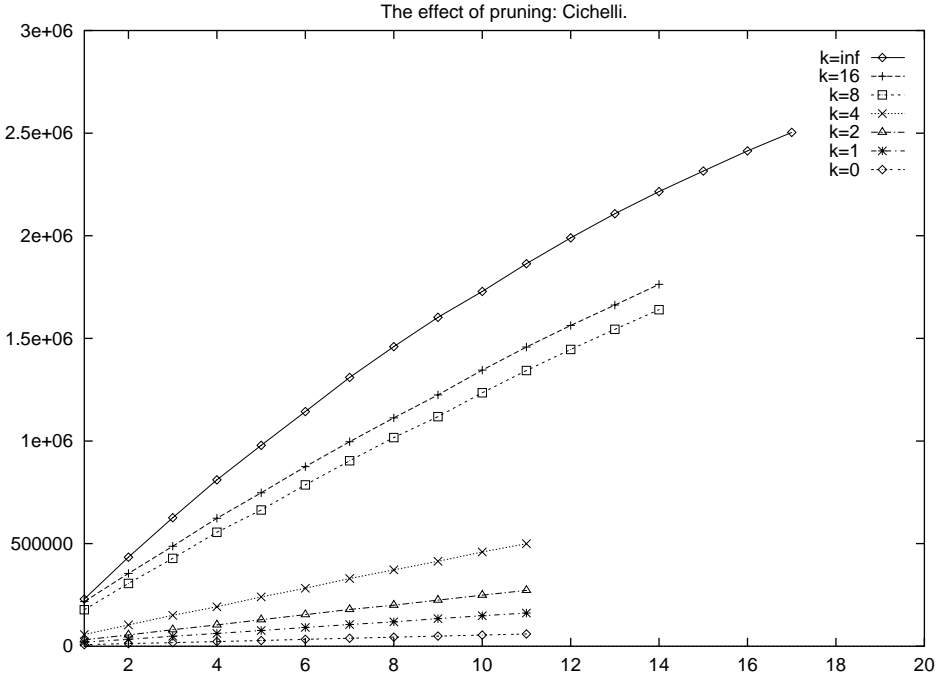


Fig. 9. The effect of pruning trails for the *Cichelli* computation beyond a fixed length k at each garbage collection. The x axis shows the number of garbage collections, and the y axis the amount of live heap memory in words.

ory. The implementation would then vary k automatically to achieve (only) the necessary degree of pruning.

5.4 Larger Applications

So far we have looked at three quite different but modest benchmark computations, in which programs of up to a few pages in length perform up to a million reductions. Completing redex trails with a record of indirecting applications has not caused a blow-up in memory demands. Partial trails require far less memory than full trails and are still effective as sources of information.

One motivation for partial trails is the wish to handle more demanding applications, beyond the capacity of our prototype system. We now give results for two such applications.

A Chess End-Game: Wurzburg-mate Our first larger application involves brute-force search to solve a chess end-game problem of the *White to move and mate in N* variety. The program, called *mate*, is not so very large — about 400

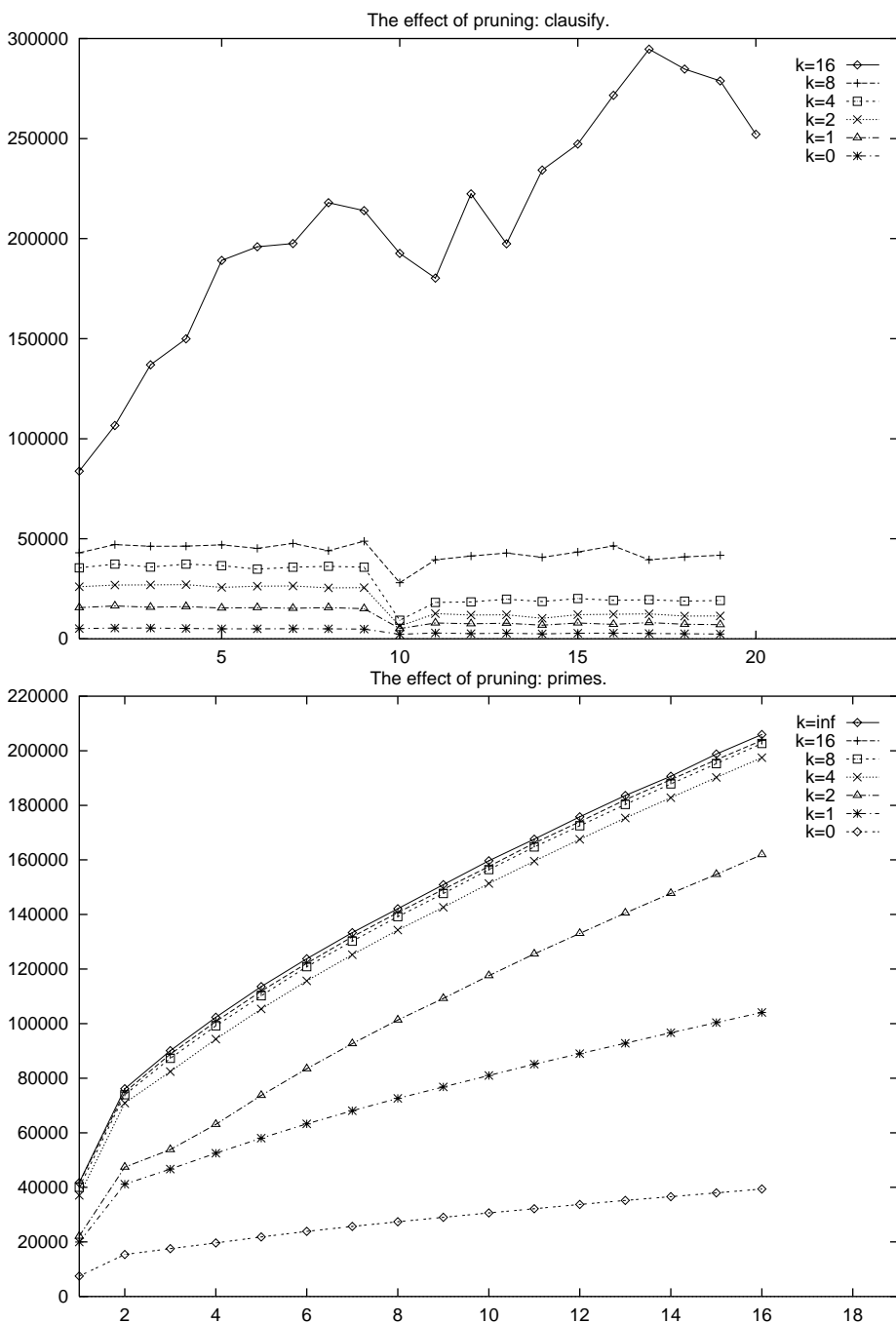


Fig. 10. The effect of pruning trails for the **clausify** and **primes** computations. Again the x axis shows the number of garbage collections, and the y axis the amount of live heap memory in words.

lines excluding **Prelude** definitions. But the problem we shall supply as input (shown in Figure 11) is sufficiently tough that the computation requires over 20 M reductions.

```
-- O. Wurzburg, Pittsburgh Gazette-Times, 1914
-- The classic pawn-promotion three-mover.

- - k - - - -
r - - - - -
- - - p K - -
- - - - P - -
- - - p p b -
- - - p - p -
- - - - -
- - - - -

White to play and mate in 3
```

Fig. 11. The problem used as input to the **mate** program. “A striking task achievement presented with due regard to artistic canons.” [4]. Chess-playing readers might like to verify that solving it requires a significant amount of computation!

As one would expect, the **mate** computation is dominated by functions testing and sifting board positions and moves. The results of many of these functions are not fresh constructions but components of arguments. So trails produced *without* indirections are far from complete: constructing a selective trail of that kind with the **Prelude** trusted requires no more than 0.5 Mb memory space. However, some of the most interesting aspects of the computation are not recorded!

So how much space is required to record the Wurzburg-mate computation in a redex trail including the so-called indirecting applications? Without some selectivity, building the trail in heap memory is not feasible — by extrapolation, it would require over 100 Mb. Table 3 shows the extent of various selective trails. Apart from the **Prelude**, the lowest level module in **mate** defines a **Board** type,

Table 3. The magnitude of redex trails for the Wurzburg-mate computation, with varying degrees of selectivity.

trusted modules	# reductions in trail	memory needed
none	20,344 k	unknown
Prelude	9,827 k	52.7 Mb
Prelude,Board	8,371 k	51.2 Mb
Prelude,Board,Move,Problem	2,571 k	21.4 Mb

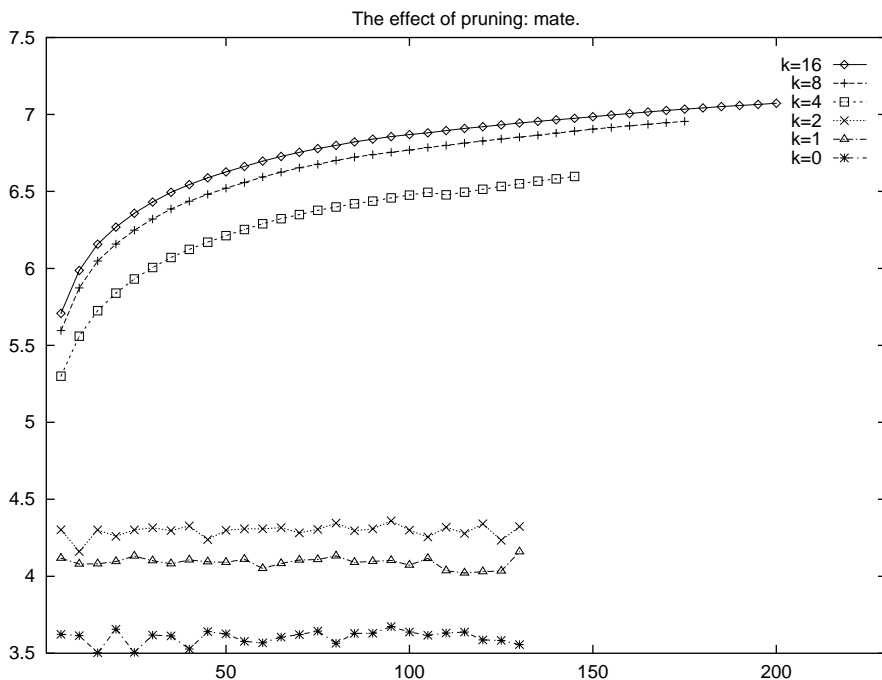


Fig. 12. The effect for the Wurzburg-mate computation of pruning trail length to k at each garbage collection, for various values of k between 0 and 16. Only the **Pre**lude module is trusted. The y -axis now uses a *log scale* with base 10: each plotted point (x, y) represents 10^y words of live heap memory recorded after x garbage collections.

and basic operations on it such as vacating a square, or occupying one with a given piece. At the next level up, the **Problem** module parses a problem statement into an initial state, and the **Move** module defines legal moves.

Even the most selective trail boosts the memory requirement from less than 10kb for an untraced (but deliberately space-efficient) search to over 20 Mb — giving some insight into the number of moves and positions the ‘suspected’ **Solution** module deals with. Also, without tracing the program runs for about two and a half minutes, but with selective trail construction this increases to between 15 and 30 minutes.

Figure 12 shows the effect for Wurzburg-mate of increasingly severe trail-pruning at each garbage collection. Light pruning ($k \geq 4$) is not enough: memory demands still grow steadily to tens of megabytes. But setting $k \leq 2$ curbs memory use very effectively: the self-tracing program runs in constant space below 100kb.

A Haskell Compilation: `nfib-nhc` Our second large-scale application is the `nhc` Haskell compiler itself. The compiler is a large program, consisting of about 100 modules with a total of over 14,000 source lines. When compiled for tracing, the compiler binary is 4.2 Mb in size, compared with 1.1 Mb for the non-traced binary. The compiler starts by reading, lexing, and parsing the Haskell input file. It then processes the interfaces for all imported files. The interface for the standard prelude is always processed. After that the source code (in the form of a syntax tree) is type checked and transformed in around 25 passes before finally the byte-code for the functions in the module is generated.

Traced programs are currently restricted to a limited repertoire of I/O operations. So for the purposes of tracing we modified the `nhc` compiler to take the source as standard input, and to deliver byte-code as standard output. Figure 13 shows the small source module supplied as input. Even compiling something as modest as the `NFib` module `nhc` takes 5.2 M reductions. But when the `Prelude` is trusted, 4.3 M of these are omitted from the final trace, leaving only about 900 k recorded reductions. We are able to trace the compilation without further pruning using around 70 Mb of memory to store the redex trail. The redex trail for the compilation can be accessed using the display program by clicking over individual instructions or operands in the byte-code output.

```
module NFib where

nfib :: Int -> Int
nfib 0 = 1
nfib 1 = 1
nfib n = 1 + nfib (n-1) + nfib (n-2)
```

Fig. 13. Input for the `nfib-nhc` computation.

6 Related Work

The problem of tracing functional computations has been tackled previously by numerous researchers. In his recent thesis [11], Watson usefully reviews over twenty different schemes. But many proposed solutions have been unsuitable for full-scale applications. We concentrate here on a selection of previous work specifically addressing the problems of working with large computations.

Five years ago, Andrew Tolmach described in his thesis [10] a comprehensive tracing and debugging system for the *eager* functional language ML. A shorter and more recent account is given in [9]. Tolmach's system, like ours, worked by compile-time transformation of the program. He also happened to use machines very similar to our own (MIPS workstations with about 100 Mb of memory). He successfully applied his system to programs of up to 750 source lines, performing computations of over 50 million *events* (such as the binding of a value to

a name — events are not directly comparable with reductions, but at a similar level). The overhead in execution time was impressively low: even compared with highly optimised normal compilation, the slow-down was only about a factor of three. However, the basis of tracing in Tolmach’s system was navigation through a *sequence of states*, where state information was recorded only at check-point intervals of many thousand events — up to 256,000 for the largest application. This interval, and the size of a *check-point cache* were the main ways of controlling memory use. Because of the sparse recording policy during traced execution, *repeated computation* was needed to provide information about events of interest to the programmer, whereas we record all available information in the trace structure. Though our aims are similar, at a detailed level our techniques are quite different, partly because of differences in evaluation strategy between an eager language such as ML and a lazy one such as Haskell.

In a paper at last year’s IFL workshop [1] Jarvis and Morgan reported their experience extending the *cost-centre profiler* in Glasgow’s GHC compiler [7] for use with large programs. Their motivating application was the LOLITA natural language system — at over 50,000 source lines one of the largest Haskell applications to date. Cost-centres are labels for parts of a program, added by the programmer as annotations (or by the compiler for default options such as one centre per top-level definition). Jarvis and Morgan introduced *cost-centre stacks* to gain precision when attributing costs of functions applied in many places, though the cost of maintaining distinct records for each permutation on the stack grows to tens of megabytes for LOLITA. As a side benefit, they found that the current stack of names provides a useful trace of the evaluation context when an error occurs in a large program. Full redex trails in our system carry all the information in a cost-centre stack, and far more besides.

Nilsson and Sparud use a trace structure similar in some respects to redex trails but with *forward* rather than backward links. To cope with larger computations they propose in [3] (and more fully in [2]) to build only some initial part of the trace when the program is run. When the programmer examines the trace, if a point is reached where the trace goes no further, the program is *re-executed* to obtain a trace extension at the point where it is needed. This process can be repeated many times. It is not clear how to use this incremental technique with backward traces, but the idea is interesting since it allows tracing in full detail for parts of a large computation.

We have already mentioned Watson’s thesis [11] for its useful review. Watson’s own approach to the tracing problem also involves transforming the program to augment values with traces of their computation, but he chooses to construct low-level traces recording the *sequence of evaluation*. His interpretive prototype incurs very high costs in both space and time, so he cannot demonstrate large applications. However, in a later chapter discussing the problems of scale, Watson discusses selective tracing and proposes a scheme much like our own for excluding trusted applications in a trusted context.

7 Summary, Conclusions, and Future Work

We had two goals. The first was to extend our redex trail system so that in principle a *complete* record of a computation, including every reduction, can be made. In practice, complete records are often prohibitively expensive and overwhelm the user of the tracing system with too much information. So our second goal was to develop effective ways of specifying and constructing *partial* redex trails, recording only the reduction steps most likely to be of interest to the programmer. An important motivation for partial trails was to make tracing practical even for large computations.

We have attained both goals. Completeness has been achieved by extending our trail representation to incorporate indirections to exactly those applications previously unrecorded. By combining selective construction with controlled pruning we are able to trace much larger applications than before.

We conclude that the approach of compile-time transformation to generate redex trails is viable for a full-scale tracing system — something that implementations of lazy higher-order languages have lacked for too long.

We intend to develop the techniques further. As noted in Section 5.3, it would sometimes be more convenient to specify a bound on the size of heap memory than on the length of trails; this should be straightforward using a two-pass pruning routine. We also plan an option to archive redex trails to file as the computation proceeds: this will not only save space in main memory, but also allow traces to be re-examined as often as the programmer wishes.

The price we currently pay for these tracing techniques includes greatly increased execution times: programs take about *fifteen times* longer to run when constructing redex trails than when executed normally. This is not acceptable. We have begun work on low-level replacements for the traced application combinators defined in Haskell. Another help towards faster partial tracing would be some way to link modules compiled to build redex trails with others compiled normally. It is not clear how to achieve such linkage as the types of traced and untraced versions of a function differ; we suspect that naive conversion (stripping trails from values, or adding null trails to values) would be too expensive. By whatever means, however, our aim is to bring the speed of traced computation within a factor of two or three of normal speed in most cases.

We have said very little about the way programmers examine traces, though it is an important issue. We have constructed a special-purpose browser for redex trails, as described in [8]. This browser is normally applied when a computation is over, with the resulting output or run-time error as the starting point. But trail-building computations can also be interrupted, in which case the browser is attached to the trail for whatever is currently being evaluated; this mode of use allows non-terminating computations to be investigated.

Ultimately any claim to practical success in the field of tracing must be tested by experiments measuring the effectiveness of tracing tools in actual programming tasks. We hope to conduct such experiments in the near future. Meanwhile we have in our sights what many implementors would regard as the ultimate

test of a tracer they themselves would wish to use — constructing an informative trace of a boot-strapping compilation.

Acknowledgements

Niklas Røjemo built the `nhc` compiler that we have adapted for our experiments in tracing. The work reported in this paper was funded by the Engineering and Physical Sciences Research Council (ROPA grant GR/K64334). Related implementation work was funded by Canon Research Europe.

References

- [1] S. A. Jarvis and R. G. Morgan. The results of: profiling large-scale lazy functional programs. In W. Kluge, editor, *Implementation of Functional Languages, 8th International Workshop (IFL'96), Selected Papers*, pages 200–221. Springer LNCS 1268, September 1996.
- [2] Henrik Nilsson and Jan Sparud. The evaluation dependence tree: an execution record for lazy functional debugging. Research Report LiTH-IDA-R-96-23, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, August 1996.
- [3] Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Journal of Automated Software Engineering*, 4(2):152–205, April 1997.
- [4] H. Phillips. *The week-end problems book*. Nonesuch Press, 1932.
- [5] N. Røjemo. Highlights from `nhc` — a space-efficient Haskell compiler. In *Proc. 7th Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 282–292. ACM Press, June 1995.
- [6] N. Røjemo and C. Runciman. Lag, drag, void and use — heap profiling and space-efficient compilation revisited. In *Proc. Intl. Conf. on Functional Programming*, pages 34–41. ACM Press, June 1996.
- [7] P. M. Sansom and S. L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–85, March 1997.
- [8] J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. In *Proc. 9th Intl. Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, September 1997.
- [9] A. Tolmach and A. W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, April 1995.
- [10] A. P. Tolmach. *Debugging Standard ML*. PhD thesis, Princeton, USA, October 1992.
- [11] R. D. Watson. *Tracing Lazy Evaluation by Program Transformation*. PhD thesis, Southern Cross, Australia, October 1996.

Engineering Large Parallel Functional Programs

Hans-Wolfgang Loidl¹ and Phil Trinder²

¹ Department of Computing Science, University of Glasgow, Glasgow, Scotland, U.K.

E-mail: hwloidl@dcs.glasgow.ac.uk

² The Computing Department, The Open University, Milton Keynes, England, U.K.

E-mail: P.W.Trinder@open.ac.uk

Abstract. The design and implementation of useful programming languages, whether sequential or parallel, should be driven by large, realistic applications. In constructing several medium- and large-scale programs in Glasgow Parallel Haskell, GPH, a parallel extension of Haskell, the group at Glasgow has investigated several important engineering issues:

- *Real Application Parallelism.* The programs achieve good wall-clock speedups and acceptable scale-up on both a shared-memory and a distributed memory machine. The programs typify a number of application areas and use a number of different parallel paradigms, e.g. pipelining or divide-and-conquer, often combining several paradigms in a single program.
- *Language Issues.* Although the largely implicit parallelism in GPH is a satisfactory programming model in general the base constructs for introducing and controlling parallelism tend to obfuscate the semantics of large programs. As a result we developed evaluation strategies, a more abstract, and systematic mechanism for introducing and controlling parallelism.
- *Engineering Environment.* The development and performance tuning of these programs emphasised the importance of an integrated engineering environment. In the process we have refined components of this environment like the simulator, the runtime system, and the profiling tools.

1 Introduction

The development and especially the performance tuning of a parallel lazy algorithm requires more than just a compiler that incorporates parallelism directives for a parallel runtime-system. In the absence of a system for implicit parallelism, the programmer needs some form of language support for exposing parallelism and tools for examining the dynamic behaviour of the execution. These aspects become painfully clear when the program to be parallelised is fairly large.

We have recently constructed an integrated programming environment to support programming in Glasgow Parallel Haskell, GPH, a parallel extension of Haskell. The environment consists of a highly parameterised simulator, GRANSIM, as well as a portable parallel runtime-system, GUM. This environment enables the programmer to develop a parallel program in an idealised setting of a simulator before focusing on machine specific aspects. A set of visualisation tools,

available for both systems, shows activity, granularity, and resource management on several levels of detail. Both GRANSIM and GUM are built on top of the Glasgow Haskell Compiler, GHC, using its support for program analysis, optimisation, and profiling.

The reasons for undertaking the time consuming effort of parallelising medium-to large-scale programs are threefold:

- The programs allow us to investigate whether GPH can achieve wall-clock speedups and acceptable scale-up for a range of real applications on both distributed- and shared-memory machines.
- Large programs are essential to evaluating the strengths and weaknesses of any language. This is especially true of functional languages whose modularity and high level of abstraction help to structure large programs. In addition to the aspects of engineering sequential applications, we investigate the language constructs for introducing and controlling parallelism.
- Large programs also test our implementation technology and tools. They reveal whether our implementation techniques scale-up to handle large bodies of code running for extended periods. They also test whether our profiling and visualisation tools provide us with the information required to tune the parallel behaviour.

Most of the programs we have studied entail symbolic computation with functions operating on complex data structures. These programs are suitable because they make greater use of the strengths of functional languages like algebraic data types and higher-order functions. Our hope is to parallelise the program with limited effort, and certainly without a total rewrite. This approach is made possible because a non-strict functional language does not enforce a specific evaluation order and is in stark contrast to the trend in parallel (super-) computing where considerable programming time is invested in every program.

We have written many small GPH programs, several medium-sized programs and parallelised one large program, the Lolita natural language engineering system, which is described in detail in [7]. These programs use various different parallelism paradigms, mostly fine-grained parallelism, and in some cases a limited, benign form of speculative parallelism. In this paper we discuss the following programs:

- *Alpha-Beta search*, a program for performing a heuristic search in a tree structure, usually used in game programming. It is a typical program for AI applications.
- *Accident Blackspots*, a program for locating accident blackspots from police traffic accident reports. It is typical of data-intensive complex-query programs.
- *LinSolv*, a program for finding an exact solution of a system of linear equations. It uses a structure typical to many algorithms in computer algebra.

The structure of the remainder of the paper is as follows. Section 2 describes both parallelism in GPH, and evaluation strategies. Section 3 discusses the guidelines we are developing for parallelising large functional programs. Section 4 describes several parallel variants of the Alpha-Beta search algorithm. Section 5

describes the parallelisation of the accident blackspots program, including obtaining wall-clock speedups. Section 6 outlines the parallelisation of the LinSolv linear equation solver. Section 7 discusses what has been learnt from engineering these programs including design lessons for the parallel implementation of functional languages.

2 GPH and Evaluation Strategies

2.1 GPH Parallelism

The essence of the problem facing the parallel programmer is that, in addition to specifying *what* value the program should compute, explicitly parallel programs must also specify *how* the machine should organise the computation. In GPH we have to specify only a few aspects of the parallel execution, with the runtime system managing details like synchronisation of threads and transfer of data. The programmer is only required to indicate those values that might usefully be evaluated by parallel threads and, since our basic execution model is a lazy one, perhaps also the extent to which those values should be evaluated. We term these programmer-specified aspects the program's *dynamic behaviour*.

Parallelism is introduced in GPH by the **par** combinator, which takes two arguments that are to be evaluated in parallel. The expression **p** '**par**' **e** has the same value as **e**, and is not strict in its first argument, i.e. \perp '**par**' **e** has the value of **e**. Its dynamic behaviour is to indicate that **p** could be evaluated by a new parallel thread, with the parent thread continuing evaluation of **e**. We say that **p** has been *sparked*, and a thread may subsequently be created to evaluate it if a processor becomes idle. Since the thread is not necessarily created, **p** is similar to a *lazy future* [10].

Since control of sequencing can be important in a parallel language [12], we introduce a sequential composition operator, **seq**. If **e1** is not \perp , the expression **e1** '**seq**' **e2** has the value of **e2**; otherwise it is \perp . The corresponding dynamic behaviour is to evaluate **e1** to weak head normal form (WHNF) before returning **e2**. Note that this presentation of strategies is based on Haskell 1.2.

2.2 Evaluation Strategies

Evaluation strategies use lazy higher-order functions to separate the two concerns of specifying the algorithm and specifying the program's dynamic behaviour. A function definition is split into two parts, the algorithm and the strategy, with values defined in the former being manipulated in the latter. The algorithmic code is consequently uncluttered by details relating only to the parallel behaviour. In fact the driving philosophy behind evaluation strategies is that *it should be possible to understand the semantics of a function without considering its dynamic behaviour*. A complete description and discussion of strategies can be found in [15].

A strategy is a function that specifies the dynamic behaviour required when computing a value of a given type. A strategy makes no contribution towards

the value being computed by the algorithmic component of the function: it is evaluated purely for effect, and hence it returns just the nullary tuple `()`.

```
type Strategy a = a -> ()
```

2.3 Strategies Controlling Evaluation Degree

The simplest strategies introduce no parallelism: they specify only the evaluation degree. The simplest strategy is termed `r0` and performs no reduction at all. Perhaps surprisingly, this strategy proves very useful, e.g. when evaluating a pair we may want to evaluate only the first element but not the second.

```
r0 :: Strategy a
r0 _ = ()
```

Because reduction to WHNF is the default evaluation degree in GPH, a strategy to reduce a value of any type to WHNF is easily defined:

```
rwhnf :: Strategy a
rwhnf x = x 'seq' ()
```

Many expressions can also be reduced to *normal form* (NF), i.e. a form that contains *no* redexes, by the `rnf` strategy. The `rnf` strategy can be defined over built-in or data types, but not over function types or any type incorporating a function type as few reduction engines support the reduction of inner redexes within functions. Rather than defining a new `rnfX` strategy for each data type `X`, it is better to have a single overloaded `rnf` strategy that works on any data type. The obvious solution is to use a Haskell type class, `NFData`, to overload the `rnf` operation. Because NF and WHNF coincide for built-in types such as integers and booleans, the default method for `rnf` is `rwhnf`.

```
class NFData a where
  rnf :: Strategy a
  rnf = rwhnf
```

For each data type an instance of `NFData` must be declared that specifies how to reduce a value of that type to normal form. Such an instance relies on its element types, if any, being in class `NFData`. Consider lists and pairs for example.

```
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x 'seq' rnf xs

instance (NFData a, NFData b) => NFData (a,b) where
  rnf (x,y) = rnf x 'seq' rnf y
```

2.4 Combining Strategies

Because evaluation strategies are just normal higher-order functions, they can be combined using the full power of the language, e.g. passed as parameters or composed using the function composition operator. Elements of a strategy are combined by sequential or parallel composition (**seq** or **par**). Many useful strategies are higher-order, for example, **seqList** below is a strategy that sequentially applies a strategy to every element of a list. The strategy **seqList r0** evaluates just the spine of a list, and **seqList rwhnf** evaluates every element of a list to WHNF.

```
seqList :: Strategy a -> Strategy [a]
seqList strat []      = ()
seqList strat (x:xs) = strat x 'seq' (seqList strat xs)
```

2.5 Data-Oriented Parallelism

A strategy can specify parallelism and sequencing as well as evaluation degree. Strategies specifying data-oriented parallelism describe the dynamic behaviour in terms of some data structure. For example **parList** is similar to **seqList**, except that it applies the strategy to every element of a list in parallel.

```
parList :: Strategy a -> Strategy [a]
parList strat []      = ()
parList strat (x:xs) = strat x 'par' (parList strat xs)
```

Data-oriented strategies are applied by the **using** function which applies the strategy to the data structure **x** before returning it.

```
using :: a -> Strategy a -> a
using x s = s x 'seq' x
```

A parallel map is a useful example of data-oriented parallelism; for example the **parMap** function defined below applies its function argument to every element of a list in parallel. Note how the algorithmic code **map f xs** is cleanly separated from the strategy. The **strat** parameter determines the dynamic behaviour of each element of the result list, and hence **parMap** is parametric in some of its dynamic behaviour.

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs = map f xs 'using' parList strat
```

2.6 Control-Oriented Parallelism

In control-oriented parallelism subexpressions of a function are selected for parallel evaluation. A control-oriented strategy is typically a sequence of strategy applications composed with **par** and **seq** that specifies which subexpressions of a function are to be evaluated in parallel, and in what order. The sequence

is loosely termed a strategy, and is invoked by either the **demanding** or the **sparking** function. The Haskell **flip** function simply reorders a binary function's parameters.

```
demanding, sparking :: a -> () -> a
```

```
demanding = flip seq
sparking  = flip par
```

Several control-oriented strategies, including an example of how to describe a thresholding mechanism as part of a strategy, are discussed in the accompanying paper [7].

3 Parallelisation Guidelines

From our experiences engineering several parallel programs using evaluation strategies we are developing guidelines for parallelising large non-strict functional programs. The approach is top-down, starting with the top level pipeline, and then parallelising successive components of the program. The first five stages are machine-independent. Our approach uses several ancillary tools, including time profiling [14] and the GRANSIM simulator [1]. Several stages use GRANSIM, which is fully integrated with the GUM parallel runtime system [17]. A crucial property of GRANSIM is that it can be parameterised to simulate both real architectures and an idealised machine with, for example, zero-cost communication and an infinite number of processors.

The stages in our methodology are as follows.

1. *Sequential implementation.* Start with a correct implementation of an inherently-parallel algorithm or algorithms.
2. *Seek top-level parallelism.* Often a program will operate over independent data items, or the program may have a number of stages, e.g. lex, parse and typecheck in a compiler. Both top-level data parallelism and pipelining are very easy to specify, and often gain some parallelism for minimal change.
3. *Time Profile* the sequential application to discover the “big eaters”, i.e. the computationally intensive pipeline stages.
4. *Parallelise Big Eaters* using evaluation strategies. It is sometimes possible to introduce adequate parallelism without changing the algorithm; otherwise the algorithm may need to be revised to introduce an appropriate form of parallelism, e.g. divide-and-conquer or data-parallelism.
5. *Idealised Simulation.* Simulate the parallel execution of the program on an idealised execution model, i.e. with an infinite number of processors, no communication latency, no thread-creation costs etc. This is a “proving” step: if the program isn’t parallel on an idealised machine it won’t be on a real machine. This often indicates that some restructuring of the code is necessary to achieve good parallel performance. We now use GRANSIM, but have

previously used HBCPP [13]. A simulator is often easier to use, more heavily instrumented, and can be run in a more convenient environment, e.g. a workstation.

6. *Realistic Simulation.* GRANSIM can be parameterised to closely resemble the GUM runtime system for a particular machine, forming a bridge between the idealised and real machines. A major concern at this stage is to improve thread granularity so as to offset communication and thread-creation costs.
7. *Real Machine.* In an execution of the program on a real machine further aspects, such as the impact of operating system calls on the parallel performance, have to be considered. In order to analyse the parallel behaviour of the program, the GUM runtime system supports some of the GRANSIM performance visualisation tools. This seamless integration helps understand real parallel performance.

It is more conventional to start with a sequential program and then move almost immediately to working on the target parallel machine. This has often proved highly frustrating: the development environments on parallel machines are usually much worse than those available on sequential counterparts, and, although it is crucial to achieve good speedups, detailed performance information is frequently not available. It is also often unclear whether poor performance is due to use of algorithms that are inherently sequential, or simply artifacts of the communication system or other dynamic characteristics. In its overall structure our methodology is similar to others used for large-scale parallel functional programming [3].

4 Alpha-Beta Search

The first example program is the Alpha-Beta search algorithm, typical of artificial intelligence applications. It is mainly used for game-playing programs to find the best next move by generating all possible moves up to a certain depth, applying a static evaluation function to each of the leaves in this search tree, and combining the result by picking the best move for the player assuming that the opponent picks the worst move for the player.

We discuss two versions of the Alpha-Beta search algorithm: a *simple* version, and a *pruning* version. Both versions are based on the Miranda¹ code presented by John Hughes [4] in order to demonstrate the strengths of lazy functional languages. The pruning version relies on laziness to improve the efficiency of the sequential algorithm by pruning the search tree based on intermediate results. In this section we parallelise both versions and study the parallel runtime behaviour. We investigate the use of strategies to develop an efficient parallel algorithm without sacrificing the advantages of the original lazy algorithm. A more detailed comparison of both algorithms with a discussion of their parallel performance is given in [8].

¹ Miranda is a trademark of Research Software Ltd.

4.1 Simple Algorithm

In the simple algorithm each possible next move is evaluated independently yielding a compositional structure of the algorithm. The result is either the maximum (player's move) or the minimum (opponent's move) of the evaluations of all next positions. This algorithm can be very naturally described as a sequence of function compositions performing the following tasks:

1. Build a tree with positions as nodes and all possible next moves as subtrees.
2. Prune the tree, which might be infinite at this stage, to a fixed depth to bound the search.
3. Map a static evaluation function over all nodes of the tree.
4. Crop off subtrees from winning or losing positions. If such a position is found it is not necessary to search deeper in a subtree.
5. Finally, pick the maximum or minimum of the resulting evaluations in order to determine the value of the current position via `mise f g`. The functions `f` and `g` represent the combination functions for the two players and alternate when traversing the tree.

Dynamic Behaviour. The compositional nature of this algorithm makes parallelisation rather easy. For both versions of the algorithm we study the following three sources of parallelism:

Parallel Static Evaluation Function. The idea of a parallel static evaluation function is to reduce the costs of the function, which will be mapped over the leaves of the pruned search tree. In our example implementations, the static evaluation function is very simple: it computes the distance of the current position to a set of known winning positions. The parallel version computes all distances in parallel.

Parallel Higher-order Functions over Trees. Parallelising the definitions of some higher-order functions is a bottom-up approach. It can be used for the parallelisation of many functional programs. In this case we use a parallel version of a map function over search trees.

Data Parallelism over All Possible Next Moves. In a data parallel approach the goal is to evaluate all possible next moves in parallel. It is a top-down approach and turns out to be the best source of parallelism for such a compositional algorithm with no dependencies between the evaluations of the subtrees. A simple `parMap rnf` strategy can be used to capture the dynamic behaviour of this function. The only necessary change in the algorithm affects the `mise` function in Stage 5 of the algorithm, shown in Figure 1. As arguments to this function either the binary `max` or `min` function is folded over the list of results from the subtrees. Note that the functions `f` and `g` change position in the recursive call to record the switch in turns.

```
-- This does simple minimaxing without pruning subtrees
mise :: Player -> Player -> (Tree Evaluation) -> Evaluation
mise f g (Branch a []) = a
mise f g (Branch _ l) = foldr f (g OWin XWin) (parMap rnf (mise g f) l)
```

Fig. 1. Data parallel combination function in the simple Alpha-Beta search algorithm.

Performance Measurements. Our measurements of both versions of the algorithm under the GRANSIM simulator are summarised in Table 1. In all test runs we used a GRANSIM setup modelling a tightly connected distributed memory machine with 32 processors, a latency of 64 machine cycles, and bulk fetching. The first four data columns of this table show the results of the simple algorithm when using the different sources of parallelism. All runtimes are given in machine-independent kilocycles. The work column measures the total work compared to a sequential run and is therefore a measure of the redundant work, in particular of speculative parallelism. The three horizontal sections in the table represent three different positions that have been analysed: a standard opening position (I), a winning position (II), and a position generating a large search tree (III).

Table 1. Measurements of the simple and the pruning Alpha-Beta search algorithm

	Simple Algorithm				Pruning Algorithm			
	Runtime (kilocycles)	Avg Par	Total Work	Speedup	Runtime (kilocycles)	Avg Par	Total Work	Speedup
<i>Position I</i>	(standard opening)							
Sequential	60,297				34,363			(1.75)
Par Static Eval	21,091	3.1	108%	2.85	12,099	3.1	109%	2.84
Data Par	3,503	26.4	153%	17.21	2,265	23.7	156%	15.17
Par h.o. fcts	4,954	20.9	172%	12.16	4,248	24.2	299%	8.08
<i>Position II</i>	(early solution)							
Sequential	4,427				4,703			(0.94)
Par Static Eval	1,772	2.9	116%	2.49	1,898	2.9	117%	2.47
Data Par	1,152	13.9	362%	3.84	1,075	13.1	299%	4.37
Par h.o. fcts	759	9.6	165%	5.83	811	9.0	155%	5.79
<i>Position III</i>	(large search tree)							
Sequential	145,720				90,377			(1.61)
Par Static Eval	48,808	3.3	111%	2.98	29,891	3.3	109%	3.02
Data Par	6,621	29.1	132%	22.00	7,699	16.2	138%	11.73
Par h.o. fcts	9,345	21.4	137%	15.59	8,093	24.6	220%	11.16

The *parallel static evaluation function* generates conservative parallelism shown by the small amount of total work performed. However, the resulting parallelism is rather small and very fine-grained, yielding a rather poor speedup. The *data parallelism* over all next positions proves to be the best source of parallelism. The simple algorithm will only cut-off subtrees if it finds a winning position in one of the subtrees. Therefore, this data parallelism is conservative except for the case where a winning position is found as in Position II. Finally, the *higher order functions* approach generates the largest amount of redundant work for Positions I and III, which is shown by the high total work percentage. Here a parallel map of the static evaluation function is used. However, this also maps the evaluation function on nodes that are actually pruned in the sequential algorithm.

4.2 Pruning Algorithm

The simple algorithm described in the previous section lacks one crucial optimisation of the Alpha-Beta search: the pruning of subtrees based on intermediate results. The pruning algorithm returns an increasing list (player's move) of approximations with the exact value as last list element rather than a single value. The main pruning function, `minleq`, has to test whether the opponent's move from a subtree, represented as a decreasing list, can be ignored. This is the case if the worst result of the decreasing list l , i.e. its minimum, is no better, i.e. less than or equal to, the intermediate result x . Or more formally: $\min l \leq x \Leftrightarrow \text{minleq } l \ x$. Since `minleq` works on decreasing lists it can stop examining the list as soon as it finds a value less than x . Thus, laziness is used to ignore parts of the list of approximations, which amounts to pruning subtrees in the search tree. A complete description of this lazy functional pruning algorithm can be found in [4].

Dynamic Behaviour. Unfortunately, the pruning version seriously complicates the parallelisation of the algorithm. We have already seen in the simple algorithm that the most promising source of parallelism is the parallel evaluation of all next positions. However, using a simple `parList rnf` strategy over all next positions is no longer advisable, since this might result in a lot of redundant work, if many subtrees can be pruned. The measurements of the data parallel strategy on the pruning algorithm in Table 1 show a rather high degree of redundant work. In fact, in the data parallel strategy on Position III the parallel simple version is even faster than the highly speculative pruning version of the algorithm!

A better approach for parallelisation is to force only an initial segment in the list of possible next positions. We call the length of this segment the "force length". We have experimented with static force lengths as well as dynamic force lengths that depend on the level in the search tree. To date the best results have been obtained from using a static force length as shown in the code in Figure 2. Note that the force length represents a trade-off between increasing the degree of parallelism and reducing the total amount of work being done.

```

-- Parallel version of the pruning version
mise :: Player -> Player -> (Tree Evaluation) -> [Evaluation]
mise f g (Branch a []) = [a]
mise f g (Branch _ l) = -- force the first n elems of the result list
  f ((map (mise g f) l)
    'using' \ xs -> if force_len==-1 -- infinity
      then parList rnf xs 'par' ()
      else parList rnf (take force_len xs) 'par'
        parList rwhnf (drop force_len xs) 'par' () )

```

Fig. 2. Strategy for a pruning Alpha-Beta search with a static force length

Performance Measurements. Figure 3 compares the speedups of the pruning version of Alpha-Beta search under GRANSIM in the same setup as in the previous measurements. The x-axis shows the static force length, the y-axis the speedup. The left hand graph uses a program implementing tic-tac-toe, the right hand graph uses an implementation of a similar game, escape, with a search space of comparable size but asymmetric winning conditions.

The left hand graph shows for the data parallel strategy a large improvement when increasing the force length, in particular for Position III. A purely conservative data parallel strategy (i.e. the force length is 0) achieves a speedup of only 8.58 because the amount of available parallelism drops early on in the computation. In contrast, with a force length of 4 the speedup is 15.71. After that the percentage of redundant work done in the parallel algorithm increases too much to achieve a further improvement. For Position II, which finds a winning position early on in the search, parallelism can achieve hardly any improvement because almost all potential parallelism in the algorithm is pruned. The right hand side of Figure 3 shows an even larger improvement with increasing force length. In both cases the versions additionally using a parallel static evaluation function usually outperform the versions with data parallelism alone. The small amount of conservative parallelism in the static evaluation can be usefully exploited while the machine is idle.

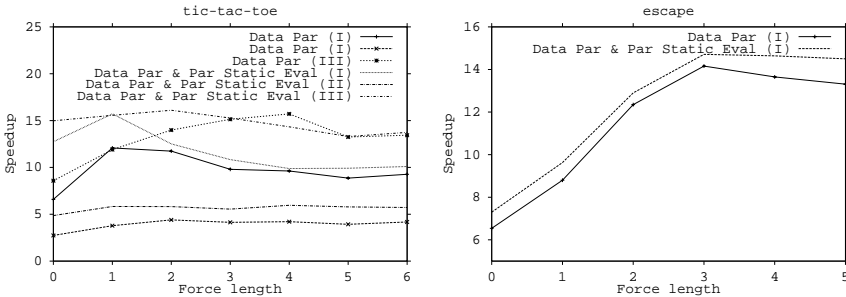


Fig. 3. Speedup of a pruning Alpha-Beta search with varying static force length

5 Accident Blackspots

This section outlines a data-intensive GPH program that solves a real problem using real data and achieves good wall-clock speedups on two very different parallel machines: a shared memory Sun SPARCserver and a distributed-memory network of workstations. A more detailed description of the program and parallelisation can be found in [16].

5.1 Problem Description

Given a set of police accident records (modified only to preserve privacy) the task is to discover accident blackspots: locations where two or more accidents have occurred. A number of criteria can be used to determine whether two accident reports are for the same location: they may have occurred at the same junction number, at the same pair of roads, at the same grid reference, or within a small radius of each other. The problem amounts to combining several partitions of a set into a single partition. For example if the partition on road pairs is $\{\{2,4,5\},\{3\},\{6,7\}\}$ and on grid references is $\{\{2,5\},\{3\},\{4,6\},\{7\}\}$, the combined partition is $\{\{2,4,5,6,7\},\{3\}\}$.

5.2 Sequential Implementations

The PFL Implementation. The application was originally written at the Centre for Transport Studies [18] in PFL and has subsequently been rewritten in Haskell. PFL is an interpreted functional language [11], designed specifically to handle large deductive databases. Unusually for a functional language, PFL provides a uniform persistent framework for both data and program.

The Haskell Implementation. The Haskell implementation constructs a binary relation containing an element for each pair of accidents that match under one of the four conditions. The combined partition is formed by repeatedly finding all of the accidents reachable in `sameSite` from a given accident. The program has four major phases: reading and parsing the file of accidents; building indices over the accident data; constructing `sameSite`, and indices over `sameSite`; forming the partition. The program is a 300-line module, together with 3 library modules totalling 1300 lines.

5.3 Parallel Variants

Following our guidelines, we initially investigated the application's parallelism using an idealised simulation. Once adequate parallelism was obtained, we used a realistic simulation of our first 4-processor shared-memory target machine. Table 2 reports the results obtained from the simulators when just 1000 accidents are partitioned, runtimes and work are in GRANSIM megacycles.

Table 2. Idealised simulation of Blackspots

Parallel Variant	Work (megacycles)	Average Parallelism	Runtime (megacycles)
Pipeline only	327	1.2	273
Par. Pipeline Stages	335	2.8	124
Par. Pipeline Stages & preconstructed Ixs	304	3.5	87
Geographically Partitioned	389	3.7	105

Pipeline only. The first version simply converts the 4 phases of the program outlined in Section 5.2 into a pipeline. The speedup of 1.2 is disappointingly low, because the pipeline is blocked by the trees passed between stages.

Parallel Pipeline Stages. The next version introduces parallelism within each pipeline stage using a variety of paradigms. The file reading and parsing stage is made data parallel by partitioning the data and reading from n files. Control parallelism is used to construct the accident indices. The stages constructing the same-site relation and the partition both use benign speculative parallelism.

Parallel Pipeline Stages and Preconstructed Indices. Parallelism is further improved by merging the first two pipeline stages. That is, the indices on the accident data were constructed before the program is run, and the program reads the indices rather than constructing them. The resulting parallelism is satisfactory on an idealised simulation of a 4-processor machine, but poor under a realistic simulation. The poor realistic results are due to the fine-grained parallelism and the volume of data being communicated.

Table 3. Realistic SPARCserver simulation of Blackspots

Parallel Variant	Work (megacycles)	Average Parallelism	Runtime (megacycles)
Par. Pipeline Stages & preconstructed Ixs	393	2.3	171
Geographically Partitioned	394	3.7	105

Geographically Partitioned. A very different, coarse-grained, parallel structure can be obtained by splitting the accident data into geographical areas, each area, or *tile*, can be partitioned in parallel before aggregating the results. Accidents occurring near the edge of a tile must be treated specially. In fact this approach is only feasible because every accident has a grid reference and we assume that accidents occurring more than 200m apart cannot be at the same site. Accidents

occurring within 100m of the nominal edge between two tiles are duplicated in both tiles. Splitting the original data into 4 tiles results in a 4% increase in data volume.

Breaking the data into tiles reduces the work required to form a partition as long as the border is much smaller than the body of the tile. Less work is required because each accident is compared with fewer accidents: the trees constructed during the partition are smaller.

Our environment, in particular GRANSIM and strategies, have allowed us to carry out low-cost experiments with several possible parallel variants of the program. Most of the program, the partitioning algorithm in particular, remained substantially unchanged while different parallel strategies were investigated. The tiled variant is selected for execution on the real machine because it delivers good coarse-grained parallelism under both idealised and realistic simulation.

5.4 Apparatus

Machines. The program is measured on two very different machines, making use of the portability of the GUM runtime system. One is a shared-memory architecture and the other distributed-memory. The shared-memory machine is a Sun SPARCserver with 4 Sparc 10 processors and 256Mb of RAM. The machine is shared with other users, but measurements are performed when it is very lightly loaded. The distributed-memory machine is a network of up to 16 Sun 4/15 workstations each with 24Mb of RAM, and connected on a single ethernet segment.

Data. The original data set of 7300 accident records occupies 0.3Mb and is split into 2 different sized tiles: 8 small tiles of ca. 1000 accidents occupying 37Kb, and 4 large tiles of ca. 2000 accidents occupying 73Kb. Both architectures use a shared file system, via NFS, and are warm started, i.e. the program is run at least once before measurements are taken. Warm starts reduce runtime because the data is preloaded into RAM disk caches in the file system.

Program. One change is required to the GRANSIM version of the program to enable it to run under GUM. GUM processors don't inherit file handles from the main thread, and hence to permit them to simultaneously read files the program uses the 'unsafe' C-interface supported by Glasgow Haskell.

5.5 Measurements

Original Data Set. For these measurements the data is in 8 small tiles on the workstations, and in 4 large tiles on the SPARCserver. The runtimes of the program under GUM drop from 136.52 seconds on 1 workstation to 76.53 seconds on 2 workstations, and to 26.74 seconds on 8 workstations. The runtime under GUM on a single processor is 28% longer than the runtime for the optimised program because GUM imposes some additional overheads [17].

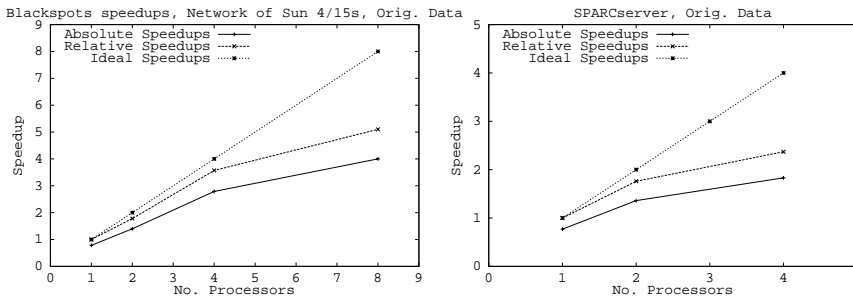


Fig. 4. Speedups of Blackspots on original data

Parallelism results are often reported as speedup graphs like Figure 4. The top line is ideal, or linear speedup. The second line is the *relative* speedup, i.e. compared to a single processor running the parallel program. The bottom line is the *absolute* or wall-clock speedup, i.e. compared to a single processor running the optimised sequential code.

These results are the first wall-clock speedups obtained for a GPH program solving a real problem using real data. On the SPARCserver the speedup is good at 2 processors, but poor thereafter. On the workstations the speedups are good up to 4 processors (3.6 relative speedup), but fall off thereafter. This indicates that the initial data set is too small to get good results on the larger parallel machine configurations.

Multiple Tiles per Processor. The data set can be increased by using larger tiles or by using more tiles. The latter approach is taken mainly because it makes good use of the dynamic load management supported by GUM: when a processor becomes idle it locates the next work item, in this case a tile. With many tiles each processor can be kept busy. Furthermore, Section 5.3 showed that many small tiles are more efficient than a few large tiles.

Figure 5 shows the speedups obtained partitioning 1.8Mb of data in 40 tiles: 32 small and 8 large. The larger data set gives an improved speedup on both architectures: 12 relative and 10 absolute on 16 workstations, and 2.8 relative and 2.2 absolute on the SPARCserver.

6 A Linear Equation Solver

In this section we discuss LinSolv, a parallel algorithm for finding an exact solution of a system of linear equations over integer values. It represents a typical application from the area of symbolic computation dealing with arbitrary precision integer values rather than floating point numbers. A more detailed discussion of this algorithm including a comparison with the pre-strategy code is given in [9].

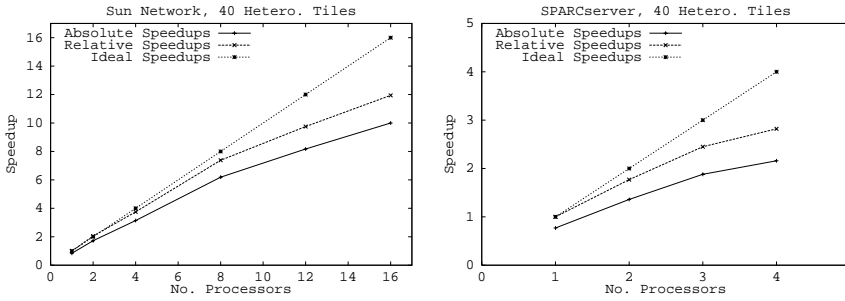


Fig. 5. Speedups of Blackspots on heterogeneous tiles

6.1 Sequential Algorithm

LinSolv uses an approach that is very common for computer algebra algorithms: a multiple homomorphic images approach [5]. The main idea of this approach is to solve a problem in a set of simpler domains, called homomorphic images, and then to reconstruct, or lift, the overall solution from the solutions in the individual domains.

In the case of the LinSolv algorithm the original domain is \mathbb{Z} , the set of all integer values, and the homomorphic images are the domains \mathbb{Z}_p , the set of integers modulo p with p being a prime number. The advantage of this approach becomes clear when the input numbers are very big and each prime number is small enough to fit into one machine word. In this case the basic arithmetic in the homomorphic images is ordinary fixed precision arithmetic with the results never exceeding one machine word. No additional cost for handling arbitrary precision integers has to be paid. Even in the sequential case the gain in efficiency by using cheap arithmetic outweighs the additional costs of combining the solutions. In the case of \mathbb{Z} as original domain the well-studied Chinese Remainder Algorithm (CRA) can be used in the combine step [6]. This overall structure of the algorithm is shown in Figure 6. Note that we use an algorithm that factors out the rational part $\frac{s}{t}$ in the result matrix so that the result vector x only contains integer values.

In order to describe the parallelism in this algorithm it is important to understand the structure of the main intermediate data structure, **xList**, an infinite list of the solutions in all homomorphic images. Each solution is a list with the following components: a prime number, which is the basis of the homomorphic image; the image of the determinant of the input matrix, which is needed to decide whether the result in this image can be used for constructing the overall result; and the result vector of solving the linear system of equations.

6.2 Parallelisation

The strategy in Figure 7 controls the parallelism over three variables in the algorithm: **noOfPrimes**, a guess of how many prime numbers are needed in order

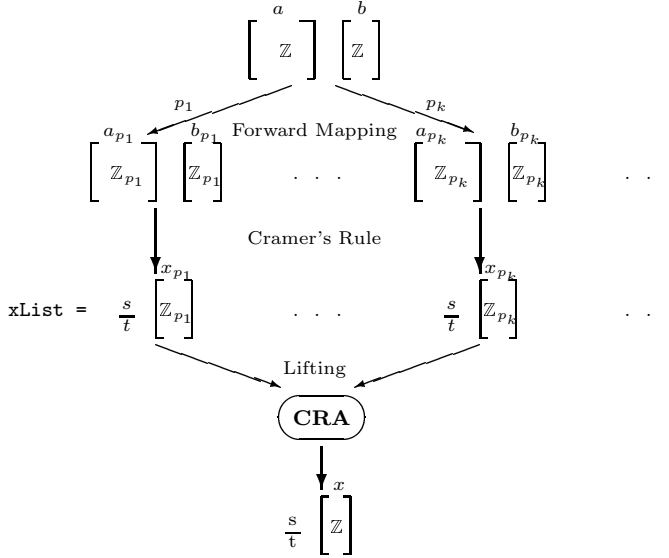


Fig. 6. Structure of the LinSolv algorithm

to construct the overall result; \mathbf{xList} , the infinite list of homomorphic solutions; and \mathbf{x} , the result vector in the original domain. Based on the result of `noOfPrimes` the strategy uses `parListN` to generate data parallelism over an initial segment of \mathbf{xList} . This is similar to the use of a static force length in the Alpha-Beta search algorithm. However, in this case the guess is conservative and will not generate redundant computation. Note that without controlling the parallelism over \mathbf{xList} the CRA will demand each solution sequentially, because it performs a list fold operation. The final strategy application `parList rnf x` specifies that all elements of the overall result should be combined in parallel.

```

rnf noOfPrimes                                     'seq'
parListN noOfPrimes par_sol_strat xList             'par'
parList rnf x
where
  par_sol_strat :: Strategy [Integer]
  par_sol_strat = \ (p:modDet:pmx) -> rnf modDet 'seq'
                                     if modDet /= 0
                                     then parList rnf pmx
                                     else ()

```

Fig. 7. Strategy for the parallel LinSolv algorithm

Using `parList` inside the `par_sol_strat` strategy causes each component of the result to be evaluated in parallel. However, it is necessary to check whether the determinant of the original matrix is zero in this homomorphic image to avoid redundant computation. In order to minimise data dependencies in the algorithm we do not already check the determinant when computing `noOfPrimes`. If some images cannot be used for constructing the result, the CRA will evaluate more results by demanding a so far unevaluated list element of `xList`.

6.3 Summary

In order to describe nested parallelism the higher-order nature of strategies is important. In the case of `LinSolv` an outer strategy defines the parallelism over `xList` with the strategy `par_sol_strat` as argument that defines the parallelism over the elements of this list. This demonstrates the use of data-oriented parallelism: the parallel behaviour is defined over the generated data structures rather than inside the functions that operate on the data structures. This approach, which exploits the non-strict semantics of data structures, allows the programmer to maintain a high degree of modularity when controlling the parallelism of the program. In the case of `LinSolv` it was sufficient to apply the strategy in Figure 7 to the top level function without changing the algorithm at all.

A direct comparison with a pre-strategy version of this program shows that the performance tuning of the parallel program is greatly facilitated by the separation between algorithmic and behavioural code. This is demonstrated by the development of three parallel versions of the code in [9]. In contrast to the strategic version, the pre-strategy version of the code combined the computation of the result with a specific dynamic behaviour suitable for parallelism, yielding rather opaque parallel code.

7 Discussion

This paper has described the construction of several medium-scale parallel functional programs, each program typifying a different application area. The construction has exposed several issues:

Real Application Parallelism. The accident blackspots application achieves good wall-clock speedup and acceptable scale-up on both a distributed-memory machine and a shared-memory machine. Both `LinSolv` and the Alpha-Beta search achieve good speedups on the simulator in realistic settings and their final versions demonstrated similar wall-clock speedups on a four processor shared-memory machine. The programs use a number of different parallelism paradigms, e.g. pipelining, divide-and-conquer and data-oriented parallelism, often combining several paradigms in a single program leading to irregular parallelism. From our experience it is too restrictive to support only certain paradigms without the possibility to combine and nest them. The regular use of higher-order strategies in this paper emphasises this aspect.

Language Issues. The clear separation of behavioural and algorithmic code was essential for the performance tuning of the algorithms. This is even more so in programs like Alpha-Beta search or LinSolv that make essential use of laziness. It was far easier to test about a dozen different variants of the strategic version of LinSolv than to change the code in various modules in order to achieve different parallel behaviour in the pre-strategy version.

The data-oriented parallelism used in the presented algorithms is in stark contrast to the control-oriented parallelisation typically used in imperative languages. The definition of parallelism over intermediate data structures increases the modularity of the parallel algorithm because the functions operating on the data structure can remain unchanged. A detailed discussion of data-oriented parallelism and a comparison between the strategic approach and imperative parallel programming can be found in [8]. As further work it would be interesting to investigate the use of strategies in a strict language or to use strategies as a parallel coordination language for components written in a strict language.

The definition of our strategies module evolved over the course of parallelising these programs. In particular, strategic function composition was added in order to facilitate the specification of data-oriented parallelism. This highlights the importance of using large applications for testing a programming technique like evaluation strategies.

Engineering Environment. Developing these programs has refined GRANSIM, GUM, and the profiling tools. For example, the design of the GRANSIM-Light setup providing idealised simulation was mainly motivated by our experiences in parallelising programs like LinSolv and the Accident Blackspots program. Considering that only a few systems for parallel functional programming have been engineered beyond prototype stage, it is particularly gratifying that the implementation technology is available on several architectures and handles programs as large as 47,000 lines. Our parallelisation methodology uses both sequential and parallel profilers, and the fact that GRANSIM is parameterisable is crucially important. The possibility to simulate an idealised machine as well as several different target architectures, and to run the parallel program in the same integrated environment greatly reduces the time needed to develop and tune a parallel program. It also allows the programmer to use the same visualisation tools for both simulation and parallel execution.

Although our visualisation tools have been essential in understanding the dynamic behaviour of these programs, the parallelisation process also demonstrated the need for better parallel profilers. In particular we would like to relate the active threads at any given time in a program execution to the strategy that created them. This observation was the main motivation for designing the parallel profiler GRANCC, which is currently under development [2].

References

1. K. Hammond, H-W. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *HPFC'95 — High Performance Functional Computing*, pp. 208–221, Denver, CO, Apr. 10–12, 1995.
2. K. Hammond, H-W. Loidl, and P.W. Trinder. Parallel Cost Centre Profiling. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Sep. 15–17, 1997.
3. P.H. Hartel, R.F.H. Hofman, K.G. Langendoen, H.L. Muller, W.G. Vree, and L.O. Hertzberger. A Toolkit for Parallel Functional Programming. *Concurrency — Practice and Experience*, 7(8):765–793, Dec. 1995.
4. R.J.M. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, Apr. 1989.
5. M. Lauer. *Computing by Homomorphic Images*, pp. 139–168. In *Computer Algebra — Symbolic and Algebraic Computation*. Springer-Verlag, 1982.
6. J. D. Lipson. Chinese Remainder and Interpolation Algorithms. In *SYMSAM*, pp. 372–391, 1971.
7. H-W. Loidl, R. Morgan, P.W. Trinder, S. Poria, C. Cooper, S.L. Peyton Jones, and R. Garigliano. Parallelising a Large Functional Program; Or: Keeping LOLITA Busy. In *IFL'97 — Intl. Workshop on the Implementation of Functional Languages*, Univ. of St. Andrews, Scotland, Sep. 10–12, 1997.
8. H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Dept. of Computing Science, Univ. of Glasgow, Mar. 1998.
9. H-W. Loidl. LinSolv: a Case Study in Strategic Parallelism. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Sep. 15–17, 1997.
10. E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, Jul. 1991.
11. A.P. Poulovassilis and C. Small. A Domain-Theoretic Approach to Logic and Functional Databases. In *VLDB'93 — Intl. Conf. on Very Large Databases*, pp. 415–426, 1993.
12. P. Roe. *Parallel Programming using Functional Languages*. PhD thesis, Dept. of Computing Science, Univ. of Glasgow, Feb. 1991.
13. C. Runciman and D. Wakeling. Profiling Parallel Functional Computations (without Parallel Machines). In *Glasgow Workshop on Functional Programming*, Workshops in Computing, pp. 236–251, Ayr, Scotland, Jul. 5–7, 1993. Springer-Verlag.
14. P.M. Sansom and S.L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, Mar. 1997.
15. P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), Jan. 1998.
16. P. Trinder, K. Hammond, H-W. Loidl, S.L. Peyton Jones, and J. Wu. A Case Study of Data-intensive Programs in Parallel Haskell. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Jul. 8–10, 1996.
17. P. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel implementation of Haskell. In *PLDI'96 — Programming Languages Design and Implementation*, pp. 79–88, Philadelphia, PA, May 1996.
18. J. Wu and L. Harbird. A Functional Database System for Road Accident Analysis. *Advances in Engineering Software*, 26(1):29–43, 1996.

Parallelising a Large Functional Program or: Keeping LOLITA Busy

Hans-Wolfgang Loidl¹, Richard Morgan², Phil Trinder³, Sanjay Poria²,
Chris Cooper², Simon Peyton Jones¹, and Roberto Garigiano²

¹ Department of Computing Science, University of Glasgow,
Glasgow, Scotland, U.K.

² Department of Computer Science, University of Durham
Durham, England, U.K.

³ The Computing Department, The Open University,
Milton Keynes, England, U.K.

Abstract. In this paper we report on the ongoing parallelisation of LOLITA, a natural language engineering system. Although LOLITA currently exhibits only modest parallelism, we believe that it is the largest parallel functional program ever, comprising more than 47,000 lines of Haskell. LOLITA has the following interesting features common to real world applications of lazy languages:

- the code was not specifically designed for parallelism;
- laziness is essential for efficiency in LOLITA;
- LOLITA interfaces to data structures outside the Haskell heap, using a foreign language interface;
- LOLITA was not written by those most closely involved in the parallelisation.

Our expectations in parallelising the program were to achieve moderate speedups with small changes in the code. To date speedups of up to 2.4 have been achieved for LOLITA running under a realistic simulation of our 4 processor shared-memory target machine. Most notably, the parallelism is achieved with a very small number of changes to, and without requiring an understanding of most of the application. On the Sun SPARCserver target machine wall-clock speedup is currently limited by physical memory availability.

1 Introduction

Despite the advantages of the functional programming model for parallel computation, there are few large non-strict parallel functional programs. We believe that this is due to the scarcity of robust parallel functional language implementations as well as the lacking support for high level languages describing some behavioural aspects of the program. We have recently constructed GUM, a parallel runtime system for Haskell, based on the Glasgow Haskell Compiler [10]. In order to facilitate parallel program design we have developed evaluation strategies, describing the dynamic behaviour of lazy parallel programs. GUM is portable, and available on both shared- and distributed-memory architectures, including

the CM5 [1], Sun SPARCServer shared-memory multiprocessor and networks of Suns and Alphas. It is freely available and has users and developers worldwide. GUM is also integrated with the GRANSIM parameterisable simulator and its visualisation tools [3]. The superset of Haskell used to program parallel machines is called Glasgow Parallel Haskell, GPH.

The LOLITA natural language engineering system [8] has been developed at Durham University over several years. It has not originally been written with a parallel execution of the code in mind. The team's interest in parallelism is partly as a means of reducing runtime, and partly also as a means to increase functionality within an acceptable response-time. The groups at Glasgow and Durham have cooperated to parallelise LOLITA, and we believe that the result is the largest non-strict parallel functional program ever constructed. Such a large program tests the scalability both of our parallel software engineering techniques and of the underlying implementation technology, i.e. the runtime system, simulator and the visualisation tools. It also makes excellent use of our integrated engineering environment for parallel program development and performance evaluation. We focus on this particular aspect in the accompanying paper [5].

The LOLITA system has several features that make it an interesting system to parallelise. First of all its size alone tests both our engineering environment as well as our methodology for developing parallel programs. Secondly, it has not been specifically designed with parallelism in mind. The declarative nature of Haskell, which leaves the evaluation order undefined, allows us to refine this order and to specify parallelism. Finally, laziness is essential for the efficiency of LOLITA, and therefore we have to preserve the degree of laziness specified in the sequential program.

The structure of the remainder of the paper is as follows. Section 2 briefly describes LOLITA with a special focus on those issues that are of special interest for the parallelisation. Section 3 gives a reference to a detailed discussion of GPH, and our new mechanism for introducing and controlling parallelism, evaluation strategies. Section 4 describes the parallelisation to date, and ongoing work. Section 5 discusses the lessons learnt from parallelising LOLITA and addresses runtime-system issues for parallel functional languages.

2 Introducing LOLITA

The LOLITA (Large-scale Object-based Linguistic Interactor Translator and Analyser) system is a state of the art natural language processing system, able to grammatically parse, semantically and pragmatically analyse, reason about, and answer queries on complex texts, such as articles from the financial pages of quality newspapers. In contrast to many existing systems LOLITA was designed as a general, domain-independent knowledge representation and reasoning system. Written substantially in Haskell, with some small components in C, it consists of over 47,000 lines of Haskell source code [2] (excluding comments).

The system has been under development at the Laboratory of Natural Language Engineering, University of Durham since 1986. The project currently involves a team of approximately 20 developers working simultaneously on different components of the system. In June 1993 the LOLITA system was demonstrated to the Royal Society in London. Research in the group follows a pragmatic approach to natural language processing; it is the production of a robust and useful working system that is of primary interest.

The team's interest in parallelism is partly as a means of reducing run-time, and partly also as a means to increase functionality within an acceptable response-time. The overall structure of the program bears some resemblance to that of a compiler, being formed from the following large stages:

- Morphology (combining symbols into tokens; similar to lexical analysis);
- Syntactic Parsing (similar to parsing in a compiler);
- Normalisation (to bring sentences into some kind of normal form);
- Semantic Analysis (compositional analysis of meaning);
- Pragmatic Analysis (using contextual information from previous sentences).

These stages form the core of LOLITA. Depending on how LOLITA is to be used, a final additional stage may perform a discourse analysis, the generation of templates or text (e.g. in a translation system), or it may perform inference on the text to answer queries. This modular structure of LOLITA is shown in Figure 1.

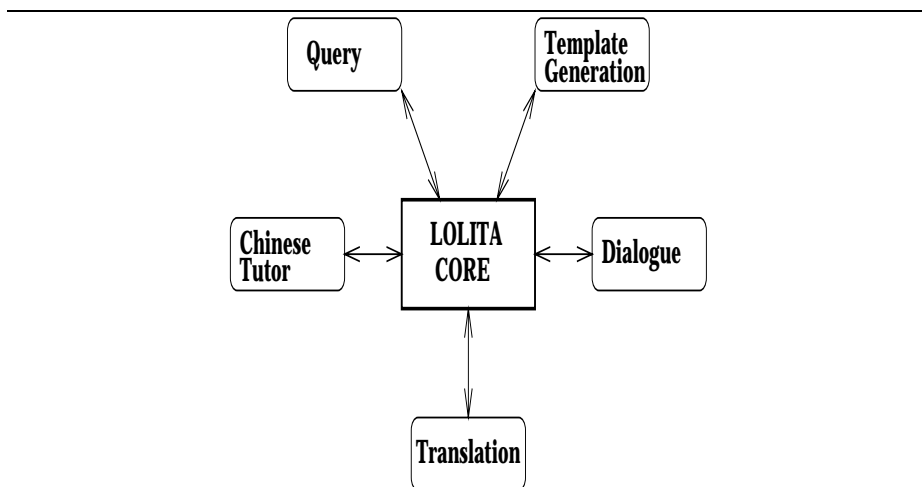


Fig. 1. Applications built around the LOLITA core

Central to LOLITA's flexibility is the semantic network, called SemNet, a graph based knowledge representation used in the core of LOLITA. In SemNet

concepts and relationships are represented by nodes and arcs respectively, with knowledge being elicited by graph traversal. SemNet has been designed for an efficient form of inference, namely inheritance [6], and additionally supports other forms of reasoning such as analogy [7]. The task of the analysis stages is to transform the possibly ambiguous input into a piece of SemNet. Application-dependent backend stages can then extract pieces of the SemNet and present it in the required form. Currently, SemNet comprises approximately 100,000 nodes or 12Mb.

LOLITA is a large system, and a detailed description of all its features is beyond the scope of this paper. The interested reader is referred to [8]. However, the following features are important because they have a direct impact on our introduction of parallelism.

- LOLITA has already been carefully tuned, using time and heap profiling, for efficient sequential execution. We are parallelising an efficient program.
- Preserving laziness is crucial at several points in LOLITA. For example, in both parsing and semantic analysis the program selects the “best” parse- and semantic-trees from a forest. The cost of evaluating a single additional tree is prohibitively expensive. It is important that the parallel program is no more strict than the sequential program at these crucial points.
- The parser is substantially written in a “foreign language”, namely C. As a result we must control parallelism in both languages, and the consequences of this are discussed in Section 5.
- LOLITA represents the knowledge used in the semantic analysis, SemNet, as a large (12Mb) C data structure. SemNet is stored outside the Haskell heap, and read using C function calls. The issues raised by making this structure available on each node of a multiprocessor are discussed in Section 4.6 and Section 5.

3 GPH and Evaluation Strategies

The GPH parallel language and the associated programming environment and methodology are described in Section 2 of a companion paper in these proceedings, namely *Engineering Large Parallel Functional Programs* [5]. The reader is referred to this section now as an understanding of these concepts is required to appreciate how parallelism is introduced in LOLITA. A complete description and discussion of evaluation strategies can be found in [9].

4 Parallelising LOLITA

4.1 Sequential Profiling

As a preparation for parallelising such a large program we performed sequential profiling of the code. This did not reveal a particular hotspot in the program although the syntactic parsing stage is the biggest component in the top level

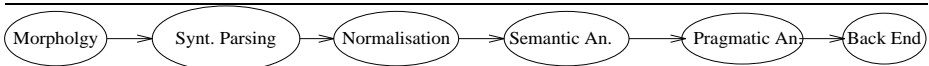


Fig. 2. Overall pipeline structure of LOLITA

structure with about 20% of the execution time. This stage makes heavy use of C-functions, called from within Haskell, to optimise the time consuming parsing process. These foreign language calls complicate the parallelisation of the parsing stage. The Haskell part of the parsing, however, can be parallelised without major recoding.

4.2 Top Level Pipeline

Without a dominating hotspot in the sequential execution of the program a pipeline approach is a promising way to achieve enough parallelism for a 4 processor shared-memory machine like the Sun SPARCServer we are aiming at. The structure of a pipeline parallel version is shown in Figure 2. Each stage discussed in Section 2 is executed by a separate thread, which are linked to form a pipeline. In contrast to classical pipelines, which require a large input set to achieve good parallelism, the non-strict semantics of Haskell makes it possible that several stages of the pipeline work on the same unit of input data in parallel. In particular it is possible for later stages to process parts of the generated data structure while earlier stages are still generating other parts of this data structure.

```

infixl 6  $\$/$ ,  $\$//$       -- strategic function application
infixl 9  $\cdot/$ ,  $\cdot//$       -- strategic function composition

( $\$/$ ), ( $\$//$ ) :: (a -> b) -> Strategy a -> a -> b
( $\cdot/$ ), ( $\cdot//$ ) :: (b -> c) -> Strategy b -> (a -> b) -> (a -> c)

( $\$/$ ) f s x = f x 'demanding' s x
( $\$//$ ) f s x = f x 'sparkling' s x

( $\cdot/$ ) f s g = \ x -> let gx = g x
                      in f gx 'demanding' s gx
( $\cdot//$ ) f s g = \ x -> let gx = g x
                      in f gx 'sparkling' s gx
  
```

Fig. 3. Definition of strategic function application

The key step in constructing the pipeline is to define strategies on the complex data structures (e.g. parse trees) passed between stages. In order to avoid naming the data structures we use a *strategic function application* operator, which comes in a sequential $\$/$ and in a parallel $\$//$ variant. An expression of

```

wholeTextAnalysis opts inp global =
  result
  where
    -- (1) Morphology
    (g2, sgml) = prepareSGML inp global
    sentences = selectEntitiesToAnalyse global sgml

    -- (2) Parsing
    rawParseForest = map (heuristic_parse global) sentences 'using' parList rnf

    -- (3)-(5) Analysis
    anlys = stateMap_TimeOut (parse2prag opts) rawParseForest global2

    -- (6) Back End
    result = back_end anlys opts

-- Pick the parse tree with the best score from the results of
-- the semantic and pragmatic analysis. This is done speculatively!

parse2prag opts parse_forest global =
  pickBestAnalysis global $|| evalScores $
  take (getParsesToAnalyse global) $
  map analyse parse_forest
  where
    analyse pt = mergePragSentences opts $ evalAnalysis
    evalAnalysis = stateMap_TimeOut analyseSemPrag pt global
    evalScores = parList (parPair rwhnf (parTriple rnf rwhnf rwhnf))

-- Pipeline the semantic and pragmatic analyses
analyseSemPrag parse global =
  prag_transform $|| rnf $
  pragm $|| rnf $
  sem_transform $|| rnf $
  sem (g,[]) $|| rnf $
  addTextrefs global $| rwhnf $
  subtrTrace global parse

back_end inp opts =
  mkWholeTextAnalysis $| parTriple rwhnf (parList rwhnf) rwhnf $
  optQueryResponse opts $|| rnf $
  traceSemWhole $|| rnf $
  addTitleTextrefs $|| rnf $
  unifyBySurfaceString $|| rnf $
  storeCategoriseInf $|| rnf $
  unifySameEvents opts $| parPair rwhnf (parList (parPair rwhnf rwhnf)) $
  unpackTrees $| parPair rwhnf (parList rwhnf) $
  inp

```

Fig. 4. The top level function of LOLITA

the form $f \ \$|| \ s \ \$ \ x$ applies the strategy s to the input value x and in parallel applies the function f to x . Thus, the strategy is mainly used to specify the evaluation degree whereas the $\$||$ operator specifies the pipeline parallelism. The $\$$ operator in Haskell denotes (explicit) function application, i.e. $f \ \$ \ x = f \ x$, and the $.$ operator denotes function composition, i.e. $(f \ . \ g) \ x = f \ (g \ x)$. The definitions of the new combinators are given in Figure 3.

These new combinators are particularly useful for defining data-oriented parallelism because they can be used to define parallelism on the intermediate data structures, via the strategy parameter. This approach simplifies the top-down parallelisation of very large systems, since it is possible to define the parts of the data structure that should be evaluated in parallel without considering the algorithms that produce the data structures. Note that the data-oriented parallelism

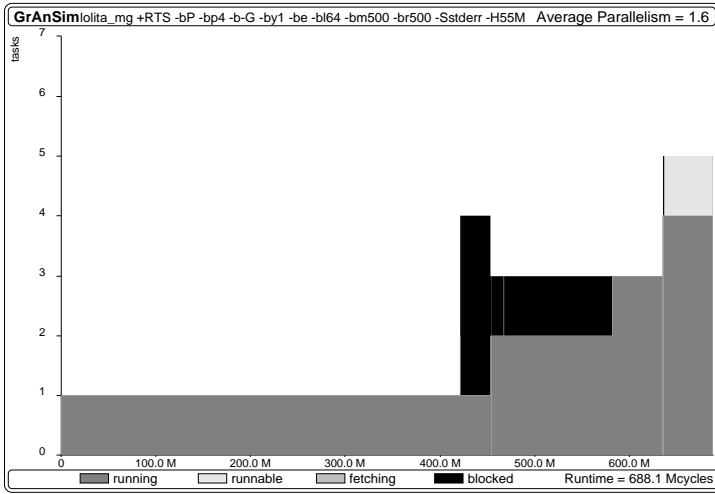


Fig. 5. Activity profile of LOLITA with pipeline parallelism

on the data structures can be defined independently from the control-oriented parallelism of the pipeline itself.

The code of the top level function `wholeTextAnalysis` in Figure 4 clearly shows how the algorithm is separated from the dynamic behaviour in each stage by using the `$||` operator. In a first parallel version we achieved the same separation with an explicit pipeline strategy. However, this required to name every intermediate value in the pipeline which did not reflect the structure of the code in quite the same way. This experience was our main motivation for developing the strategic function application operator.

Note that this code uses a `parList` strategy in the definition of the variable `rawParseForest` in the parsing stage to describe data parallelism over the whole input by processing sentences in the input text in parallel. At the moment we cannot use this source of parallelism because the C code in this stage is not re-entrant. Changing the C code to exploit this form of parallelism is ongoing work.

The semantic and pragmatic analysis stages are wrapped into a timeout function in order to guarantee a worst case response time of the system. This indicates that these stages can be very computationally intensive. Therefore, both analyses are kept rather simple in the sequential system. By providing the strategy `evalScores`, in `parse2prag`, speculative parallelism is defined, which allows the system to perform a more sophisticated analysis by examining several possible parse trees. The goal of this strategy is therefore to improve the quality of the result. Section 4.4 discusses this issue in more detail. In general, it would be very desirable to improve the quality of semantic and pragmatic analysis

```

mergeStrategy :: (NFData a, NFData b) =>
  (ParseForest, FeatureForests) -> Span -> MergeStrategy a b

mergeStrategy (pf, ff) span
| totalSpan == 0          = MStrat serialMerge
| percentSpanned >= minSpan = MStrat parallelMerge
| otherwise               = MStrat serialMerge
  where
    percentSpanned = (span * 100) `div` totalSpan
    totalSpan = forestSpan pf
    minSpan = getParsingParPercent (forestGlobal pf)

parallelMerge :: (NFData a, NFData b) =>
  [(a,b)] -> [(a,b)] -> Strategy [(a,b)]
parallelMerge as bs _
  = fstPairFstList bs 'par'
  = fstPairFstList as 'seq'
  ()

fstPairFstList :: (NFData a, NFData b) => Strategy [(a,b)]
fstPairFstList = seqListN 1 (seqPair rwhnf r0)

serialMerge :: (NFData a, NFData b) =>
  [(a,b)] -> [(a,b)] -> Strategy [(a,b)]
serialMerge as bs
  = r0

```

Fig. 6. A granularity control strategy used in the parsing stage

in the system. Parallelism inside these stages could be used to maintain good performance despite the increased complexity of the system.

On first glance the activity profile in Figure 5 seems to exhibit very poor parallelism. However, part of the sequential front end of this algorithm is the C part of the parsing, which cannot be parallelised in this fashion because of its strictness. Furthermore, the consistent parallelism towards the end of the compilation yields a fairly good utilisation for a four processor machine. Considering the minimal changes in the code to achieve parallelism this is a reasonable first step in the parallelisation process.

4.3 Parallel Parsing

One major source of parallelism in the time consuming syntactic parsing phase is the merging of possible parse trees in order to build a parse tree for a whole sentence. One complication in the parsing of natural languages is their ambiguity. Because of this ambiguity the parsing stage produces not just one but a list of possible parse trees. Internally, however, the result is represented as a single tree, which at some points contains alternatives (“or-nodes”) representing different possible parses of the subtrees. A lazy function is used to convert this single tree into a list of possible parse trees. In each or-node the parser, which returns a list of parse trees, must merge the lists of parse trees produced by the recursive calls. In merging these lists the possible parse trees have to be sorted based on some simple syntactic criteria representing the likelihood of a parse, and the laziness

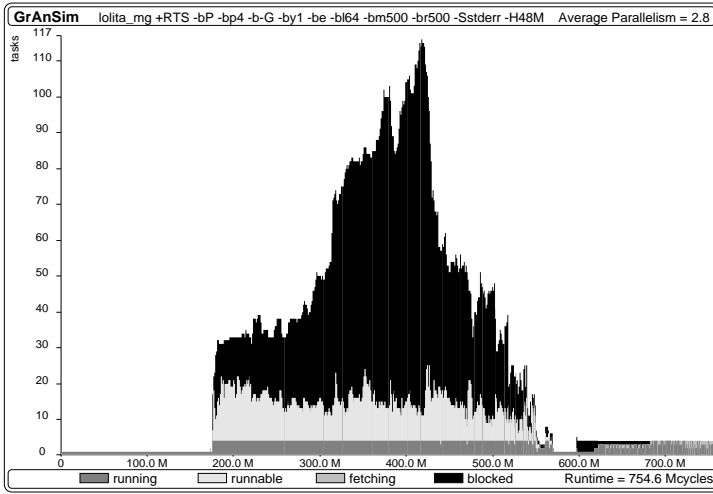


Fig. 7. Activity profile of LOLITA with a span threshold of 50%

of Haskell is crucial. In order to produce one parse tree in an or-node it is only necessary to evaluate the first element in the lists produced by all alternatives.

From a parallelism point of view this behaviour explains why we cannot force the evaluation of parts of the parse forest without risking to introduce a high degree of redundant work. Furthermore, within the parsing process the merging of lists triggers the evaluation of sublists, in particular the evaluation of the quality of possible parses. Although the merging itself is very cheap it triggers work that can be usefully done in parallel.

In order to improve the granularity of the threads produced by the parallel tree traversal in the parsing stage, we apply a thresholding strategy shown in Figure 6 to the “span” in the tree. The span value, which is attached to each node in the tree, specifies the number of leaves in the current subtree. The threshold for generating a parallel process in order to merge all possible subtrees is specified as a percentage of leaves that can be reached from the current node, and this percentage is part of the global system environment. Checking the threshold is very cheap because it only involves the comparison of the `span` argument, as a percentage, with a system parameter assigned to `minSpan`.

The two parallel calls to `fstPairFstList` in `parallelMerge` define parallelism in this stage. Only the first element of the pair is evaluated because it contains the value determining the quality of the resulting parse tree. Thus, the `fstPairFstList` strategy specifies an evaluation degree that is sufficient to se-

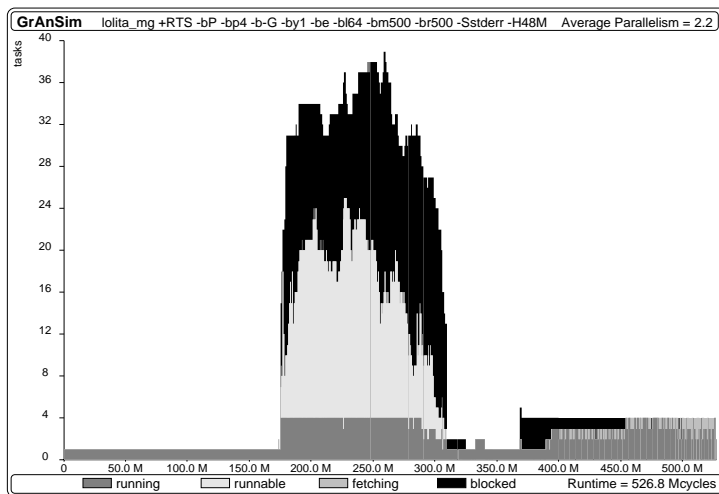


Fig. 8. Activity profile of LOLITA with a span threshold of 90%

lect the tree to return as the result of the syntactic parsing stage but without evaluating the tree itself more than necessary.

One strength of strategies is their reusability for different algorithmic code that has the same dynamic behaviour. We were able to exploit this feature with `mergeStrategy` in Figure 6 by applying the same polymorphic thresholding strategy to two lists of different types within the syntactic parsing stage. This reuse is highlighted by the parameterisation of the `MergeStrategy` datatype over the two possible types in the list.

We have performed a series of measurements under GRANSIM in order to determine the best value for the span in this strategy. We have used a setup that models the 4 processor shared-memory Sun SPARCServer available at Durham. Time is always measured in machine cycles.

Figure 7 shows the activity profile for a span threshold of 50%. During the syntactic parsing stage we achieve a quite good utilisation of the system. However, almost 100 blocked threads and a high number of runnable threads are generated, too. These impose significant runtime overhead in the system. The histogram of threads sorted by thread size in the granularity profile at the left hand side of Figure 9 reveals that most of the threads are very fine-grained: 3,422 of the 5,122 threads (67%) are shorter than 2,000 cycles. This leads to a bad ratio of computation versus parallelism overhead.

In comparison, when increasing the span threshold to 90%, in Figure 8, the number of blocked and runnable threads is reduced significantly (at most 36),

and the number of small threads drops drastically, as shown at the right hand side of Figure 9. Now, only 67 of the 165 threads are shorter than 2,000 cycles (40%). Corresponding to this drop in the total number of threads, especially fine-grained threads, the runtime drops from 754.6 Mcycles in the previous version to 526.8 Mcycles in this version.

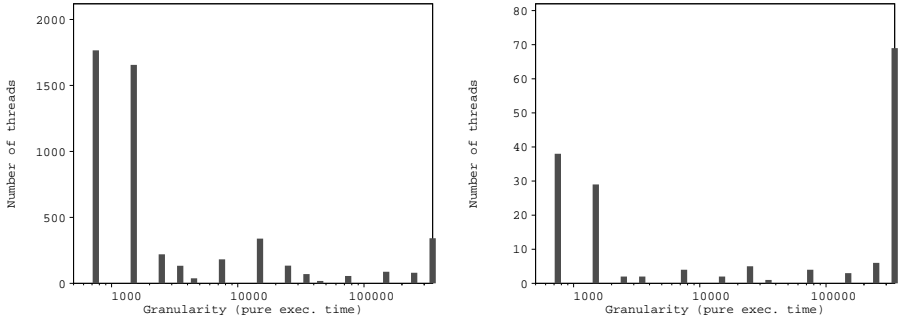


Fig. 9. Granularity profiles of LOLITA with a span thresholds of 50% and 90%

As a result of these measurements and considering the low amount of parallelism that is required to fully utilise the 4 processor shared-memory machine, we use span thresholds around 90% for GUM executions of LOLITA.

4.4 Parallel Semantic Analysis

Another source of parallelism can be used to improve the quality of the analysis by applying the semantic and pragmatic analyses in a data-parallel fashion on different possible parse trees for the same sentence. Because of the complexity of these analyses the sequential system always picks the first parse tree, which may cause the analysis to fail, although it would succeed for a different parse tree. In this case the system cannot produce a result for the current sentence in a sequential setup. Therefore, parallelism in this stage would not reduce runtime of the system, but might improve the quality of the result.

This additional data parallelism is defined by the strategy `evalScores` in the function `parse2prag` (see Figure 4). The parse forest `rawParseForest` contains all possible parses of a sentence. The semantic and pragmatic analyses are then applied to a predefined number (specified in the global system environment `global`) of these parses. The data parallel strategy `evalScores` is applied to the list of these results and demands only the score of each analysis (the first element in the triple) in order to avoid unnecessary computation at this stage.

This score is used in `pickBestAnalysis` to decide which of the parses to choose as the result of the whole text analysis.

So far we have not systematically measured the improvements in the quality of the result by analysing several possible parse trees. However, considering that about 70% of all sentences that are analysed have several possible parse trees, the possibility to analyse several of them without large additional costs is very attractive from a natural language engineering point of view.

4.5 Overall Parallel Structure

Figure 10 summarises the overall parallel structure arising when all of the sources of parallelism described above are used. The syntactic parsing stage is internally parallelised using the granularity control strategy shown in Figure 6. Note that the analyses may add nodes to the semantic net. This creates an additional dependence between different instances of the analysis (indicated as vertical arcs). Lazy evaluation ensures that this does not completely sequentialise the analyses, and it could be turned off for parallel queries to the system.

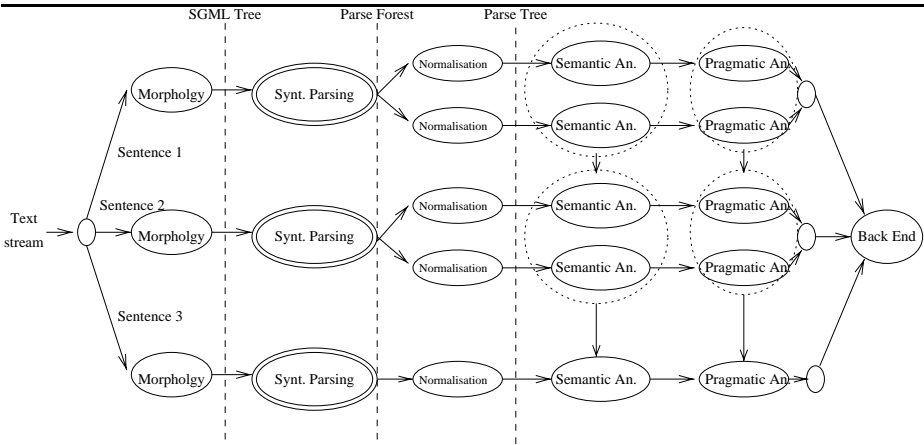


Fig. 10. Detailed structure of LOLITA

It should be emphasised that specifying the strategies that describe this parallel behaviour entailed understanding and modifying only two of about three hundred modules in LOLITA and three of the thirty six functions in that module. So far, the only module we have parallelised is the syntactic parsing stage. In fact, the most tedious part of the code changes was adding instances of `NFData` for intermediate data structures, which are spread over several dozen modules. However, in the meantime this process has been partially automated [11].

If it proves necessary to expose more parallelism we could parallelise other sub-algorithms, which also contain significant sources of parallelism. For example

in the inference process, which tries to extract information from the semantic net, standard graph algorithms such as computing the shortest distance between nodes are used. Parallel versions of these graph algorithms could increase the performance further.

4.6 Sun SPARCServer Implementation

We are currently tuning the performance of LOLITA on the Sun SPARCServer. A realistic simulation showed an average parallelism between 2.5 and 3.1, using just the pipeline parallelism and parallel parsing. The actual speedup, however, does not exceed 2.4. Our measurements with varying span values indicate that this is partly caused by fine-grained parallelism in the parsing stage.

Since LOLITA was originally written without any consideration for parallel execution and contains a sequential front end of about 10–15% (the C part of the syntactic parsing stage), we are pleased with this amount of parallelism. In particular the gain for a set of rather small changes is quite remarkable.

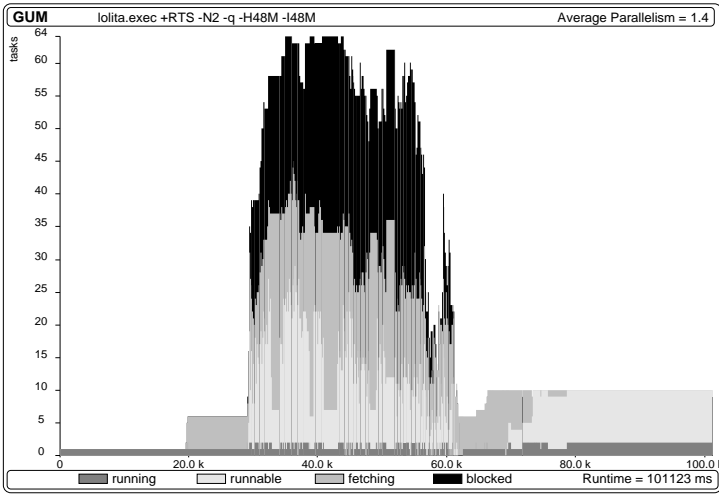


Fig. 11. Activity profile of LOLITA run under GUM with 2 processors

However, the wall-clock speedups obtained to date do not quite match the simulation results. As shown in Figure 11 a two processors execution on small inputs achieves an average parallelism of 1.4. We use a high span value to bound the amount of parallelism in the parsing phase. This also bounds the total heap

residency in the system, which proves to be very important. With more processors the available physical memory is insufficient and heavy swapping causes a drastic degradation in performance. The reason for this is that GUM, which is designed to support distributed-memory architectures uniformly, loads a copy of the entire code, and a separate local heap, onto each processor. LOLITA is a very large program, incorporating large static data segments (totalling 16Mb), and requires 100Mb of virtual memory in total in its sequential incarnation. Clearly a great deal of this data could be shared, an opportunity we are exploring. We are also making the C parsing functions re-entrant which will allow the analysis to be performed in a data-parallel fashion over a set of input sentences.

The multiprocessor GUM activity profiles can only be loosely compared with the GRANSIM profiles, e.g. Figure 8. However the shapes of the curves are similar, except that the GUM activity profile shows a larger degree of fetching. This is probably caused by the rather expensive but generic PVM communication routines used by GUM. In contrast, GRANSIM measures mainly the hardware costs for performing communication. Together with the fine granularity of the generated threads this increased overhead leads to a significantly smaller utilisation in the parsing stage. However, the later pipeline stages in the computation are still an effective source of parallelism.

5 Discussion

At the current stage of parallelising LOLITA we use top level pipeline parallelism as well as divide-and-conquer parallelism in the most expensive stage of the pipeline. Additional speculative parallelism can be used in the analysis stage in order to improve the quality of the result. We are currently working on changes in the C code to provide global data parallelism over the whole program (the changes in the Haskell code are in place already).

The current parallel version of LOLITA achieves a parallelism between 2.5 and 3.1 under GRANSIM emulating a 4-processor shared-memory machine. The corresponding speedup, however, does not exceed 2.4. This is partly due to overhead caused by very fine-grained parallelism and partly due to strategies that perform speculative computations (although we avoided speculation on potentially expensive components). The thresholding strategy in the parsing stage proved to be very effective in increasing the average granularity of the generated threads.

We have yet to obtain significant wall-clock speedups on our target machine. This, however, is not due to a lack of parallelism but due to the very high memory consumption of the application, which exceeds the available main memory in the current setting. We also hope to obtain further parallelism by making the C parsing code re-entrant, allowing us to exploit data parallelism by analysing sentences in parallel.

Parallelising LOLITA has taught us a great deal about large-scale parallel functional programming. The most intriguing aspect is that the parallelism is achieved using a very small number of changes to the Haskell parts of the

application. We have been able to use a top-down approach of the parallelisation to an extent, which would be very difficult in a strict language. All of the parallelism has been specified by evaluation strategies acting on the data structures passed between modules. As a result, the parallelism has been introduced without changing, and indeed without understanding most of the program. This abstraction is crucial when working on an application of this size. For example, introducing top-level parallelism entailed changing just one out of around three hundred modules. Considering this small amount of code changes in a large application, the fact that it was not designed to be executed in parallel, and the presence of inherently sequential foreign language calls, we are pleased with the achieved degree of parallelism.

Our experiences with LOLITA and several other medium-scale programs are being distilled into some guidelines for engineering large parallel functional programs [5]. Another concrete outcome of parallelising LOLITA has been the introduction of strategic function application to support pipeline parallelism. This extension to our strategy module has in the meantime proven useful for several other parallel programs like the parallelising compiler Naira [4].

The parallelisation of LOLITA pointed out the importance of several runtime-system issues. The choice of one fixed heap size for each processor proved to be too inflexible and we had to slightly change the runtime-system. In a shared-memory setting we would like to have heaps that can grow dynamically if necessary. In general either program annotations or runtime-system options for tuning the data-locality of the program would be useful. Uniform access to an external data-structure from every processor is important because the semantic net is constructed outside the Haskell heap. For a distributed memory usage of such data-structures the communication mechanism in GUM would have to treat them specially. In the current version, the presence of such a conceptually persistent data structure required recoding of the C routines for reading and sharing this data structure. In fact, recoding just this part of the C code required more changes than the actual parallelisation of the Haskell program itself. General support of persistent data in a distributed setting would simplify the program and its parallelisation considerably.

References

1. K. Davis. MPP Parallel Haskell. In *IFL'96 — Intl. Workshop on the Implementation of Functional Languages*, pp. 49–54, Bad Godesberg, Germany, Sep. 1996. Draft Proceedings.
2. A. Gill and P. Wadler. Real World Applications of Functional Programs. <URL:<http://www.dcs.gla.ac.uk/fp/realworld.html>>
3. K. Hammond, H-W. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *HPFC'95 — High Performance Functional Computing*, pp. 208–221, Denver, CO, Apr. 10–12, 1995. <URL:<http://www.dcs.st-and.ac.uk/~kh/papers/hpfc95/hpfc95.html>>
4. S. Junaidu. NAIRA: A Parallel2 Haskell Compiler. In *IFL'97 — Intl. Workshop on the Implementation of Functional Languages*, Univ. of St. Andrews, Scotland, Sep. 10–12, 1997.

5. H-W. Loidl and P.W. Trinder. Engineering Large Parallel Functional Programs. In *IFL'97 — Intl. Workshop on the Implementation of Functional Languages*, Univ. of St. Andrews, Scotland, Sep. 10–12, 1997.
6. D. Long and R. Garigiano. Inheritance hierarchies. Technical Report 4/88, Dept. of Computer Science, Univ. of Durham, 1988.
7. D. Long and R. Garigiano. *Reasoning by Analogy and Causality: A model and application*. Artificial Intelligence. Ellis Horwood, 1994.
8. R.G. Morgan, M.H. Smith, and S. Short. Translation by Meaning and Style in Lolita. In *Intl. BCS Conf. — Machine Translation Ten Years On*, Cranfield University, Nov. 1994.
9. P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), Jan. 1998.
10. P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel implementation of Haskell. In *PLDI'96 — Programming Languages Design and Implementation*, pp. 79–88, Philadelphia, PA, May 1996.
11. N. Winstanley. A Type-Sensitive Preprocessor for Haskell. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Sep. 15–17, 1997.

Naira: A Parallel² Haskell Compiler

Sahalu Junaidu*, Antony Davie, and Kevin Hammond

Division of Computer Science, University of St. Andrews
{sahl,ad,kh}@dcs.st-and.ac.uk

Abstract. Naira is a compiler for a parallel dialect of Haskell, compiling to a graph-reducing parallel abstract machine with a strong dataflow influence. Unusually (perhaps even uniquely), Naira has itself been parallelised using state-of-the-art tools developed at Glasgow and St Andrews Universities. Thus Naira is a parallel, parallelising compiler in one. This paper reports initial performance results that have been obtained using the GranSim simulator, both for the top-level pipeline and for individual compilation stages. We show that a modest but useful degree of parallelism can be achieved even for a distributed-memory machine. The simulation results have been verified on a network of distributed workstations using the GUM parallel implementation of Haskell.

1 Introduction

The Naira compiler was written to explore the problems of parallelising a modern functional language compiler [7]. It compiles from a substantial subset of Haskell [4] to a RISC-like target language that has been extended with special parallel constructs [10]. The front end of the compiler comprises about 5K lines of Haskell code organised in 18 modules.

This paper explores the process of parallelising this compiler using state-of-the-art profiling tools that were developed at Glasgow and St Andrews Universities [3]. Our initial results are promising, indicating that acceptable speedups can be achieved within individual compiler passes, notably the type inference pass, and these results are confirmed by measurements on the GUM [17] system. There is, however, a large sequential component caused by file I/O and parsing which limits the overall speedup that can be obtained.

The rest of this paper is structured as follows. Sections 2 and 3 respectively describe the parallelisation and analysis of the top-level pipeline and the individual passes. Section 4 describes related work. Finally Section 5 concludes.

2 The Top-Level Pipeline

We use a top-down parallelisation methodology, as in [18], starting with the top-level pipeline, then proceeding to parallelise successive pipeline stages. We

* Supported by the Federal Government of Nigeria under Federal Scholarship Scheme. I also acknowledge the support of Islamic Relief, Makkah, Saudi Arabia.

concentrate on parallelising four main compiler passes — the pattern matcher, lambda lifter, type checker, and the optimiser. The parallelisation proceeded in a data-oriented fashion by annotating the intermediate data structures used to communicate results between the passes. We have experimented with lists and binary trees to represent the intermediate structures.

2.1 Structure of the Top-Level Pipeline

The overall top-level pipeline structure of the compiler is as depicted in Figure 1. The first, analysis, pass consists of the lexical analyser and the parser. The next four passes implement the pattern matching compiler, the lambda lifter, the type checker and the intermediate language optimiser respectively. The detailed organisation and implementation of these passes is described elsewhere [7].

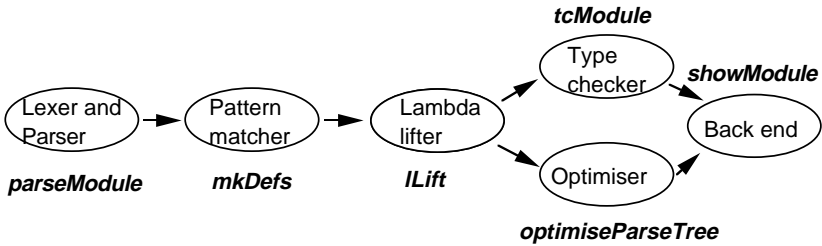


Fig. 1. The Structure of the Top-Level Pipeline

The two-way split after the lambda lifting pass indicates that the result of the lambda lifter can be piped simultaneously to both the type checker and the optimiser and that these latter two phases can proceed in parallel combining their results, using *showModule*, to produce the intermediate code.

2.2 Parallelising the Top-Level Pipeline: Version 1—Direct

Following the top-down parallelisation methodology outlined by Trinder *et al* [18], we start by parallelising the top-level pipeline of Naira before delving into the parallelisation of its main phases.

Figure 2 shows the function, *analyseModule*, that implements this top-level pipeline. It is called immediately following symbol table construction, and forwards its arguments to each compiler pass as appropriate. The underlined definition, *strat* is used to specify the parallel behaviour of *analyseModule* and is the only code that need to be added to ensure this parallelisation. This evaluation strategy, *strat*, sparks five parallel tasks using *par* (*c* ‘*par*’ *r* creates a task to evaluate *c*, then continues executing *r*), one for each of the pipeline phases shown in Figure 1. The *parList* strategy applies its first argument (the *rnf* strategy

```

analyseModule fileName modName imports exports symbTabs defs =
    result 'using' strat

where
  (stPM,stTE,stOpt) = symbTabs
  (dats,syns,funs)   = defs
  (aInfo,tInfo,impVals) = imports
  pMatchedDefs = mkDefs fileName stPM funs
  liftedDefs   = lLift fileName stPM pMatchedDefs
  typeList     = tcModule fileName stTE exports tInfo syns liftedDefs
  intermCode   = optimiseParseTree fileName exports stOpt aInfo liftedDefs
  result       = showModule modName impVals dats
                exports (intermCode,typeList)

  strat res     = parForceList funs      'par'
                  parForceList pMatchedDefs 'par'
                  parForceList liftedDefs  'par'
                  parForceList typeList    'par'
                  parForceList intermCode  'par'
                  ()
parForceList = parList rnf

```

Fig. 2. The Top-Level Compiler Function: **analyseModule**

in this case, which reduces its argument to normal form) to each element of its second (list) argument in parallel thereby creating new tasks to reduce each element of the list to normal form in parallel. Therefore, **parForceList** forces the parallel evaluation to *normal form* of elements of a list.

A disadvantage of using strategies in this form (with the **using** combinator) is that every intermediate value must be named. There is an efficiency issue associated with this (see Section 2.4) and we give an alternative parallelisation using a parallel application combinator (**\$||**) (see [18]) in the next section.

2.3 Parallelising the Top-Level Pipeline: Version 2—Using (**\$||**)

With the aid of the parallel combinator (**\$||**), the code for **analyseModule** can be rewritten more concisely, but perhaps less intuitively, as shown in Figure 3. The combinator (**\$||**) is a higher-order function that takes a function **f**, a strategy **s**, and a value **x** and applies the strategy **s** to **x** in parallel with the evaluation of **f x**.

As in Figure 2, the added behavioural code is underlined in Figure 3. Notice that in both these figures, there is a clear separation between algorithmic code and behavioural code.


```

analyseModule fileName modName imports exports symbTabs defs =
  showModule modName impVals exports $||
    parPair parForceList parForceList $
    fork (optimiseParseTree fileName exports stOpt aInfo,
          tcModule fileName stTE exports tInfo syms) $|| parForceList $
    lLift fileName stPM $|| parForceList $
    mkDefs fileName stPM $|| parForceList $ funs
where (stPM,stTE,stOpt) = symbTabs
      (dats,syms,funs) = defs
      (aInfo,tInfo,impVals) = imports
fork (f, g) inp = (f inp, g inp)
($||) :: (a → b) → Strategy a → a → b
f $|| s = \ x → s x 'par' f x

```

Fig. 3. analyseModule rewritten using Pipeline Strategies

2.4 Parallelising the Top-Level Pipeline: Simulated Results

In order to assess the compiler, we experimented with two different simulated machine configurations using the GrAnSim simulator: a low-latency shared-memory configuration, and a medium-latency distributed-memory configuration [7]. We also (see Section 2.9) measured the compiler on a real parallel machine. Both configurations were for 32-processor machines. Our test input comprised the 18 source modules for the Naira compiler itself. Speedup and average parallelism results are shown in Table 1.

<i>Module</i>	Avg.	Speedup	<i>Module</i>	Avg.	Speedup
	Par.			Par.	
MyPrelude	4.0 (3.0)	3.85 (2.91)	MatchUtils	3.4 (2.6)	3.32 (2.55)
DataTypes	4.3 (3.0)	4.17 (2.90)	Matcher	2.7 (2.1)	2.58 (2.06)
PrintUtils	1.5 (1.5)	1.44 (1.43)	LambdaUtils	2.1 (2.0)	1.98 (1.87)
Printer	2.1 (1.3)	2.08 (1.30)	LambdaLift	2.4 (2.1)	2.36 (2.07)
Tables	3.6 (2.7)	3.49 (2.59)	TCheckUtils	3.5 (3.2)	3.46 (3.19)
LexUtils	2.9 (1.9)	2.79 (1.83)	TChecker	1.8 (1.8)	1.76 (1.75)
Lexer	1.6 (1.5)	1.50 (1.45)	OptmiseUtils	1.5 (1.4)	1.48 (1.43)
SyntaxUtils	3.2 (2.1)	3.21 (2.09)	Optimiser	1.1 (1.1)	1.07 (1.06)
Syntax	1.3 (1.3)	1.24 (1.23)	Main	1.7 (1.7)	1.64 (1.63)

Table 1. Top-level Pipeline: Speedup and Parallelism for a Simulated Shared-memory Machine (Distributed-memory Results in Brackets).

We found from our experiments that as well as being less concise, the first version of `analyseModule` is also less efficient than the second version for most of

our sample inputs. For our 18 sample input modules, we found that the second version was up to 20% more efficient than the first. There were, however, two cases where the new version using (\$||) was inferior. Overall speedup relative to the sequential case ranges from 1.07 to 4.17 (mean: 2.41) for the shared-memory configuration, or 1.06 to 3.19 (mean: 1.96) for the distributed-memory configuration. While the average parallelism is the same in five cases (`PrintUtils`, `Syntax`, `TChecker`, `Optimiser` and `Main`), the speedup is higher in all cases for the shared-memory results as shown in Table 1.

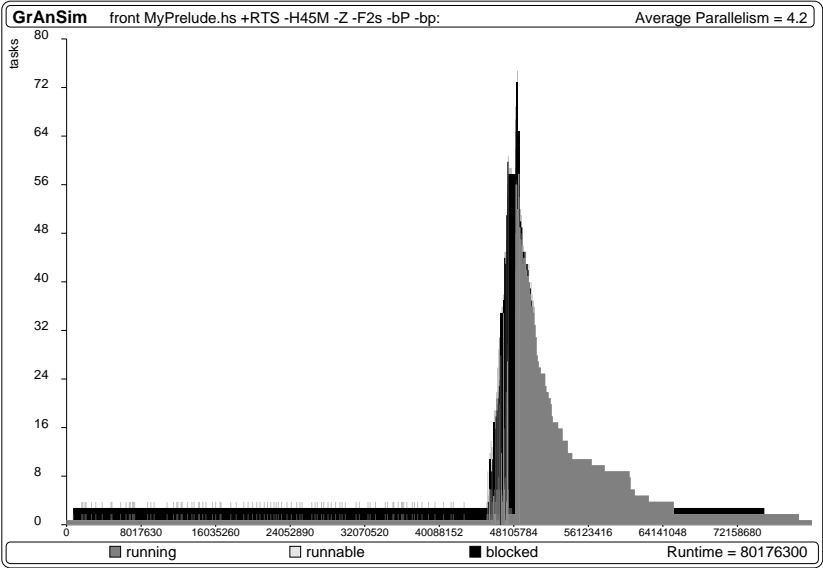


Fig. 4. Top-Level Pipeline compiling `myPrelude` (Distributed Memory Machine)

Figure 4 shows a sample parallelism profile for one of our input modules. Such a profile is usually used to provide visual information about the parallel behaviour of an input program during its execution life-time (see [8] for more details). It shows, as time progresses (horizontal axis), the number of tasks (vertical axis) in various states of execution—running, runnable and blocked. Although interpreting activity profiles such as this is very difficult without the aid of specialised tools [2], we speculate, armed with experience and knowledge of our source code, that the initial sequential segment in this profile is due to parsing and I/O overheads. The initial black-shaded portion indicates that several parallel tasks block immediately because the initial parse tree on which they will operate has not yet been produced. The rest of the profile depicts raggedly-declining parallelism. This indicates that a large amount of parallelism is available immediately after

termination of the I/O task, but that the parallelism reduces as these tasks complete and their results are absorbed into the overall result of the computation.

Compared with experimental results that have been previously achieved by other researchers using the same technology to measure other programs [16,18], these results are quite encouraging. These earlier studies achieved average parallelism of 1.2 on a database-type problem [16] and about 2.5 on the Lolita natural language parser [18] at the same top-level parallelisation stage.

3 Parallelising the Individual Passes

In the preceding section we have summarised our experimental results of parallelising the top-level pipeline of Naira and we now consider the parallelisation of each compiler pass. Each of sections 3.2–3.5 considers a single pass. Section 3.6 considers the overall effect of combining the parallelisation of all the passes.

Before describing the parallelisation of the individual compiler phases, we briefly present the name supply mechanism that we used in conjunction with evaluation strategies to successfully parallelise the phases.

3.1 Unique Name Servers

Data dependency can be a significant hindrance to the effective exploitation of parallelism. In the subalgorithms of Naira discussed in the next sections, unique name servers are used to help break data dependencies and so expose additional parallelism. Our early experiences with some name supply mechanisms suggest that a simple name server similar to that of Hancock [11] is acceptable, and more complex name servers such as those described by Augustsson *et al.* [1] are not needed.

3.2 The Pattern Matching Compiler

The pattern matching compiler transforms function definitions that use equational patterns into equivalent ones involving case expressions with simple variable patterns, as described by Peyton Jones [11].

Once the definitions within a module have been parsed, the pattern matching transformation can be applied to each of these definitions independently and in parallel. The pattern matching compiler is implemented using the function `mkDefs` (Figure 5), whose arguments are the file name, a pattern matching symbol table, `env`, and the list of definitions output from the parser.

This definition of `mkDefs` introduces coarse-grained parallelism which allows the analysis of each binding to proceed in parallel with that of the others. `parallelPair` simultaneously applies `parForceList` to both components of a 2-tuple. `sparkList` causes the parallel evaluation to *weak head normal form* of elements of a list.

In order to provide additional finer-grained parallelism, we have tried three further parallelisation steps: 1) compiling local pattern definitions in parallel

```

mkDefs :: Name → (AssocTree [Char] (Int, [String])) → [Def] → [Def]
mkDefs fileName env [] = []
mkDefs fileName env l =
    mkAppend $|| parallelPair parForceList $
    fork2 (checkAdjDefs fileName env,
           mkDefs fileName env) $|| parallelPair parForceList $
    partition (sameDef (head l)) $|| sparkList $ l
fork2 (f, g)(x,y) = (f x, g y)
sparkList = parList rwhnf

```

Fig. 5. The Pattern Matching Compiler: **mkDefs**

with their top-level parents; 2) parallelising heavily used auxiliary functions and 3) changing the parse tree representation to be a list rather than a binary tree. None of these made any significant difference to the overall performance.

3.3 The Lambda Lifter

The Naira lambda lifter is fairly conventional, comprising a scope analyser, a renamer, a dependency analyser, and the final lifting operation. As usual (see [6,13]), this transformation is relevant only for (mutually) recursive top level bindings and for function definitions which have local definitions.

```

lLift :: String → AssocTree String (Int,[String]) → [Def] → [Def]
lLift fileName stPM defs =
    id $|| sparkList $
    lifter.triplet1 $|| parallelTriple sparkList $
    scopeAnalysis fileName stPM [] [] initNS 1 $|| sparkList $ defs
where triplet1 (x, y, z) = x

```

Fig. 6. The Lambda Lifter: **lLift**

The two main functions, **scopeAnalysis** and **lifter**, which collectively perform the bulk of the computation in the lambda lifting process are combined into a two-stage pipeline. The first stage performs scope analysis, and incorporates the renamer and dependency analyser. The second stage (**lifter**) computes the transitive closure of each function's free variables and performs appropriate substitutions. Notice the use of the identity function **id** which ensures that **sparkList** is applied to the result of **lifter** before being returned. **parallelTriple** simultaneously applies **sparkList** to a 3-tuple of lists (see Figure 6). This parallelisation leads to a modest performance improvement: average

parallelism improves to 1.1 overall for eight of our input modules with consequent modest speedups.

Because the renamer simply associates an identifier with a small integer, and since only locally defined identifiers are renamed (the parser would have reported any name clashes amongst top level identifiers), it performs very little work, and thus is not suitable for parallelisation. Although, in comparison, the dependency analyser performs a relatively large amount of computation, it is based on sequential graph algorithms and therefore, unfortunately, it cannot be easily parallelised.

In the second pipeline stage, the lambda lifter collects free variables and forms and solves a series of equations [6] in order to determine the complete set of free variables for each function. It transpires that for our sample programs and, we surmise, for many programs this is not an expensive process, since each local binding contains very few local definitions, and so it is not worth parallelising. This is a consequence of separating the source definitions into minimal dependency groups in order to aid type checking [11]. Overall we obtained modest speedups when the pattern matching compiler is parallelised as detailed in [7].

3.4 The Type Checker

Cost-centre profiling [14] reveals that, as is often the case, the type checker is the most expensive part of the compiler, both in terms of space usage and runtime. In fact it is more expensive than all the other compiler passes put together. This is largely because the type checking process incorporates complex sub-algorithms like unification. The effectiveness of our overall parallelisation therefore depends significantly on how much useful parallelism can be extracted from the type checker.

Parallelising the Type Checker: Initial Step—The Top-Most Function

The function `tcModule` (Figure 7) is used to implement the type inference algorithm for a collection of definitions in a module. The first three arguments to this function are the file name, the type environment and a list of exported values. The last three arguments contain the types of imported values, a list of type synonyms and the list of definitions in the module.

The function `tcModule` initially associates each bound name with an initial type variable creating an auxiliary environment, `auxEnv`. These initial types usually become specialised as unifications and substitutions are performed. Inferred types are also checked against user-declared type signatures in accordance with the usual Haskell rules [4].

The type checker is parallelised using a parallel name server to minimise data dependencies and thus avoid sequentialising the inference process. For instance, to typecheck two quantities d_1 and d_2 , we analyse them simultaneously in the current type environment, each returning a type and a substitution record. If a variable v common to both d_1 and d_2 is assigned (possibly different) types t_1

```

tcModule fileName env exports typeList syns defs =
    (typeList ++ topDefsTypes) 'using' parForceList

where
    (ns0,ns1)    = split initNS
    tIds          = map getDefId defs
    auxEnv        = mkTypeVars tIds ns0
    topDefsTypes = tcTopDefs fileName env auxEnv exports initSubs ns1
                  syns defs

```

Fig. 7. The Type Checker: `tcModule`

and t_2 from these two independent operations, t_1 and t_2 will be unified in the presence of the resulting substitutions and the unified type associated with v .

For the initial parallelisation step of the type checker we obtained promising results with mean average parallelism and mean speedup values of 2.04 and 2.01 respectively. These results are quite similar to those obtained for the parallelisation of the top-level pipeline (see [7] for more details).

There are three obvious avenues for further exploitation of parallelism in the type checker. These are: a) inside local definitions, b) on calls to frequently used functions; and c) at other expression constructs. We will consider each of these in turn in the following subsections.

Parallelising the Type Checker Step 2—Local Definitions

The second step is to add strategic code to the function `tcLocalDefs` (Figure 8) so as to create additional parallel tasks to infer the types of the locally defined identifiers.

```

tcLocalDefs fileName env subs ns syns [] = ([], [], subs)
tcLocalDefs fileName env subs ns
    syns (VDef (IdPat id) args e:defs) =
    res 'using' parTriple rnf parForceList rwhnf

where
    (ns0,ns1)    = split ns
    (infTy,subs1) = typeCheck fileName
                    id (mkLam args e) env subs syns ns1
    (idsL,tysL,subs4) = tcLocalDefs fileName env subs ns0 syns defs
    res            = (id:idsL, infTy:tysL,subs4)

```

Fig. 8. Type Inference for Local Definitions: `tcLocalDefs`

The strategic code `res 'using' parTriple rnf parForceList rwhnf` ensures the creation of parallel tasks for `res`, the result of `tcLocalDefs`. This modifica-

tion leads to a modest increase in parallel activity, but no significant increase in speedup [7].

Parallelising the Type Checker: Step 3—Unification

In the third step, we introduce parallelism in the unification algorithm, `mkUnify`. It is clear that type unification is one of the most frequent operations during type inference, and we therefore hope to get some speedup by doing it in parallel with the main type inference.

We introduce parallelism here by sparking a child task to carry out unification on sub-trees, by applying the strategy below to *each call* of `mkUnify` inside the type checker. In the code segment below `result` is the result of the inference step of which `resOfUnify`, the result returned by `mkUnify`, is a part.

```
(resOfUnify 'using' parPair rnf rnf) 'par' result
```

We found that “factoring” the `parPair rnf rnf` strategies from the type checker and using only one of them at the definition of `mkUnify` (rather than at the call sites of `mkUnify`) will degrade performance. This is because using the strategy inside `mkUnify` alone amounts to creating parallel tasks inside a substructure whose enclosing structure may only be lazily demanded. Using the strategies in both the type checker and `mkUnify` on the other hand is not efficient since it leads to duplicate applications of the same strategy on the same data structure. The best option we have found is to apply strategies to as outer a level as possible.

Figure 9 shows the activity profile that results when compiling `MyPrelude` at parallelisation step 3. We recorded a modest overall performance improvement as a result of parallelising unification.

Parallelising the Type Checker Final Step—Structured Constructs

In the final step of parallelising the type checker, we make use of our name supply mechanism (in addition to the parallelisation code in the preceding steps) to further break data dependencies so that subcomponents of bigger expressions like conditionals can be type checked in parallel and so that the resulting substitution records can be composed in parallel as well. Composing substitutions in this way is important since this happens very frequently in the Naira compiler.

Simulated results for this step are shown in Table 2. Overall speedups range from 1.11 to 4.24 (mean: 2.43) for the shared-memory and from 0.75 to 3.57 (mean: 2.05) for the distributed-memory set-up. It is not clear why typechecking the Optimiser module yields a slow-down for the distributed-memory machine, but communications costs presumably dominate this particular computation.

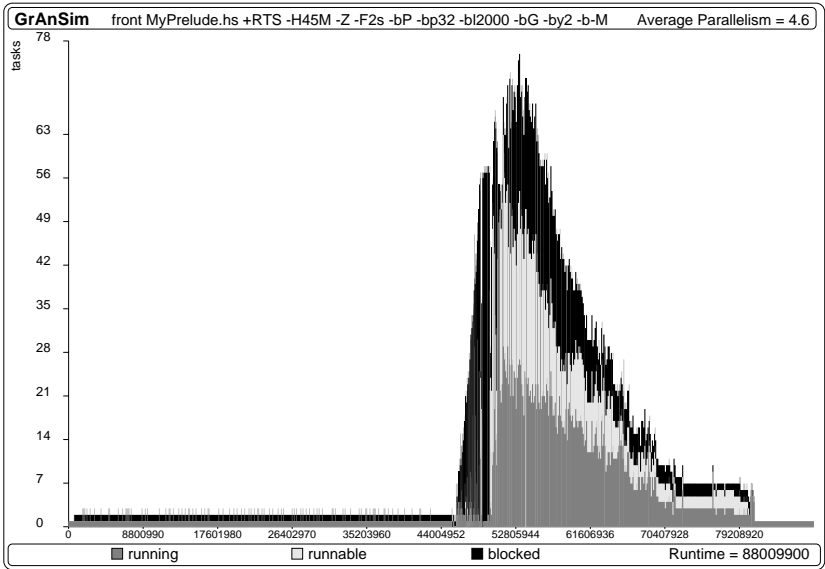


Fig. 9. Type Checking myPrelude – Step 3 (Distributed Memory Machine)

<i>Module</i>	Avg. Par.	Speedup	<i>Module</i>	Avg. Par.	Speedup
MyPrelude	6.3 (6.1)	3.70 (3.57)	MatchUtils	6.4 (6.0)	3.39 (3.17)
DataTypes	7.1 (4.2)	4.24 (2.55)	Matcher	6.8 (5.1)	2.64 (1.93)
PrintUtils	2.2 (1.9)	1.48 (1.23)	LambdaUtils	3.4 (3.2)	2.01 (1.87)
Printer	3.4 (3.1)	2.13 (1.95)	LambdaLift	4.2 (6.8)	2.39 (2.06)
Tables	6.6 (5.1)	3.58 (2.71)	TCheckUtils	6.1 (4.4)	3.26 (2.58)
LexUtils	9.1 (7.3)	2.85 (2.40)	TChecker	2.7 (2.7)	1.66 (1.64)
Lexer	5.2 (4.6)	1.60 (1.40)	OptimiseUtils	2.6 (3.2)	1.50 (1.41)
SyntaxUtils	4.5 (4.3)	3.26 (3.10)	Optimiser	1.2 (4.5)	1.11 (0.75)
Syntax	1.9 (1.8)	1.26 (1.23)	Main	5.5 (5.2)	1.67 (1.51)

Table 2. Type Checker (Final Parallelisation): Speedup and Parallelism for a Simulated Shared-memory Machine (Distributed-memory Results in Bracket).

3.5 The Optimiser

The optimisation pass specialises general function applications (using arity information to short-circuit argument satisfaction checks [12]) and transforms case-expressions to a simplified form better suited for code generation.

One parallel task is created to collect arity information for the module in parallel with optimising the module (producer/consumer parallelism). Since once

arity information is available there are no data dependencies between the definitions, all parse tree simplifications can then be performed in parallel.

Once again, it transpires that this compiler pass is not computationally intensive, and so there is little advantage to the parallelisation process. Indeed, the resulting activity profiles are very similar to those for the pattern matcher and lambda lifter (Sections 3.2 and 3.3).

3.6 Combined Parallelisation

In the preceding sections we have shown how to parallelise the top-level pipeline and our four main compiler passes. In this section we consider the effect of combining all our parallel optimisations.

<i>Module</i>	Avg. Par.	Speedup	<i>Module</i>	Avg. Par.	Speedup
MyPrelude	7.9 (7.7)	3.97 (3.88)	MatchUtils	6.9 (4.7)	3.43 (2.35)
DataTypes	8.3 (8.1)	4.37 (4.28)	Matcher	7.4 (7.3)	2.63 (2.61)
PrintUtils	2.6 (2.5)	1.60 (1.56)	LambdaUtils	4.4 (4.3)	2.40 (2.23)
Printer	3.7 (3.7)	2.18 (2.16)	LambdaLift	3.4 (3.4)	2.40 (2.37)
Tables	7.1 (7.2)	3.56 (3.55)	TCheckUtils	5.6 (5.3)	3.80 (3.74)
LexUtils	10.1 (9.6)	2.88 (2.78)	TChecker	2.9 (2.8)	2.18 (1.57)
Lexer	5.5 (5.3)	1.61 (1.58)	OptimiseUtils	3.5 (3.5)	1.55 (1.49)
SyntaxUtils	4.7 (2.7)	3.26 (1.86)	Optimiser	2.4 (2.3)	1.12 (1.08)
Syntax	2.0 (2.0)	1.27 (1.26)	Main	5.5 (5.1)	1.73 (1.51)

Table 3. Combined Parallelisation

Table 3 shows the results that are obtained when all our parallelisations were used. Compared with the results of Table 2, we observe that there is some interference between individual parallelisations, and so the overall performance is not as high as might be hoped for. Overall speedups range from 1.12 to 4.37 (mean: 2.46) for the shared-memory configuration, or 1.08 to 4.28 (mean: 2.32) for the distributed-memory configuration. This represents a slight improvement over simply parallelising the top-level pipeline. Once again performance is not significantly degraded for the distributed-memory configuration.

3.7 Further Parallelisation

In the previous section we summarise our results for the combined parallelisation. In this section we summarise results obtained after re-examining more closely the algorithms on which the individual phases were based. This was motivated by the fact that our experimentation with different evaluation strategies led to no significant overall performance improvements.

<i>Input module</i>	Average parallelism			Speedup: 8 processors		
	Ideal	SMM	DMM	Ideal	SMM	DMM
MyPrelude	6.2	4.7	4.5	4.32	3.34	3.31
DataTypes	6.5	4.5	4.8	5.18	3.71	3.96
PrintUtils	2.7	2.5	2.4	2.74	2.65	2.50
Printer	4.2	3.8	3.8	4.25	3.96	3.90
Tables	7.8	4.9	5.1	4.84	3.14	3.27
LexUtils	10.9	6.3	5.8	4.95	3.47	4.45
Lexer	5.9	4.8	4.7	4.38	3.69	4.18
SyntaxUtils	8.4	5.0	4.7	8.13	5.81	5.53
Syntax	1.9	1.8	1.8	1.40	1.39	1.35
MatchUtils	5.2	4.0	3.5	5.30	4.20	3.61
Matcher	6.7	4.1	4.2	4.64	3.74	3.59
LambdaUtils	4.1	3.6	3.3	2.53	2.22	2.09
LambdaLift	6.7	4.5	4.5	6.11	4.03	4.39
TCheckUtils	7.3	4.9	2.8	6.47	4.68	5.32
TChecker	3.1	2.6	3.4	1.89	1.81	1.76
OptimiseUtils	3.1	3.0	5.6	3.68	3.61	3.33
Optimiser	3.5	3.4	2.7	4.79	4.79	4.70
Main	3.9	3.6	3.5	2.93	2.67	2.66

Table 4. Further parallelisation results summary: 8 processors.

In our search for culprits we focussed on the type checker because it is more expensive than all the other parts of the compiler. Our main finding was that composition of substitutions, which is performed quite often in Naira, forms the main bottleneck in the parallel performance of the compiler. We revised our implementation of this algorithm and fine-tuned our strategic code resulting in substantial performance improvements as summarised in Table 4.

There are some important differences between the experimental set-up used to obtain the results in Table 4 and that used to obtain previous results. Firstly, we have simulated an 8-processor machine rather than a 32-processor one in order to show the minimum number of processors that can be used to obtain the best speedups for our input programs. Secondly, we have enabled task migration on the 8-processor machine in order to achieve the best processor utilisation.

Comparing the results of Table 3 with the corresponding distributed-memory results in Table 4 we see that the performance is now much better even though we are simulating only eight processors. The results obtained by on an idealised GrAnSim set-up with zero communication costs in Table 4 are, as expected, better than those obtained from both the shared-memory and the distributed-memory machine emulations. Comparing the shared-memory and the distributed-memory results we see, however, that the shared-memory results are only slightly better. This probably indicates that we have successfully obtained a good parallel decomposition of the program leading to cost-effective communication.

3.8 Naira on a Network of SUN Workstations

The results presented so far were based on the GrAnSim simulator. In this section we summarise results obtained on a network of Sun workstations (SPARCstations 4/20), running Solaris 2 and connected to a common Ethernet segment.

The experiments were deliberately carried out under different network conditions so as to avoid possible scheduling accidents from having drastic effects on our results. The experiments were run with varying numbers of processors using GUM [17], a portable parallel implementation of Haskell, which exploits the standard message passing communication harness of PVM (Parallel Virtual Machine). GUM does not support task migration and tasks are distributed lazily, although data distribution is performed somewhat eagerly¹. It uses a fair task scheduler whereby runnable tasks are executed using a round-robin policy.

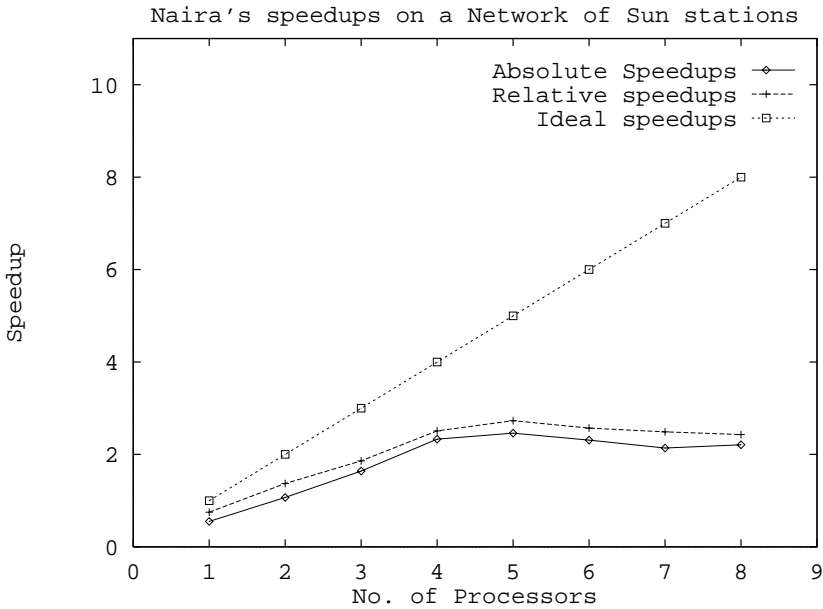


Fig. 10. Speedup summary of Naira on GUM.

We measured a wall-clock speedup of 2.46, and a relative speedup of 2.73 on a network of five workstations as depicted in the graphs of Figure 10. These results are in agreement with those obtained using the GrAnsim simulator which predicted a speedup of 3.01. Because of the additional overhead in the parallel program, the parallel code on a single processor slows down by about 25%. Better

¹ By speculatively packing some ‘nearby’ reachable graph into the reply message for a closure request.

speedups would be expected if the machine resources were to be dedicated solely to our experiments and if task migration, which has been demonstrated to be necessary for good parallel performance, were to be implemented in GUM. Full details of these results are presented elsewhere [7].

4 Related Work

While there have been many successful attempts to produce parallelising compilers for pure functional languages (e.g. [17,9,15]), we know of no similar attempt to parallelise a complete working compiler. There have, however, been a few attempts to consider individual compiler stages. For example, Hammond has described a parallel type inference algorithm based on using monads to exploit parallelism within type graphs at a finer granularity than that described here [5].

5 Conclusions and Further Work

Using the GranSim simulator, we have demonstrated that speedup can be achieved for a functional language compiler running as a parallel program, even in the relatively harsh environment of a distributed-memory machine. As expected, the overall speedup we have achieved so far is useful rather than dramatic. Unusually, the speedups achieved for the shared-memory configuration are not significantly greater than for the distributed-memory configuration. While this may reflect a good parallel decomposition with low communication overheads, it may also indicate that hard data dependencies (probably within the top-level pipeline) impose a major limitation on the overall parallel potential of the problem. This would repay further investigation.

Clearly the most important overall parallelisation step of those we have attempted is the parallelisation of the top-level pipeline. While we had hoped to achieve better results from parallelising individual compiler passes (and may still do so with further effort), this does support the contention that it is possible to achieve modest performance improvement for modest effort.

Of the individual compiler passes, the type checker was clearly the most productive from a parallel perspective, yielding performance equivalent to that obtained from parallelising the top-level pipeline in isolation. This is because its cost dominates that of the overall compilation process. Other passes are relatively cheap, and therefore give less overall improvement. Unfortunately, it has so far proved impossible to combine the speedup obtained from the type-checker with that obtained from the top-level pipeline.

Careful study of the parallelism profiles reveals that file I/O and parsing accounts for a significant part of the remaining sequential component to the computation (and therefore by Amdahl's law represents a major limitation on further optimisation). Parallelising I/O can be quite difficult, and is probably beyond the scope of the work reported here.

Our experiences with parallelising individual compiler passes has shown that even with the tools available it is quite hard to understand and predict the

performance of the compiler. Small changes in the parallelisation code can also lead to significant changes in parallel behaviour for some inputs. Clearly, there is a need for even better parallel performance monitoring tools. Parallel cost-centre profiling may be a step in this direction [2].

We note that Naira is an experimental compiler rather than a state-of-the-art optimised compiler like the Glasgow, Chalmers and Yale Haskell compilers. While it would be interesting to start from the basis of a compiler such as this, and our results would naturally be more directly useful to a number of real users, the fact that GHC is already highly optimised for *sequential* compilation inevitably makes it a much harder proposition than Naira to parallelise successfully. We hope that the directions we have explored will help focus any available parallelisation effort in production compilers such as this.

Acknowledgements We thank H-W Loidl and PW Trinder for the valuable meetings we had with them and for the use of their parallelisation tools and techniques. We also thank the anonymous referees for their helpful comments.

References

1. L Augustsson, M Rittri and D Synek, "On Generating Unique Names". *Journal of Functional Programming*, 4(1), pp. 117–123, January, 1994.
2. K Hammond, CV Hall, H-W Loidl and PW Trinder, "Parallel Cost Centre Profiling". In *1997 Glasgow Workshop on Functional Programming*, Ullapool, Scotland, September 1997.
3. K Hammond, H-W Loidl and AS Partridge, "Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell". In *HPFC'95—Conf. on High Performance Functional Computing*, pp. 208–221, Denver, CO, April 10–12, 1995.
4. P Hudak, SL Peyton Jones, and PL Wadler (eds.), "Report on the Programming Language Haskell Version 1.2". *ACM SIGPLAN Notices* 27(5), May 1992.
5. K Hammond, "Efficient Type Inference Using Monads". In *Proceedings, 1990 Glasgow FP Workshop*, Ullapool, Scotland, August 1990.
6. T Johnsson, *Compiling Lazy Functional Languages*. Ph.D. Thesis, Department of Computer Science, Chalmers University of Technology, Gothenborg, Sweden, 1987.
7. S Junaidu, *A Parallel Functional Language Compiler for Message Passing Multiprocessors*. Forthcoming PhD thesis, School of Mathematical and Computational Sciences, St Andrews University, Scotland, 1998.
8. H-W Loidl, "GranSim User's Guide, Version 0.03". Department of Computing Science, University of Glasgow, July 1996.
9. EGJMH Nöcker, JEW Smetsters, MCJD van Eekelen and MJ Plasmeijer, "Concurrent Clean". In *Proc. PARLE '91*, Springer-Verlag LNCS 506, pp. 202–219.
10. G Ostheimer, *Parallel Functional Programming for Message Passing Multiprocessors*. PhD Thesis, Department of Mathematical and Computational Sciences, St. Andrews University, Scotland, 1993.
11. SL Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice Hall International, 1987.
12. SL Peyton Jones, "Implementing Functional Languages on Stock Hardware: the Spineless Tagless G-machine". *Journal of Functional Programming*, 2(2), pp. 127–202, 1992.

13. SL Peyton Jones, D Lester, *Implementing Functional Languages: A Tutorial*. Prentice Hall International, 1991.
14. PM Sansom and SL Peyton Jones, "Time and Space Profiling for Non-strict Higher Order Functional Languages". In *Proc. 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 1995.
15. S Skedzielewski, "Sisal". In *Parallel Functional Languages and Compilers*, Frontier Series, ACM Press, New York 1991.
16. PW Trinder, K Hammond, H-W Loidl, SL Peyton Jones, and J Wu, "A Case Study of Data-intensive Programs in Parallel Haskell". In *Proc. 1996 Glasgow Workshop on Functional Programming 1996*, Ullapool, Scotland, July 8-10.
17. PW Trinder, K Hammond, JS Mattson Jr., AS Partridge and SL Peyton Jones, "GUM: A Portable Parallel Implementation of Haskell". In *PLDI '96 — Programming Languages Design and Implementation*, pp. 78-88, Philadelphia, PA, May 1996.
18. PW Trinder, K Hammond, HW Loidl and SL Peyton Jones "Algorithm + Strategy = Parallelism". In *Journal of Functional Programming*, **8**(1), January, 1998.

Lazy Thread and Task Creation in Parallel Graph-Reduction

Manuel M. T. Chakravarty

Institute of Information Sciences and Electronics
University of Tsukuba, Japan
chak@is.tsukuba.ac.jp
<http://www.score.is.tsukuba.ac.jp/~chak/>

Abstract. Lightweight threads can be used to cover latencies occurring in distributed-memory implementations of declarative programming languages. To keep the costs of thread management low, this paper proposes two techniques: first, to distinguish locally scheduled *threads* from globally distributed *tasks*; and second, to create both threads and tasks lazily. The paper focuses on the integration of these techniques into compiled graph-reduction, which was neglected by other researchers; in particular, their approach prohibits both tail call optimization and the use of the push-enter model of function evaluation.

1 Introduction

Lightweight multi-threading is a popular technique for covering communication latencies in parallel computers with distributed memory [6, 13, 2] [7, 11, 3]. It attempts to maximize both the overlap of communication with computation and the overlap of independent communication operations by providing fast, data-driven scheduling of fine-grained parallelism. This is important in parallel implementations of declarative languages, which operate on a global, distributed heap and exhibit highly irregular data-access patterns.

Previous work covered the integration of an abstract notion of threads into the abstract machine language of the Spineless Tagless G-machine (STGM), calling the result STGM_{MT} [3]. The present paper discusses an efficient implementation technique for the thread model of the STGM_{MT}. To minimize the costs of thread scheduling, we discuss two techniques: first, locally scheduled *threads* are distinguished from globally distributed *tasks*; and second, both threads and tasks are created lazily. Threads are fine-grained, locally scheduled work packages that are used to cover communication latencies with useful work. In contrast, tasks provide medium to large sized workloads that are distributed globally to maintain load balance. Threads are only needed when communication occurs and tasks are only needed in case of load imbalance; hence, lazy thread and task creation minimize resource usage.

Although previous work [16, 6, 7, 19, 11] already contained some of these ideas, the proposed overall scheme is original. In particular, the previous proposals conflict with basic techniques of graph-reduction, such as tail-call optimization, the use of the push-enter model of function evaluation, and updating

of thunks. The only exception being GUM [19], which uses a lazy task creation scheme, but does not consider lightweight multi-threading. In summary, the main features of the present proposal are the following:

- Tasks and threads are distinguished as means for load balancing and latency covering.
- Lazy task creation minimizes overheads when work is distributed well.
- Lazy thread creation minimizes overheads when data access is largely local.
- Tail-call optimization, the push-enter model, and updating of thunks are supported.
- Communication is by thread posting.

1.1 Functional Programming and Dataflow Computing

The research in functional programming and the dataflow computing always overlapped, but the overlap was mainly in language design rather than implementation techniques. When it comes to parallel implementations, the focus of the two communities is, to a certain degree, complementary. The implementations of the functional programming community are highly optimized to exploit sequential stock hardware, and parallel implementations add some communication and load balancing facilities on top of that. In contrast, the dataflow community traditionally focuses on low-overhead communication and a data-driven model that tries to minimize the impact of communication latency.

Exaggerating a little, we could say, the functional programming community focused on the sequential parts of a parallel implementation and the dataflow community on the parallel part. Surprisingly, both neglected the techniques of the other largely. Hence, the techniques are partly incompatible. For example, Goldstein et al's [11] lazy threads require that the size of an activation frame never changes after allocation, thus preventing tail-call optimization and the use of the push-enter model, and Flanagan and Nikhil's [7] recent proposal cannot handle frequent updates of heap cells, thus preventing updating of thunks (an essential technique in a lazy implementations). The present paper attempts to bridge the gap by integrating recent multi-threading technology with compiled graph-reduction.

1.2 Outline

The paper is structured as follows: Section 2 provides the setting by reviewing some key concepts of graph reduction and of the Spineless Tagless G-machine (STGM). Section 3 provides a simple taxonomy of parallelism and derives from it a scheduling hierarchy that motivates the distinction between tasks and threads. Afterwards, Section 4 and 5 detail the implementation of tasks and threads, respectively. Then, Section 6 briefly discusses the communication model. Finally, Section 7 reviews related work and Section 8 concludes.

2 Implementation Techniques for Functional Languages

The present paper focuses on threads and tasks in graph reduction; in particular, it discusses their integration into the STGM [17]. Therefore, we start by briefly reviewing some implementation techniques used in the STGM.

2.1 Key Techniques

The following implementation techniques are central for (lazy) functional languages.

Tail recursion elimination. If the last operation that a function performs is a recursive call, it is *tail recursive* and the recursion can be implemented by a loop. Then, the stack space requirement is constant instead of linear to the depth of the recursion. More generally, whenever the last operation of a function is a call to some other function (tail call), the stack space of the caller can be released before the tail call is executed. As a result, the stack space can be reused by the callee. It is common practice to maximize the number of tail calls in functional programs, to improve their space behaviour.

The push-enter model of function evaluation. Implementations of functional languages that support currying have to deal with function applications where the number of supplied arguments is less or greater than the number of arguments expected by the function. Furthermore, complex expressions can be applied to some arguments (not only named functions). In this context, the *push-enter model* of function evaluation is convenient. In the push-enter model, an application $e\ a$ is evaluated by pushing a onto an argument stack, and then, entering (evaluating) e . The evaluation of e may push further arguments, before eventually a named function is called, which takes as many arguments from the stack as it needs. For a more detailed discussion, see [17].

Updating of thunks. Implementations of lazy languages usually implement lazy evaluation by representing yet unevaluated expressions as closures—also called *thunks*—in the heap. Whenever such a thunk is evaluated, it is afterwards overwritten with the result of the evaluation. This is important as it avoids repeated evaluation of the same thunk, and thus, prevents duplication of work when the result of a lazily evaluated subexpression is required repeatedly.

2.2 The Intermediate Language

The STG-language is a restricted functional language that features an immediate operational interpretation. Previous work [3] extends the STG-language with support for multi-threading on an abstract level. We will discuss these extensions later; first, we review some basics of the original STG-language.

To support a direct operational reading, the STG-language enforces some syntactic restrictions. In particular, arguments to functions and data constructors can only be variables, i.e., intermediate values have to be explicitly bound to

names. The STG-languages interprets **let(rec)** bindings as allocation of heap-located closures. The right hand-sides of these bindings are required to have the form

$$\{v_1, \dots, v_n\} \setminus \pi \{a_1, \dots, a_m\} \rightarrow e$$

which is called a *lambda form*. It represents an m -ary function with arguments a_1 to a_m , body e , and free variables v_i .

A thunk (i.e., a yet unevaluated subexpression) is a lambda form where m is zero, i.e., it is a parameterless function. Evaluation of thunks is demanded only in **case** expressions, which have the form

$$\text{case } e \text{ of } \{p_1 \rightarrow e_1; \dots; p_n \rightarrow e_n\}$$

A **case** expression starts by evaluating e , and then, inspecting the result, continues with the branch $p_i \rightarrow e_i$ whose pattern p_i matches the result of e . The special case **case** e_1 **of** $\{x \rightarrow e_2\}$ always matches its single branch, but *forces* the evaluation of e_1 and binds the resulting value to x . The extension proposed in [3], which we discuss in detail later, associates the action of demanding unevaluated expressions with two new constructs, so that **case** expressions are merely used to select one of multiple alternatives; we will actually define the abstract semantics of the new constructs by relating them to **case** expressions of the original STG-language.

As an example, consider the well known **map** function, which is implemented by the following STG-code:

```
map = {} \n {f, l} ->
  case l of
    Nil {}      -> Nil {}
    Cons {x, xs} -> let y  = {f, y}  \u {} -> f {y}
                      ys = {f, xs} \u {} -> map {f, xs}
                      in Cons {y, ys}
```

The function **map** itself is represented by a lambda form, which starts by scrutinizing the second argument l , which is therefore evaluated if it is still is a thunk. In the case where l is a **Cons** cell, the two subcomputations **f y** and **map f xs** are delayed by being wrapped into a thunk referenced by y and ys , respectively.

3 Different Kinds of Parallelism

It is useful to distinguish the following two kinds of parallelism:

- *Logical parallelism* is the overall parallelism contained in a program, i.e., all the computations that may, according to the semantics of the programming language, be executed in parallel or, for that matter, in any sequential order.
- *Physical parallelism* is the fraction of the logical parallelism that is actually employed to utilize multiple PEs simultaneously in a particular program run.

Logical parallelism is a static property of the program, whereas physical parallelism is a property of a particular run of a program; notably, the latter is zero in the case of a sequential run. It is well known that automatic load balancing and dynamic scheduling require the logical parallelism to be substantially greater than the physical parallelism. Otherwise, automatic load balancing may have trouble keeping all PEs busy and communication latencies cannot be covered with useful work.

The fraction of the logical parallelism that is not exploited as physical parallelism, in a particular run, is called *excess parallelism*. Excess parallelism is an important source of useful alternative work while a computation has to wait for the result of some communication operation. By dynamically scheduling multiple workloads on a single PE, we can cover communication latencies using the freedom in the evaluated order provided by the excess parallelism. We call the fraction of excess parallelism that is used to cover communication latencies by dynamic rescheduling *interleaved parallelism*.

In other words, in a given program run, part of the logical parallelism is exploited as either physical or interleaved parallelism. In the former case, the operations are actually executed *at the same time* on different PEs, whereas, in the latter case, the operations are executed sequential, but they are *reordered* to cover some communication latency.

3.1 Tasks Versus Threads

It may seem that, on the implementation level, all logical parallelism should be uniformly represented, so that it can be freely used as physical parallelism (to utilize multiple PEs) and as interleaved parallelism (to cover communication latency). But such uniform treatment conflicts with the opposing requirements of physical and interleaved parallelism: Physical parallelism comes with substantial overhead (mainly due to inter-PE communication), and thus, should use coarse-grained chunks of work to amortize the costs. On the other hand, interleaved parallelism is cheaper for fine-grained than for coarse-grained workloads—as the latter induce higher context-switching costs.

Therefore, a scheduling hierarchy is useful: it uses coarse-grained workloads as physical parallelism and fine-grained parallelism within these workloads as interleaved parallelism. We call the coarse-grained workloads *tasks* and represent the fine-grained parallelism within tasks by *threads*. Tasks are distributed PE-global, to balance the work load over the available PEs. Usually, the work packages that may turn into tasks are explicitly marked in the high-level source (this implies an explicit partitioning scheme). On the other hand, threads are a PE-local resource, which allows to overlap communication with computation and to overlap different communication operations. Threads are meant to exploit the freedom in the evaluation order admitted implicitly by the language semantics.

The following presentation introduces two new constructs for the STG-language that explicitly mark a potential for task and thread creation. Afterwards, the implementation of the new constructs is discussed in depth. The orthogonal topic of generating an intermediate code containing the new constructs from

high-level source code is not detailed in this paper, but it is briefly discussed in Section 8.

3.2 Tasks in the Intermediate Language

As mentioned, the STG-language is a functional kernel language with an immediate operational semantics. We add tasks (i.e., an abstract notion of PE-global workloads) with the **letrem** construct, which has the following general form:

$$\mathbf{letrem} \ x = \underbrace{\{v_1, \dots, v_n\} \setminus \pi \ \{\}}_{\text{potential task}} \rightarrow e_1 \ \mathbf{in} \ e_2 \quad (1)$$

In line with the purely functional semantics of the STG-language, we define the abstract semantics of the new construct by the following equation, which relates the new construct to an expression of the original STG-language:

$$\begin{aligned} \mathbf{letrem} \ x &= \{v_1, \dots, v_n\} \setminus \pi \ \{\} \rightarrow e_1 \ \mathbf{in} \ e_2 \\ &= \\ &\mathbf{case} \ e_1 \ \mathbf{of} \ x \rightarrow e_2 \end{aligned}$$

A **letrem** binding does not allow recursion, i.e., x may not occur in e_1 (the above equation would not make sense in that case).

This indirectly defines the denotational semantics of the new language construct, but it is *not* meant as a definition of its operational semantics (otherwise, the construct would be redundant in the first place). The details of the operational interpretation are given below; for now let us note that, as indicated in Equation (1) above, the lambda form bound to x introduces a potential task. The lambda form is a thunk (it has an empty argument list). It may either be shipped to another PE and be evaluated there, or it may stay on the PE that executes this construct, and be evaluated locally. Closures other than those created by a **letrem** are not allowed to be shipped to other PEs. Therefore, the partitioning of the overall work into potentially parallel workloads is solely determined by the **letrem** constructs in a given program.

3.3 Threads in the Intermediate Language

The above definition of threads mentioned that threads are meant to exploit freedom in the evaluation order that is provided by the semantics of the source language. More precisely, we use them to reschedule computations within the limits fixed by the semantics such that a overlap between communication with computation is achieved. Program transformations can maximize the potential for such overlap if the independence from the evaluation order is made explicit in the intermediate language [3, 4]. We explicitly denote such potential using the following construct:

letpar	
$v_1 = e_1$	— required value #1
\vdots	\vdots
$v_n = e_n$	— required value #n
in	— <i>synchronization barrier</i>
e	— potential thread

where none of the bound variables v_1 to v_n may occur in any of the right-hand side expressions e_1 to e_n . As before, we define the abstract semantics of the new construct by relating it to an expression in the original STG-language:

$$\begin{aligned}
 & \text{letpar } \{v_1 = e_1; \dots; v_n = e_n\} \text{ in } e \\
 = & \\
 & \text{case } e_1 \text{ of } \{v_1 \rightarrow \dots \text{ case } e_n \text{ of } \{v_n \rightarrow e\} \dots\}
 \end{aligned}$$

And again, the operational interpretation of **letpar** justifies its inclusion into the abstract-machine language. To allow an arbitrary execution order of the bindings, the **letpar** construct requires that none of the bound variables v_1 to v_n occurs in any of the right-hand side expressions e_1 to e_n . In other words, the bindings must be independent of each other, so that they can be used to cover latencies induced by remote accesses in other bindings. The body expression e may only be evaluated after the evaluation of all bindings is completed, i.e., the keyword **in** can be regarded as a synchronization barrier for the threads computing the bindings. As a result, the evaluation of e may have to be suspended and later be scheduled out of the sequential execution order; in this case, the delayed evaluation of e has to be realized by a fully-fledged thread.

3.4 Implications of the New Constructs

From the equations that define the abstract semantics of the new constructs, it is obvious that they duplicate the **case** construct's ability to demand thunks. Therefore, it is not necessary to keep this ability in the **case** construct; instead, we change it to **case** x of $\{\dots\}$, where x is restricted to be a variable that we expect to contain an evaluated value. In contrast to the demand in a case expression, a **letpar** admits to express the demand for more than one expression at once, which marks an independence in the evaluation order of the different expressions. Furthermore, a **letrem** expression signals that the work, which has to be performed while evaluating the demanded expression, is sufficiently high to justify transmission to another PE (c.f. [3]). This operational differences make it strictly necessary to introduce both constructs, i.e., they cannot be merged into a single new kind of expression.

3.5 Lazy Task and Thread Creation

The creation of threads and tasks consumes memory as well as time, costs that we do not want to incur when these structures are not needed. This is especially important as we allow more tasks and threads than actually needed in

a particular execution, because dynamically scheduled programs require excess parallelism to be efficient and portable across different parallel machines.

As it is impossible to decide at compile time how often and when tasks are exported or threads are scheduled out of order, a runtime mechanism is needed that provides the *potential* to create tasks and threads, but creates them only when these structures are actually needed. In particular, tasks should only be created when some processor runs out of work and threads should only be created when alternative work is needed to cover some communication latency. The basic strategy to achieve such a behaviour is to use a standard stack discipline to execute the program in depth-first order, but to allow local deviation from this order where necessary. This strategy facilitates the use of the optimization techniques from sequential implementations, at least, where potential parallelism is not exploited. Formally, the advantage of this strategy is supported by Blleloch et al's [1] arguments.

The reminder of this section discusses shortly how the outlined strategy is realized using lazy task and thread creation. The following sections, then, provide the technical details of the implementation of lazy threads and tasks.

Lazy Tasks. Let us start by considering the sequential execution of a **letrem** expression, as it exactly matches the execution sequence that should be taken when the potential task is not needed for load balancing. Afterwards, we discuss how this integrates with the situation where the task is needed for load balancing.

$$\mathbf{letrem} \ x = \{v_1, \dots, v_n\} \setminus \pi \ \{\} \rightarrow e_1 \text{ in } e_2$$

The evaluation of this expressions starts with the allocation of the closure representing the lambda form bound to x . As the evaluation of that lambda form is *potential* parallel work, we first evaluate e_2 . Now, we have to distinguish two cases: (1) e_2 needs the value of x or (2) it does not need it (although, it may still enclose a reference to this closure in its result). In the first case, the closure of x will be evaluated as part of the evaluation of e_2 —we say the potential task is *absorbed* by e_2 . In the second case, e_2 produces its value while the closure is still unevaluated; in this case, the closure containing e_1 is evaluated after the evaluation of e_2 finished. Hence, in either case, both e_2 and the thunk containing e_1 are evaluated (i.e., **letrem** is strict in both expressions), but the evaluation of e_1 is delayed as long as possible, to provide potential work for other PES.

Now, let's turn to the parallel case: If, while e_2 is evaluated, the thunk referenced by x is needed for load balancing (and it was not already required by e_2), it is exported, as a new task, and evaluated in parallel to e_2 . Hence, a task is created from this potential only when actually needed; in other words, it is created lazily. In this case, when e_2 completes, the **letrem** is also completed, although the evaluation of e_1 may still be in progress. Overall, we can express the strategy for evaluating a **letrem** expression as follows: *Evaluate e_2 first, but consider the completion of e_2 as a demand for e_1 .*

The **letrem** construct is essentially the same as a *future* in Multilisp [12] and Mul-T [14]. The idea to create tasks lazily for futures was introduced by [16],

but we will see below that we realize this basic idea rather differently. A similar evaluation strategy is applied in GUM [19] for the `par` meta-function, with the difference that `par` is not strict in its first argument.

Lazy Threads. Next, consider the sequential evaluation order for the `letpar` construct. Afterwards, we define its potential parallel behaviour, which is employed when a communication operation requires some rescheduling to cover the communication latency. The general form of `letpar` is

$$\text{letpar } \{v_1 = e_1; \dots; v_n = e_n\} \text{ in } e$$

In the sequential case, the bindings are processed from left to right, and finally, e is evaluated, which realizes the conventional left-to-right, depth-first traversal.

In the parallel case, evaluation of any e_i may be delayed by a remote access. Then, the remaining bindings may be evaluated while the remote access is pending. Only the evaluation of the body e has to wait until all e_i are evaluated. If all bindings are processed, but some did not yet complete, the evaluation of e has to be postponed, i.e., it has to be scheduled out of the standard evaluation order, which requires a fully functional thread. Meanwhile, alternative bindings of enclosing `letpars` may be evaluated (these need not be statically enclosing, but may be part of any function activation that is higher in the calling hierarchy). Hence, threads (like tasks) are scheduled out of order only when necessary.

4 Global Task Scheduling

Tasks are communicated between PEs only on explicit demand from a PE that ran (or is in danger to run) out of work. The PE that needs work sends a *work request* to other PEs to obtain tasks. Such a scheme of distributing work is usually called *work stealing*; and there seems to be a consensus that it is the Right Thing to do [16, 19, 2, 7, 11], we do not discuss it further.

4.1 Potential Tasks

Each closure introduced by a `letrem` construct may turn into a task that is asynchronously evaluated on an arbitrary PE—therefore, we call these closures *task closures*. A task closure already stores most of the information that is needed to create a task, namely the code to be executed and its environment. Their remain two questions: First, how do we ensure that a task closure is, at latest, evaluated after the body expression of the `letrem` was evaluated, and second, how does a work request locate an unevaluated task closure? The answer to both questions centers around the stack management used during evaluation of a `letrem` expression.

After creating the task closure, three items are placed on the runtime stack: (1) a pointer to the task closure, (2) a link pointer initialized to `NIL`, and (3) a

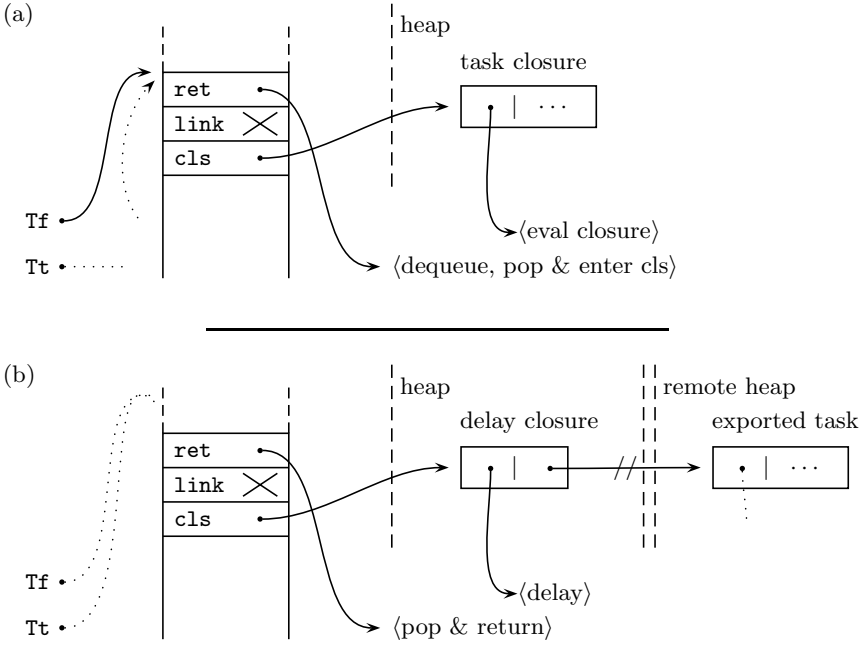


Fig. 1. Task management: (a) potential task; (b) exported task.

code pointer referencing the return continuation. Together, these items constitute a *task frame*. When the evaluation of the body expression of the `letrem` finishes, the task frame is on top of the stack and the return continuation referenced by (3) is executed. It removes the task frame and proceeds by evaluating the task closure.

As for the second question, the link pointer (2) is used to build a task queue; its front and the tail element are referenced by two variables `Tf` and `Tt`, respectively. Work requests obtain tasks from the tail, and new task frames are added to the front. In other words, the link pointer is used to find work requests in the stack without the need to search through the stack frames. We use a queue, not a stack, because the potential tasks created first, should also be stolen first; they usually represent larger work packages [12, 19]. A task frame with its task closure is displayed in Figure 1(a). The evaluation of the body of the `letrem` that created the task frame uses the dashed area of the stack (which grows bottom up in the figure). If this leads to further task frames, the `link` field and `Tf` are adjusted accordingly.

4.2 Task Creation

When a PE receives a work request and has an unevaluated potential task, two operations are performed: (1) the potential task is exported and (2) the task

queue and task frame is correspondingly updated. We call the two PEs participating in the transferal of the tasks the exporting and importing PE.

Exporting. The potential task is exported by transmitting the corresponding task closure to the importing PE. After receiving it, the latter PE obtains a task from the task closure by evaluating it. On the exporting PE, the task closure is overwritten with a delay closure referencing the copy of the closure that is created on the importing PE. When the importing PE completed the evaluation of the closure, it sends the result to the exporting PE, which overwrites the delay closure with the result value.

Updating the Local Task Frame. To reflect the fact that the potential task was exported, it is dequeued from the task queue (it is always at the tail of the queue). Furthermore, we have to alter the return continuation referenced by the task frame of the exported tasks; otherwise, it will later attempt to evaluate the task, which was exported meanwhile. Therefore, the pointer to the return continuation is overwritten by a code pointer to a new return continuation, which simply pops the task frame and returns.

The state after exporting a task is displayed in Figure 1(b). The variables **Tf** and **Tt** point into the younger regions of the stack, because we always export the oldest task. The delay closure references the exported copy of the task. If the delay closure is entered, the corresponding thread is suspended until the remote evaluation of the task completed.

Sparks Versus Potential Tasks. The representation of potential tasks is akin to sparks in GUM [19], but with a significant difference: The discussed model does not need a structure like GUM's spark pool, the queue of potential tasks is part of the runtime stack.

5 Lightweight Multi-Threading

As discussed previously, thread scheduling should use an evaluation order that is as close as possible to the sequential evaluation order, which is based on a stack discipline and traverses an expression left-to-right and depth-first. In the case of a **letpar** expression

$$\mathbf{letpar} \{v_1 = e_1; v_2 = e_2; \dots; v_n = e_n\} \mathbf{in} e \quad (2)$$

the sequential evaluation scheme would push a return continuation for

$$\mathbf{letpar} \{v_2 = e_2; \dots; v_n = e_n\} \mathbf{in} e \quad (3)$$

on the stack and, then, evaluate e_1 . After completion of e_1 , the return continuation proceeds similarly, until all bindings are processed and the body can be evaluated.

As long as no communication latencies need to be covered and we can keep this evaluation order, we want to use the same procedure. Only when, say, the evaluation of e_1 is stalled due to a remote access, we want to proceed with the return continuation (i.e., the evaluation of the expression in (3)) *without* waiting for the completion of e_1 . As we require all bindings in a **letpar** to be independent, evaluation can proceed with the other bindings in case of a delay, but when all bindings have been processed, progress critically depends on the completion of all e_i . In other words, if all bindings have been processed, but the result of one or more right hand-sides is still outstanding, the evaluation of the **letpar** has to be *suspended* until these results are available. In this case, the body of the **letpar** turns into an independent thread, which is scheduled out of the standard sequential evaluation order.

Overall, we can distinguish three states in the evaluation of a **letpar**: (a) sequential, (b) explicitly synchronizing, and (c) explicitly scheduled—in the sequential case, both synchronization and scheduling are implicit. These three states are defined as follows. The evaluation of a **letpar** is *sequential* as long as there is no right-hand side whose evaluation is delayed. The **letpar** must be *explicitly synchronized* when at least one binding is delayed; this implies to keep track of the number of delayed right hand-sides as well as to handle the asynchronous delivery of their results. Finally, a **letpar** has to be *explicitly scheduled* when it must be suspended; only after all delayed results have been provided, the **letpar** can be rescheduled to evaluate its body expression. We discuss the implementation of these three possible states of a **letpar** in the following, but first let us discuss the situation where a right-hand side is delayed in more detail.

On first sight, it might seem that the above treatment of the body expression is not sufficient when a **letpar** binding is delayed, and that the binding, or more precisely, the evaluation of its right-hand side e_1 itself also has to be explicitly suspended. Fortunately, this is not necessary, because there are only two situations where a delay can occur: either it is within the binding of a dynamically-nested¹ **letpar** or not (it may still be in the body of a nested **letpar**). In the latter case, the delay is bound to occur during a *tail call* into a thunk (a suspended expression).² In other words, the delayed value itself is the result of e_1 (and there is no remaining computation, which has to be suspended).

Now for the other case, namely when the delay occurs within the binding of a dynamically-nested **letpar**. In this case, the delay propagates to the enclosing **letpar** only if the nested **letpar** needs to be explicitly scheduled (case (c) above). This implies that the body of the nested **letpar** was already suspended and turned into a fully functional thread. Hence, there remains nothing to do to suspend the evaluation of e_1 .

In the following, we discuss the stack-based implementation of **letpar** expressions. More precisely, we generalize the conventional notion of an activation record to the concept of a thread frame. The bindings and body of a **letpar**

¹ This includes **letpars** that occur within called functions.

² It is a tail call, because the right-hand side of **letpar** bindings are the only place where evaluation is demanded (cf. Section 3.3).

are evaluated in different threads, which can be scheduled sequentially, explicitly synchronized, or explicitly scheduled as mentioned above. Furthermore, we call the currently executed thread *active*.

5.1 The Active Thread

Each PE has exactly one active thread, which occupies the computational resources of the PE, in terms of both memory and processing time. Its state is kept in processor registers as far as possible. Furthermore, it uses the top of the runtime stack to store local values and spill processor registers when necessary. All this works largely as in the original STGM.

5.2 Sequential Threads

When the active thread starts to evaluate a **letpar** expression of the form displayed in (2) it has to initiate the evaluation of the subexpressions e_1 to e_n . This requires to save the current state of the active thread on the stack, a process called “saving the local environment” in the STGM [17, Section 9.4.1]. It includes pushing a code pointer that references the return continuation, see (2), which has to be executed after the subcomputation completed. The saved environment, including the return continuation, is called the *thread frame* of the thread that initiated the evaluation of the **letpar**. So far, the procedure is as in a purely sequential implementation and the thread frame looks like a conventional activation record. Therefore, we call the thread sequential. Figure 2(a) displays the frame of a sequential thread.

The evaluation of the right-hand side e_1 of the first binding proceeds in the dashed area of the stack on top of the depicted task frame. On completion, the result of e_1 is returned to the return continuation referenced by **ret**, which stores it in the location associated with v_1 and continues with the reminder of the **letpar**.

5.3 Delaying Results: Explicit Synchronization

The situation starts to become more interesting when a remote access delays the evaluation of e_1 . In this case, control also returns to the parent thread via **ret**, but no result for e_1 is provided. The absence of the result is signaled to the parent thread by choosing a different entry point to the return continuation—similar to the idea of vectored returns in the STGM.

Apart from the fact that a delayed result may eventually cause the suspension of the parent thread, which we will discuss in the next subsection, two complications arise from delayed results:

- when the delayed values are finally produced, they have to be communicated to the parent thread and
- we have to keep track of the number of still outstanding delayed values as multiple bindings in a **letpar** may delay their result.

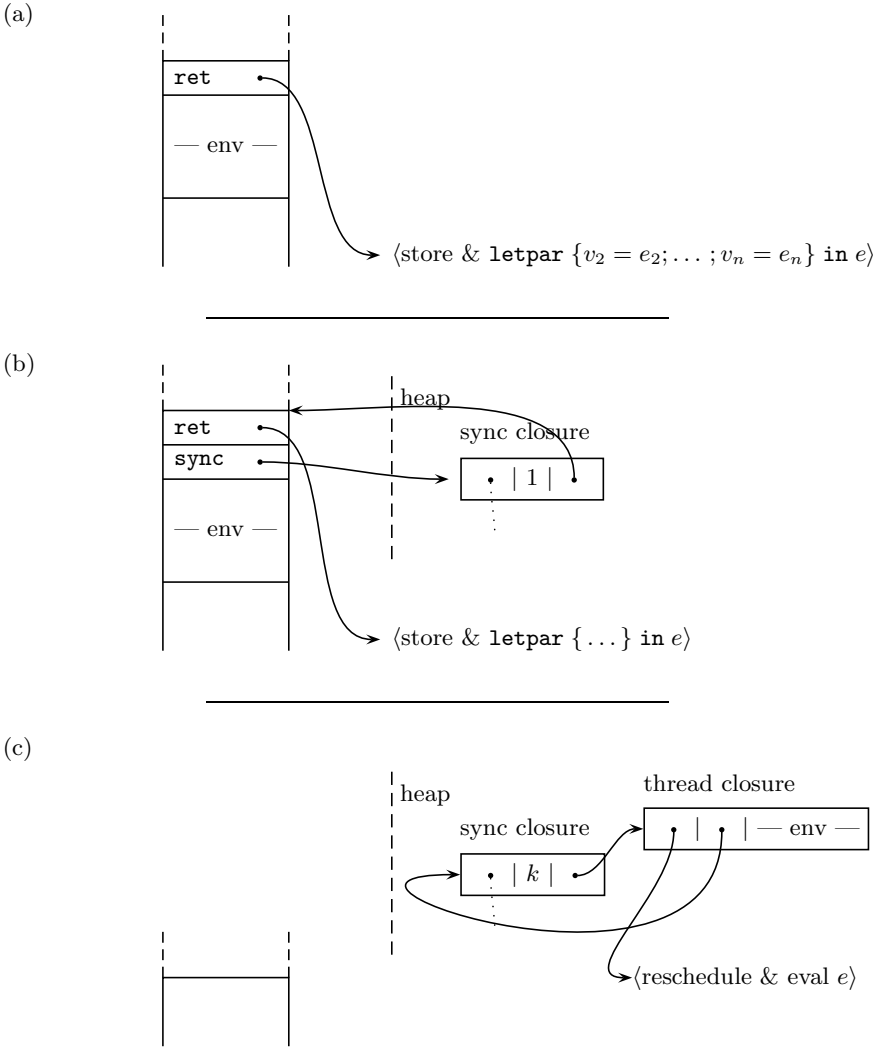


Fig. 2. Thread: (a) sequential; (b) explicitly synchronized; and (c) explicitly scheduled.

Both problems require the introduction of synchronization closures. However, we concentrate on the second problem here and defer the discussion of the first to the following Section 5.5. A *synchronization closure* (or *sync* closure) is a heap-allocated structure, which keeps track of the number of still outstanding delayed values by maintaining a *synchronization counter* (or *sync* counter) and references the thread frame whose synchronization it manages. There is at most one sync closure for each thread frame; it is allocated when the result of a binding

for the evaluated `letpar` is delayed for the first time—i.e., without at least one delayed value in a `letpar` no sync closure is allocated. The frame of an explicitly synchronized thread together with its sync closure is shown in Figure 2(b).

After all bindings have been processed, an explicitly synchronized thread tests the sync counter. If it is zero, all delayed results have been delivered and the body can be evaluated as in the purely sequential case. Otherwise, there are still outstanding results, so the thread finally has to be suspended as discussed next.

5.4 Suspension: Explicit Scheduling

Suspension of a thread occurs exactly when all bindings have been processed, but there are still some outstanding delayed values, i.e., the sync counter is greater than zero. It is in this situation that a thread has to become fully functional, i.e., it is suspended and can be arbitrarily delayed (as it will only be rescheduled after all delayed values become available).

The situation after suspension of a thread is as in Figure 2(c). The sync closure now references the thread closure. Note that the sync closure is a reliable way to access the environment of the thread whether it is on the stack or in the heap. This will be important for the implementation of the asynchronous delivery of delayed values, which is discussed below.

When the last outstanding delayed value is delivered to a suspended thread, the thread may become active again, and its thread closure is linked into a list of *waiting* threads. A waiting thread is rescheduled by entering its thread closure. For the execution of the thread, there are actually two choices: the values saved in the thread closure can be directly accessed from the closure or are first being copied back on the stack. The former would require the code of the body of a `letpar` to be compiled in two versions. Furthermore, it is not clear that copying the values to the stack is significantly more expensive as it may improve locality. Practical experiments are needed to quantify these costs.

5.5 Delayed Values

We discussed the scheduling of threads, but we still need to state how delayed values are asynchronously delivered to the threads waiting for them. When a delay is returned via the alternative entry point of a return continuation, the callee returns a reference to a *store target*, a data structure that holds two values: a reference to a sync closure and an asynchronous return continuation. On receiving a delay, the caller initializes the store target that it gets from the callee with a reference to its sync closure and a reference to a return continuation that should be executed when the delayed value is finally available.

The return continuation expects to get the returned value and the reference to the sync closure as arguments. It uses the sync closure to locate the environment of the calling thread—independent of whether it was flushed to the heap—and stores the returned value in the appropriate place in the environment. Finally,

it decreases the sync counter of the thread, to indicate the arrival of the value. If the sync counter is zero, the thread turns into a waiting thread, as discussed.

5.6 Handling of Pending Arguments

The implementation of a curried language has to consider the situation where a function is applied to more arguments than it needs. In this case, the result of the function will be a functional value, which consumes the excess arguments. Consider the example of $(f\ a\ b)$ where f is a unary function. Then, $f\ a$ will evaluate to a function that is subsequently applied to b . Using the push-enter model (cf. Section 2.1), the excess arguments are pushed onto the stack before the first function is entered, in the example a and b are pushed, then, f is entered. It grabs a and may build thread frames on top of b , which remains on the stack— b is a pending argument. When the evaluation of $f\ a$ is delayed, the thread frames on top of b may be flushed to the heap as the threads are suspended. Finally, b will end up on top of the stack, while evaluation should continue with the thread frame below it. To avoid this situation, a thread frame that is flushed to the heap includes all the pending arguments that are below it into its thread closure.

Furthermore, it is possible that during the evaluation of $(f\ a\ b)$, the function f is not a defined function name, but a variable of functional type. This variable may reference a heap structure that is located on a remote PE, which immediately induces a delay while the arguments a and b are on top of the stack. In this case, a thread closure is created that includes the arguments, waits for the delivery of the value of f , and, when rescheduled, applies that value to the saved arguments.

6 Communication

We use an asymmetric communication protocol, i.e., messages are sent without the need for an explicit receive operation. A message consists of a thread closure that is executed on the receiving PE. The message payload is part of the thread closure's environment. The code of the thread depends on the purpose of the message. For example, if such a thread returns a delayed value, its environment will contain a reference to the store target, which in turn references the sync closure expecting the value. The communicated thread calls the return continuation, passing the reference to the sync closure and the returned value as arguments. In a dynamic environment where the overlap of communication and computation is important, such asymmetric communication has repeatedly proven to be valuable [6, 15, 8].

7 Related Work

There are several proposals for the use of multi-threading in abstract machine models for functional languages, the most recent being [7] and [11]. None of

these considers the implementation of lazy functional languages, and especially not the integration of multi-threading into graph reduction. As a result, they conflict with some key implementation techniques of graph reduction.

In particular, Flanagan and Nikhil’s [7] implementation of pH, which is a lenient language, does not consider the updating of lazily evaluated thunks. Their heap caching scheme does not work in the presence of updates. Although they provide support for updates on a limited number of heap cells (called M-structures), their scheme does not work for the omnipresent updates of graph reduction. Furthermore, their representation and implementation of tasks and threads is substantially different from the present proposal.

The recent proposal of Goldstein et al. [11] was also developed in the context of a lenient language, namely Id (the predecessor of pH). Their proposal has many similarities with the present one (both originate from the work on the TAM [6]), but there are fundamental differences. Most importantly, Goldstein et al. store activation records in a cactus stack implemented by so-called stacklets. After allocation, the activation records *cannot* change their size anymore. This precludes both tail-call optimization and the use of the push-enter model. Further differences are that Goldstein et al. realize all function calls as parallel-ready calls (whereas the present proposal uses the `letrem` construct to identify potential parallel work) and that they realize multi-threading (called strands in their proposal) differently.

Previous work, such as the TAM [6] neglected the need to keep as close as possible to the standard depth-first traversal, which led to excessive resource use, and realize all function calls asynchronously, which is rather expensive.

The GUM system [19] is a parallel implementation of the original STGM, which does not employ multi-threading. Instead tasks are used both to provide potential parallel work and to provide alternative local work in case of a remote access operation. (Please note that [19] uses the word ‘thread’ where we use ‘task’.). The concept of sparks in GUM is close to those of the potential tasks in the present proposal, but sparks are stored in the heap not on the stack. Furthermore, GUM uses an external scheduler to manage tasks, whereas the present proposal employs self-scheduling. The tradeoffs between both proposals have still to be quantified by experiments.

The present proposal is an elaboration of [4, Chapter 5]. The latter (in Chapter 4) simplifies the execution model presented in [3] significantly. Overall, the present proposal does not merely detail a concrete implementation for the high-level operational semantics [3], but it does so for a significantly improved version.

8 Conclusion

This paper details an implementation of lazily created tasks and threads that fits well to key techniques of compiled graph-reduction. We argued that a distinction between globally distributed tasks, used for load balancing, and locally scheduled threads, used to cover communication latencies, is advantageous. To save resources, these structures are only created when actually needed, to achieve

load-balancing or to cover communication latencies, in a particular run of a program.

As mentioned, the generation of the STG-language intermediate code that includes `letpar` and `letrem` constructs from a high-level source code is largely orthogonal to the implementation details discussed in this paper. However, as current research results indicate that a fully automatic parallelization of functional programs is infeasible, some remarks about the generation of the intermediate code are appropriate. There exist several proposals for a high-level description of the partitioning of a functional program into parallel workloads, which allow the programmer to guarantee sufficiently high granularity to utilize a parallel computer efficiently. Examples are the *strategies* of [18] and our work on an extension of Haskell for explicit, but high-level parallel programming [5]. The partitioning information provided in these extensions of purely functional programming can be used to generate the `letrem` constructs, i.e., they determine potential tasks, where the programmer explicitly guarantees that the tasks are sufficiently coarse-grained. In contrast to the occurrences of the `letrem` construct, the compiler is expected to generate the `letpar` constructs without any programmer intervention by exploiting any freedom in the evaluation order admitted by the semantics of the source language; for example, by generating trivial (single binding) `letpars` first, and then, combining them via program transformations [3, 4].

An implementation along the lines discussed in this paper is underway. An experimental evaluation of the proposal will be performed as soon as the implementation is ready. Furthermore, it would be interesting to quantify the trade offs between the present proposal and a purely heap-based implementation of the same intermediate language, where activation frames are always allocated on the heap, and thus, the copying in case of suspension is unnecessary. The purely heap-based model incurs other costs and it is far from clear which costs are higher.

Acknowledgements. I am grateful to Simon L. Peyton Jones for the helpful discussions at the IFL'97 workshop and Gabriele Keller for comments on the paper. Furthermore, I am happy to thank the anonymous referees for their helpful comments.

References

- [1] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 1–12, 1995.
- [2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, 1995.
- [3] Manuel M. T. Chakravarty. Integrating multi-threading into the Spineless Tagless G-machine. In David N. Turner, editor, *Functional Programming, Glasgow 1995*, London, 1996. University of Glasgow, Springer-Verlag.

- [4] Manuel M. T. Chakravarty. *On the Massively Parallel Execution of Declarative Programs*. PhD thesis, Technische Universität Berlin, Fachbereich Informatik, 1997.
- [5] Manuel M. T. Chakravarty, Yike Guo, Martin Köhler, and Hendrik C. R. Lock. Goffin: Higher-order functions meet concurrent constraints. *Science of Computer Programming*, 30(1–2):157–199, 1998.
- [6] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM—a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, 1993.
- [7] Cormac Flanagan and Rishiyur S. Nikhil. pHluid: The design of a parallel functional language implementation on workstations. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 1996.
- [8] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.
- [9] Guang R. Gao, Lubomir Bic, and Jean-Luc Gaudiot, editors. *Advanced Topics in Dataflow Computing and Multithreading*. IEEE Computer Society Press, 1995.
- [10] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Enabling primitives for compiling parallel languages. In Boleslaw K. Szymanski and Balaram Sinharoy, editors, *Languages, Compilers and Run-Time Systems for Scalable Computers*, chapter 12, pages 153–168. Kluwer Academic Publishers, 1996.
- [11] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 1997. Submitted; revised version of [10].
- [12] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.
- [13] Herbert H.J. Hum et al. A design study of the EARTH multiprocessor. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 59–68. ACM Press, 1995.
- [14] David A. Kranz, Robert H. Halstead, and Eric Mohr. Mul-T: A high-performance parallel lisp. In *ACM SIGPLAN'89 Conference on Programming Languages Design and Implementation*, pages 81–90. ACM Press, 1989.
- [15] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In Bob Borchers, editor, *Proceedings of the Supercomputing '93 Conference*, pages 262–273. IEEE Computer Society Press, November 1993.
- [16] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.
- [17] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2), 1992.
- [18] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 1998. To appear.
- [19] P. W. Trinder, K. Hammond, J. S. Mattson Jr, A. S. Partridge, and S. L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Proceedings of Programming Languages Design and Implementation*, 1996.

DREAM: The DistRibuted Eden Abstract Machine^{*}

Silvia Breiting¹, Ulrike Klusik¹, Rita Loogen¹, Yolanda Ortega-Mallén², and Ricardo Peña²

¹ Philipps-Universität Marburg, D-35032 Marburg, Germany
{breiting,klusik,loogen}@mathematik.uni-marburg.de

² Universidad Complutense de Madrid, E-28040 Madrid, Spain
{yolanda,ricardop}@eucmax.sim.ucm.es

Abstract. Eden is being implemented by extending the Glasgow Haskell Compiler (GHC) which is based on the Spineless Tagless G-Machine (STGM). In this paper we present a parallel abstract machine which embodies the STGM for sequential computations and describes a distributed runtime environment for Eden programs on an abstract level.

1 Introduction

The parallel functional programming language Eden [7, 6] extends the lazy functional language Haskell [18] by syntactic constructs for *explicitly* defining processes. These processes communicate by exchanging values via communication channels modelled by head-strict lazy lists. Communication in Eden is asynchronous and *implicit*, i.e. exchange of messages is done automatically without the need for explicit send or receive commands in the program text. Eden's co-ordination language provides explicit control over process granularity and communication topology.

In this paper we will show how a distributed implementation of Eden can be developed in the same modular way as the language definition. In contrast to the prototype implementation described in [3], the distributed implementation presented in this paper incorporates a parallel runtime system that is specifically tailored for Eden. It follows the lines of other implementations of functional languages: full Eden is first desugared and translated into a core language called PEARL (Parallel Eden Abstract Reduction Language), then this language is conceptually interpreted by an abstract distributed machine called DREAM (DistRibuted Eden Abstract Machine). In fact, PEARL is finally compiled to C code with calls to the standard library MPI (*Message Passing Interface*). An important design decision for the present implementation has been to use the Glasgow Haskell compiler¹ (GHC) as a basis, and to reuse of it as much as possible to save development effort and to achieve high efficiency. In order to meet this

^{*} Work partially supported by the German-Spanish Acción Integrada HA1996-147 and the Spanish projects CAM-06T/033/96 and CICYT-TIC97-0672.

¹ see <http://www.dcs.gla.ac.uk/fp/software/ghc>

goal, we have defined PEARL as an extension of the STGL (Spineless Tagless G-machine Language) [17] underlying the GHC, and DREAM as an extension of the STGM (Spineless Tagless G-Machine).

Overview of the paper: We start with a short summary of the language Eden in Section 2. In the following section we introduce PEARL, the language on which the machine operates. In Section 4 the DREAM machine is presented and specified as a transition system. A detailed comparison with related work can be found in Section 5.

2 Eden

In this section, we will present the language Eden very briefly. For a more detailed discussion, the interested reader is referred to [7] and [6].

Process abstractions define abstract schemes for processes. A process that maps inputs in_1, \dots, in_m to outputs out_1, \dots, out_n can be specified by:

$$\text{process } (in_1, \dots, in_m) \rightarrow (out_1, \dots, out_n) \\ \text{where } equation_1 \dots equation_r$$

The optional **where** part of this expression is used to define auxiliary functions and common subexpressions which occur within the process definition. A process can have as input (respectively, output) tuples of channels and data structures of channels².

Process instantiations create processes and the corresponding channels. In such an expression, a process abstraction p is applied to a tuple of input expressions, yielding a tuple of outputs. The child process uses n independent threads of control in order to produce these outputs. Likewise, the parent process creates m additional threads that evaluate $input_exp_1, \dots, input_exp_m$:

$$(out_1, \dots, out_n) = p \# (input_exp_1, \dots, input_exp_m)$$

Communication channels are unidirectional and connect one writer to exactly one reader. Only fully evaluated data objects are communicated and lists are transmitted in a *stream*-like fashion, i.e. piecemeal. A concurrent thread will be suspended if it tries to access input that is not yet available.

Dynamic reply channels can be used in order to establish new connections at runtime. A process may generate a new input channel and send a message containing the name of this new channel to another process. Such a new input channel $chan$ and a reference ch_name to it are defined by the following expression:

$$\text{new } (ch_name, chan) \text{ exp}$$

A process that has received ch_name can use it for output³ by:

$$ch_name ! * expression_1 \text{ par } expression_2$$

² The latter case is not handled in this paper, see the separate paper [4].

³ For the channels to connect unique writers to unique readers, it is checked at runtime that no other process has connected to this channel before.

The sorting net unfolds only in case the input list has at least two elements. If we had lifted the process instantiations to the top level by an equation of the following form, an infinite process system would have been produced:

```
out = merger # (sortNet # 11, sortNet # 12)
      where (11,12) = unshuffle list
```

The design of process systems completely lies in the responsibility of the programmer. The granularity of processes as well as the communication topology is explicitly defined in Eden programs. Sequential computations are embedded within processes.

3 PEARL

PEARL is an extension of the core language STGL of the Glasgow Haskell compiler for the kernel constructs of Eden. Figure 1 shows the syntax of STGL as it has been defined in [17]. An STGL program is a non-empty sequence of bindings of variables to lambda forms. A lambda form $vars_f \backslash \pi \ vars_a \rightarrow expr \in Lfs$ is a lambda abstraction $\backslash vars_a \rightarrow expr$ which additionally contains information about the free variables $vars_f \subseteq Var$ and a flag π which indicates whether the lambda form, or to be precise, a closure with this lambda form, must be updated after its evaluation or not. The body expressions of lambda forms may be literals, applications, local definitions or case expressions. In the following, we denote the set of STGL-expressions by $Expr$, the set of alternatives of **case** expressions by $Alts$ and the set of typed data constructors by Γ .

The PEARL extensions (see Figure 1) concern bindings, lambda forms, expressions and primitive operators. In addition to the normal binding of STGL we introduce new bindings which define process instantiations. *Process instantiations* will only occur in the binding part of **let** and **letrec** expressions.

Lambda forms in PEARL can be either *process abstractions* for the definition of processes or STGL lambda forms defining functions. Process abstractions are treated as special lambda forms with the tag **process** and a process body that evaluates to an expression $\{expr_1 || \dots || expr_k\}$ defining parallel threads for the process outputs. The two lists of variables $vars_f$ and $vars_a$ contain the free variables of the process body and the imports of the process abstraction, respectively.

As process instantiations and abstractions are only handled as bindings and lambda forms, and not as special expressions, new expressions are merely introduced for dynamic channel handling. PEARL contains **new** and **par** expressions for the creation and use of dynamic channels. We use three new *primitive operations* in order to manage the sending of evaluated values via channels. The function **sendVal** is used for sending a single value, **sendHead** sends the next value of a stream and **closeStrm** closes a stream. These form the basis of the predefined function **sendChan** with type $a \rightarrow ()$ shown below, which transmits values on a channel.

$prog \rightarrow binds$	
$binds \rightarrow bind_1; \dots; bind_n$	$n \geq 1$
$bind \rightarrow var = lf$	
$lf \rightarrow vars_f \backslash \pi \ vars_a \rightarrow expr$	
$\pi \rightarrow u \mid n$	
$expr \rightarrow literal$	
$ \ constr \ atoms$	constructor application
$ \ primOp \ atoms$	primitive application
$ \ var \ atoms$	application
$ \ let \ binds \ in \ expr$	local definition
$ \ letrec \ binds \ in \ expr$	local recursive definition
$ \ case \ expr \ in \ alts$	case expression
$vars \rightarrow \{var_1, \dots, var_n\}$	$n \geq 0$, variable list
$atoms \rightarrow \{atom_1, \dots, atom_n\}$	$n \geq 0$, atom list
$atom \rightarrow var \mid literal$	
$literal \rightarrow 0\# \mid 1\# \mid \dots$	primitive integers
$primOp \rightarrow +\# \mid -\# \mid *\# \mid \dots$	primitive operations
$alts \rightarrow calt_1; \dots; calt_r; default$	$r \geq 0$, algebraic alternatives
$ \ palt_1; \dots; palt_n; default$	$n \geq 0$, primitive alternatives
$calt \rightarrow constr \ vars \rightarrow expr$	
$palt \rightarrow literal \rightarrow expr$	
$default \rightarrow var \rightarrow expr$	
$ \ default \rightarrow expr$	
<hr/>	
$bind \rightarrow vars_o = var_p \# \{var_1 \mid \dots \mid var_n\}$	process instantiation
$lf \rightarrow vars_f \backslash \mathbf{process} \ vars_a \rightarrow pBody$	process abstraction
$pBody \rightarrow letrec \ binds \ in \ parExpr$	process body
$ \ parExpr$	
$parExpr \rightarrow \{expr_1 \mid \dots \mid expr_k\}$	$k \geq 1$
$expr \rightarrow \mathbf{new} \ (var_n, var_c) \ expr$	dynamic channel
$ \ var_n \ !* \ expr_1 \ \mathbf{par} \ expr_2$	handling
$primOp \rightarrow \mathbf{sendVal} \mid \mathbf{sendHead} \mid \mathbf{closeStrm}$	primitives for communication

Fig. 1. Syntax of STGL (above) and PEARL extensions (below)

```

sendChan l = case l of {Nil          -> closeStrm;
                        Cons {y,ys} -> force y 'seq' sendHead y 'seq'
                                      sendChan ys;
                        default      -> force l 'seq' sendVal l }

```

`sendChan` uses a polytypic function `force :: a -> a` to reduce expressions to normal form. The function `seq`, as predefined in Haskell 1.3 onwards, evaluates its first argument and then returns its second argument.

```

force x = force' x 'seq' x
force' (C a1 ... an) = force a1 'seq' ... 'seq' force an 'seq' ()

```

The monomorphic instances of `force` will be derived by the compiler for any data type that may be sent. The compiler will also insert an application of `sendChan` to the expressions to be evaluated by a thread.

PEARL has been chosen to be as close as possible to STGL in order to reuse the STGM for the sequential code of Eden. Note that the only new constructions to be taken care of by DREAM are process abstractions, instantiations, the constructs `new` and `par` as well as the new primitives `sendVal`, `sendHead` and `closeStrm`.

Compiling Eden into PEARL: We extend the transformation of Haskell into STGL by rules for transforming the Eden constructs into the corresponding PEARL expressions. Only process abstractions and instantiations are transformed in a special way. The sending of data is made explicit in PEARL by inserting calls of the primitive function `sendChan` at the positions of output expressions in process abstractions and instantiations. Moreover multithreading is indicated by the construct $\{\dots || \dots || \dots\}$.

Process abstractions are transformed into special lambda forms with the update flag `\process`. The transformation ensures that lambda forms, and thus process abstractions, can only appear as right hand sides of bindings. A process abstraction of the following form

$$\text{process } (var_1, \dots, var_n) \rightarrow (out_1, \dots, out_m) \text{ where equations}$$

is translated into

$$\begin{aligned}
&\text{let } v_{new} = \{frees\} \setminus \text{process } \{var_1, \dots, var_n\} \rightarrow \\
&\quad \text{letrec equations}' \\
&\quad \quad v_{new1} = out'_1 \\
&\quad \quad \vdots \\
&\quad \quad v_{newm} = out'_m \\
&\quad \text{in } \{\text{sendChan } v_{new1} || \dots || \text{sendChan } v_{newm}\} \\
&\text{in } v_{new}
\end{aligned}$$

where the v_{newj} are new variables and out'_j and $equations'$ are the transformed output expressions and equations, respectively.

As process instantiations are special bindings in PEARL, they are abstracted out of expressions and put into local declarations (which float outwards in the optimization passes used in the GHC). An instantiation of the above process abstraction:

$$pabs \# (in_1, \dots, in_n)$$

is transformed into

```

let  $v_{new1} = in'_1$ 
   $\vdots$ 
   $v_{newn} = in'_n$ 
in let  $v_{inp1} = \text{sendChan } v_{new1}$ 
   $\vdots$ 
   $v_{inpn} = \text{sendChan } v_{newn}$ 
in let  $(newOut_1, \dots, newOut_m) = pabs \# \{v_{inp1} || \dots || v_{inpn}\}$ 
in  $(newOut_1, \dots, newOut_m)$ 

```

where the v_{newi} , v_{inpi} , and $newOut_k$ are new variables and in'_i are the transformed input expressions of the child process.

In order to pass Eden programs through the GHC front end, all Eden constructs have been embedded into Haskell using dummy functions which impose the correct types on them. Haskell's type inference will thus check the type consistency of process abstractions and instantiations. Only for the extended bindings in process bodies additional rules have been inserted in the compiler front end passes. The transformation of the Eden constructs into PEARL is done at the end of the front end passes.

Example: The `sortNet` example shown in Section 2 will be translated into the following PEARL program where the bindings for `force` and `sendChan` are provided in the global environment:

```

sortNet = {sendChan, sortNet} \process {list} ->
  {let merger = {sendChan} \process {s1,s2} ->
    {letrec smerge = ... -- STGL translation
      in let oute = {smerge,s1,s2} \u {} -> smerge {s1,s2}
      in sendChan {oute} }
  in letrec unshuffle = ... -- STGL-translation
  in letrec sort
    = {sendChan,merger,unshuffle,sort} \n {xs} ->
    case xs of
    ... -- case cascade
  default ->
    case unshuffle {xs} of
    {(l1,l2)} ->
      let inp1 = {sendChan,l1} \u {} -> sendChan {l1}
      inp2 = {sendChan,l2} \u {} -> sendChan {l2}
      in let {l1'} = sortNet # {inp1}
      {l2'} = sortNet # {inp2}

```



```

in let s11 = {sendChan,11'} \u {} -> sendChan {11'}
      s12 = {sendChan,12'} \u {} -> sendChan {12'}
in let {res} = merger # {s11,s12}
in res      }      }
in let sorted = {sort,list} \u {} -> sort {list}
in sendChan {sorted} }

```

In the next section we will present the operational semantics of PEARL, which illustrates on an intermediate level the execution of Eden programs in a distributed machine.

4 DREAM

A parallel DREAM computation is performed by a system of extended STGM instances, so-called *Dreams*, each of which represents an Eden process. Internally, a Dream maintains multiple flows of control which implement the threads producing independent outputs in Eden processes. Every machine instance uses its own local heap which contains local data structures as well as references to buffers for receiving inputs. Special `queueMe` closures are used to suspend threads that try to read inputs which are not yet available. This way of thread synchronisation is common in distributed abstract machines. DREAM differs substantially from other abstract machines used in the implementation of parallel functional languages. The details of these differences are discussed in Section 5 of the paper.

The set of the *Dreams*' states is called *DreamState*. The state of the whole DREAM is represented by a finite mapping of the set *Process_Id* of process numbers to *DreamState*. The transitions are denoted by \Rightarrow and define a binary relation on the set of DREAM states:

$$\text{DREAM} = \langle \text{Process_Id} \xrightarrow{\text{finite}} \text{DreamState}, \Rightarrow \rangle,$$

where $\text{Process_Id} \stackrel{\text{def}}{=} \mathbb{N}$. A detailed specification and explanation of the STGM can be found in [17]. The sequential transitions of the STGM remain unchanged in the parallel context. They only have to be adapted to the extended state. In the following, we will show what has to be added to the STGM in order to express the full range of parallel computations that can be programmed in Eden.

4.1 Extension of the STGM

Within a Dream, multithreading is used to concurrently compute the different outputs of the corresponding process. There is a one to one correspondence between machine instances and processes and between threads and outputs.

The state of a Dream contains for each thread the following components, where *Int* is a set of tagged integer values, *Addr* is a set of tagged heap addresses, *Val* := *Int* \cup *Addr*, and *Env* := [*Var* $\rightarrow_{\text{finite}}$ *Val*] is the set of environments:

code	$c \in Instr$, where $Instr$ is defined below,
argument stack	$as \in Val^*$,
return stack	$rs \in Cont^*$, where $Cont = Alts \times Env$
update stack	$us \in Frame$, where $Frame = Val^* \times Cont^* \times Addr$
outport (list)	$out \in (Process_Id \times Channel_Id)^*$

and, in common for all threads:

heap	$h : Addr \rightarrow Closures$, where $Closures = Lfs \times Val^*$,
global environment	$\sigma : Var \rightarrow Addr$,
import table	$ipt : Channel_Id \rightarrow Addr \times Process_Id$,
counter of blocked threads	$b \in Int$.

The first four components of a thread state as well as the heap and the global environment exactly correspond to those of the STGM state except that we extend the underlying sets of expressions (PEARL expressions), lambda forms and closures (see below).

The *outport* component within a thread state contains information about the connection to an import of another process instance which will be fed by the thread. An outport specification is a pair (m, c) where $m \in Process_Id$ is the identifier of a remote Dream and $c \in Channel_Id$ is a channel identifier in this remote machine. It denotes the destination of the output produced by the thread. In general each thread contains exactly one outport specification. There are two exceptions: The very first thread of a process which evaluates the process body before spawning the threads contains the *list of all outports* communicating with the parent process. The thread evaluating the main expression has *no* outport.

The *import table* ipt maps channel identifiers to the respective heap addresses, where received messages are stored. Additionally, the sender's id is logged here to be able to check the 1:1 correspondence of imports to outports. The heap addresses point to `queueMe` closures which represent not yet received messages.

The *counter of blocked threads* is introduced to detect the termination of processes. A process terminates when the set of active threads is empty and there are no blocked threads in the heap.

The *instructions* are the same as those of the STGM:

$$\begin{aligned}
Instr = \{ & Eval \quad e \ \rho \quad | \ e \in Expr, \rho \in Env \} \\
& \cup \{ Enter \quad a \quad | \ a \in Addr \} \\
& \cup \{ ReturnCon \ (C \ w_1 \dots w_n) \ | \ C \in \Gamma, w_i \in Val \} \\
& \cup \{ ReturnInt \ k \quad | \ k \in Int \}
\end{aligned}$$

with the following intuitive meaning: *Eval* $e \ \rho$ evaluates expression e in the environment ρ and applies its value to the arguments on the argument stack. *Enter* a applies the closure at address a to the arguments on the argument stack. *ReturnCon* $c \ ws$ returns the constructor c applied to values ws to the continuation on the return stack. *ReturnInt* k returns the primitive integer k to the continuation on the return stack.

The set of heap closures is extended by a closure **queueMe** of the form:

$$\{\} \setminus n \{\} \rightarrow \text{queueMe } q$$

which causes the suspension of a thread when entered, appending its state (code, argument stack, return stack, update stack, output) to the associated queue q . In the following, we will represent this closure as a pair $\langle \text{queueMe}, [t_1, \dots, t_n] \rangle$ where the t_i are thread states. This special closure is used to mark for stream channels the write end where new incoming values must be appended and for a one-value channel the position for storing the value. Moreover, it is used to prevent the simultaneous evaluation of updatable closures by multiple threads.

We introduce **ChanName** closures $\{\} \setminus n \{\} \rightarrow \text{ChanName } (pid, cid)$ to represent the names of dynamic channels, where (pid, cid) is an output specification.

The initial state of the system consists of a unique Dream with process id 0 embodying a thread for evaluating the main expression. Its local heap h_0 contains a closure for each globally defined variable, and the global environment σ_0 binds each global variable to the heap address of its closure. The input channel table is empty and there are no blocked threads. The initial thread has empty stacks denoted by ε and no output specification.

$\langle 0 \mapsto ((\text{Eval } \text{main } \{\} \rho_\emptyset) \ \varepsilon \ \varepsilon \ \varepsilon \ \varepsilon) \ h_0 \ \sigma_0 \ [] \ 0 \rangle$

4.2 DREAM Transitions

The execution of the code for a thread may cause changes not only to the thread itself, but also to the whole process, i.e. the shared components of the corresponding Dream. Moreover, there are evaluations which cause interactions with other Dreams, thus affecting the state of the whole system; e.g. process instantiation or communication. In the specifications we adopt the notation used in [17]. We only show the states of the machine instances involved in the transitions and use the symbol \parallel to separate concurrent threads and $|||$ to separate parallel Dream instances. The length of a sequence xs is denoted by $|xs|$.

We will present the transitions in the following order: We start with communication, which is performed between threads of different instances, afterwards explain the interaction of threads in the same instance and finally explain process management.

Communication: In PEARL, data is communicated using the primitive functions **sendHead**, **closeStrm** and **sendVal**.

Sending and receiving a single value: The output in a thread's state contains a remote inport address consisting of a process instance number and a channel identifier. The remote inport table maps the channel identifier to a **queueMe**-closure address and a sender id, which in the case of a dynamically created channel will initially be unknown (\perp). The suspended threads collected in the **queueMe**-closure are reactivated, the counter of blocked threads is decremented accordingly, and the closure is updated with the transmitted closure using the

auxiliary function *graph_copy*. In fact, the whole subheap which can be referenced by the sent closure must be transmitted and embedded into the receiver's heap.

$$\begin{array}{l}
 \langle i \mapsto ((\text{Eval } (\text{sendVal } v) \rho) \varepsilon \varepsilon (j, c)) \parallel \text{threads}_i \ h_i \ \sigma_i \ \text{ipt}_i \ b_i \rangle \\
 ||| \ \langle j \mapsto \text{threads}_j \ h_j[a_{in} \mapsto \langle \text{queueMe}, ts \rangle] \ \sigma_j \ \text{ipt}_j[c \mapsto (a_{in}, s)] \ b_j \rangle \\
 \text{with } s \in \{\perp, i\} \\
 \Rightarrow \\
 \langle i \mapsto \text{threads}_i \ h_i \ \sigma_i \ \text{ipt}_i \ b_i \rangle \\
 ||| \ \langle j \mapsto (ts \parallel \text{threads}_j) \ h'_j \ \sigma_j \ \text{ipt}_j[c \mapsto \perp] \ b_j - |ts| \rangle \\
 \text{where } h'_j = \text{graph_copy}(h_j, a_{in}, \text{subHeap}(h_i, a), a) \\
 \text{Addr } a = \text{val } \sigma \ \rho \ v
 \end{array} \tag{C.1}$$

graph_copy(h_1, a_1, h_2, a_2) creates a copy of heap h_2 with new unique addresses within another heap h_1 starting at address a_1 where the node $h_2(a_2)$ is inserted. To determine the subheap of a heap h which contains all nodes reachable from a list of node addresses ws we use the function *subHeap*. Details are omitted.

The sending thread terminates after completing its task. The receiving process adds the value to its heap and deletes the inport from the inport table.

Sending and receiving a stream element: Sending a stream of values is similar to sending a single value except that the sending of values and the termination of the thread are done by separate primitive functions.

$$\begin{array}{l}
 \langle i \mapsto ((\text{Eval } (\text{sendHead } v) \rho) \varepsilon (\text{default} \rightarrow \text{cont}, \tilde{\rho}) : rs \varepsilon (j, c) \\
 \parallel \text{threads}_i \ h_i \ \sigma_i \ \text{ipt}_i \ b_i \rangle \\
 ||| \ \langle j \mapsto \text{threads}_j \ h_j[a_{in} \mapsto \langle \text{queueMe}, ts \rangle] \ \sigma_j \ \text{ipt}_j[c \mapsto (a_{in}, s)] \ b_j \rangle \\
 \text{with } s \in \{\perp, i\} \\
 \Rightarrow \\
 \langle i \mapsto ((\text{Eval } \text{cont } \tilde{\rho}) \varepsilon rs \varepsilon (j, c)) \parallel \text{threads}_i \ h_i \ \sigma_i \ \text{ipt}_i \ b_i \rangle \\
 ||| \ \langle j \mapsto (ts \parallel \text{threads}_j) \ h'_j \ \sigma_j \ \text{ipt}_j[c \mapsto (a_t, i)] \ b_j - |ts| \rangle \\
 \text{where} \\
 h'_j = \text{graph_copy}(h_j[a_{in} \mapsto \langle \text{Cons } \{a_v, a_t\} \rangle, a_t \mapsto \langle \text{queueMe}, [] \rangle], \\
 \qquad \qquad \qquad a_v, \text{subHeap}(h_i, a), a) \\
 \text{Addr } a = \text{val } \sigma \ \rho \ v \\
 a_v \text{ and } a_t \text{ are new heap addresses}
 \end{array} \tag{C.2}$$

Streams are stored in the receiving process heap as normal lists, the only difference being that a *queueMe*-closure marks the end of the sequence of values received up to now. This closure will be rewritten when a new message for this stream arrives. Thus, sending a stream element updates the inport's closure with a new closure representing a list, the head of which points to the transmitted subheap and the tail of which is a new *queueMe*-closure with an empty list of suspended threads. The inport table is updated with the address of the new *queueMe*-closure. The sending thread continues its computation with the continuation given on top of the return stack. In fact, the continuation is the evaluation and sending of the remaining list via the channel.

Closing a channel: The closing of a channel implies the termination of the corresponding thread. On the receiver's side the threads depending on the import are reactivated, and the **queueMe**-closure is overwritten with a **Nil** closure marking the end of the list associated to the stream.

$$\begin{array}{l}
 \langle i \mapsto ((\text{Eval } \mathbf{closeStrm} \ \rho) \ \varepsilon \ \varepsilon \ \varepsilon \ (j, c)) \parallel \text{threads}_i \ h_i \ \sigma_i \ \text{ipt}_i \ b_i \rangle \\
 ||| \langle j \mapsto \text{threads}_j \ h_j[a_{in} \mapsto \langle \mathbf{queueMe}, ts \rangle] \ \sigma_j \ \text{ipt}_j[c \mapsto (a_{in}, s)] \ b_j \rangle \\
 \Rightarrow \\
 \langle i \mapsto \text{threads}_i \ h_i \ \sigma_i \ \text{ipt}_i \ b_i \rangle \\
 ||| \langle j \mapsto (ts \parallel \text{threads}_j) \ h_j[a_{in} \mapsto \langle \mathbf{Nil} \rangle] \ \sigma_j \ \text{ipt}_j[c \mapsto \perp] \ b_j - |ts| \rangle
 \end{array} \quad (\text{C.3})$$

Dynamic reply channels: The creation of a dynamic channel means the creation of a new import, i.e. the import table is extended by an entry which contains the address of a newly allocated **queueMe**-closure. As the sender to this import is not yet known, \perp is noted as sender id. When the first value of a stream is sent, \perp will be overwritten by the actual sender's id. In addition a **ChanName**-closure is included in the heap which contains the machine instance number and the new channel identifier.

$$\begin{array}{l}
 \langle i \mapsto ((\text{Eval } \mathbf{new} \ (chn, ch) \ e \ \rho) \ \text{as } rs \ us \ outp) \parallel \text{threads} \ h \ \sigma \ \text{ipt} \ b \rangle \\
 \Rightarrow \\
 \langle i \mapsto ((\text{Eval } e \ \rho[chn \mapsto \text{Addr } achn, ch \mapsto \text{Addr } ach]) \ \text{as } rs \ us \ outp) \\
 \parallel \text{threads} \ h' \ \sigma \ \text{ipt}' \ b \rangle \\
 \text{where } \text{ipt}' = \text{ipt}[c \mapsto (ach, \perp)] \\
 h' = h[achn \mapsto \langle \mathbf{ChanName} \ (i, c) \rangle, ach \mapsto \langle \mathbf{queueMe}, [] \rangle] \\
 achn \text{ and } ach \text{ are new heap addresses,} \\
 c \text{ is a new channel identifier}
 \end{array} \quad (\text{D.1})$$

When using a dynamic channel the current thread is split into two: one will continue with the evaluation of the main expression, and the other will be responsible for the new output.

$$\begin{array}{l}
 \langle i \mapsto ((\text{Eval } v! * e_1 \ \mathbf{par} \ e_2 \ \rho) \ \text{as } rs \ us \ outp) \parallel \text{threads} \ h \ \sigma \ \text{ipt} \ b \rangle \\
 \text{with } \text{val } \rho \ \sigma \ v = \text{Addr } a \text{ and } h(a) = \langle \mathbf{ChanName} \ out_{dyn} \rangle \\
 \Rightarrow \\
 \langle i \mapsto ((\text{Eval } e_1 \ \rho) \ \varepsilon \ \varepsilon \ \varepsilon \ out_{dyn}) \parallel \\
 ((\text{Eval } e_2 \ \rho) \ \text{as } rs \ us \ outp) \parallel \text{threads} \ h \ \sigma \ \text{ipt} \ b \rangle
 \end{array} \quad (\text{D.2})$$

Organizing Multithreading

Entering an updatable closure: In order to prevent multiple threads from simultaneously evaluating the same closure, an updatable closure is overwritten with a **queueMe**-closure on first entering it.

$$\begin{aligned}
& \langle i \mapsto ((\text{Enter } a) \text{ as } rs \text{ us } outp) \parallel threads \ h \ \sigma \ ipt \ b \rangle \\
& \quad \text{with } h(a) = \langle (vs \setminus u \ \{ \} \rightarrow e) \ ws_f \rangle \\
& \Rightarrow \\
& \langle i \mapsto ((\text{Eval } e \ [vs/ws_f]) \ \varepsilon \ \varepsilon \ (as, rs, a) : us \ outp) \parallel threads \ h' \ \sigma \ ipt \ b \rangle \\
& \quad \text{where } h' = h[a \mapsto \langle \text{queueMe}, [] \rangle]
\end{aligned} \tag{M.1}$$

Updating a closure: When the **queueMe**-closure is finally updated, the suspended threads are reactivated. This is the only difference between the following rules and the corresponding sequential ones. Updates are triggered in two cases. Firstly, when a *ReturnCon* instruction is executed with an empty return stack. Secondly, when a closure is entered for which there are not enough addresses on the argument stack (partial application).

$$\begin{aligned}
& \langle i \mapsto ((\text{ReturnCon } c \ ws) \ \varepsilon \ \varepsilon \ (as_u, rs_u, a_u) : us \ outp) \parallel ts \ h \ \sigma \ ipt \ b \rangle \\
& \quad \text{with } h(a_u) = \langle \text{queueMe}, bts \rangle \\
& \Rightarrow \\
& \langle i \mapsto ((\text{ReturnCon } c \ ws) \ as_u \ rs_u \ us \ outp) \parallel ts \parallel bts \ h' \ \sigma \ ipt \ b - |ts| \rangle \\
& \quad \text{where } h' = h[a_u \mapsto \langle vs \setminus n \ \{ \} \rightarrow c \ ws \rangle] \\
& \quad \quad vs \text{ is a list of distinct variables with } |vs| = |ws|
\end{aligned} \tag{M.2}$$

$$\begin{aligned}
& \langle i \mapsto ((\text{Enter } a) \ as \ \varepsilon \ (as_u, rs_u, a_u) : us \ outp) \parallel threads \ h \ \sigma \ ipt \ b \rangle \\
& \quad \text{with } h(a_u) = \langle \text{queueMe}, ts \rangle \\
& \quad \quad h(a) = \langle (vs \setminus n \ xs \rightarrow e) \ ws \rangle \\
& \quad \quad |as| < |xs| \\
& \Rightarrow \\
& \langle i \mapsto ((\text{Enter } a) \ as ++ as_u \ rs_u \ us \ outp) \parallel ts \parallel threads \ h' \ \sigma \ ipt \ b - |ts| \rangle \\
& \quad \text{where } xs_1 ++ xs_2 = xs \text{ with } |xs_1| = |as| \\
& \quad \quad f \text{ is an arbitrary variable} \\
& \quad \quad h' = h[a_u \mapsto \langle (f : xs_1) \setminus n \ \{ \} \rightarrow f \ xs_1 \ a : as \rangle]
\end{aligned} \tag{M.3}$$

Suspending a thread: When a thread enters a **queueMe**-closure, its state is saved within this closure, and the number of blocked threads is incremented.

$$\begin{aligned}
& \langle i \mapsto ((\text{Enter } a) \ as \ rs \ us \ outp) \parallel threads \ h \ \sigma \ ipt \ b \rangle \\
& \quad \text{with } h(a) = \langle \text{queueMe}, ts \rangle \\
& \Rightarrow \\
& \langle i \mapsto threads \ h' \ \sigma \ ipt \ (b + 1) \rangle \\
& \quad \text{where } h' = h[a \mapsto \langle \text{queueMe}, ((\text{Enter } a) \ as \ rs \ us \ outp) \parallel ts \rangle]
\end{aligned} \tag{M.4}$$

Process Instantiation: Now that we have presented the mechanisms used for message passing and for the interaction of threads in the same process, we can explain the creation of processes, which relies on both. Process instantiations only appear as special bindings in local declarations, i.e. *let* and *letrec* expressions. Thus, the rule for evaluating these expressions must be extended:

$$\begin{array}{l}
\langle i \mapsto ((\text{Eval } \mathbf{let}(\mathbf{rec}) \text{ binds in } e \ \rho) \text{ as } rs \text{ us } outp) \parallel \text{threads } h \ \sigma \text{ ipt } b \rangle \\
\Rightarrow \\
\langle i \mapsto ((\text{Eval } e \ \rho') \text{ as } rs \text{ us } outp) \parallel \text{threads } \parallel ts \ h' \ \sigma \text{ ipt}' \ b \rangle \\
||| \text{ newInsts} \\
\text{where } \rho' = \text{extend_env}(\rho, \text{binds}) \\
(ts, h', \text{ipt}', \text{newInsts}) = \text{handle_bindings}(i, \text{binds}, \rho', h, \sigma, \text{ipt})
\end{array} \quad (\text{P.1})$$

The function *extend_env* allocates space for closures and maps the variables on the left hand side of the bindings to the new heap addresses:

$$\begin{aligned}
&\text{extend_env}(\rho, \{\}) = \rho \\
&\text{extend_env}(\rho, \{x = lf; \text{bind}_2; \dots; \text{bind}_n\}) \\
&\quad = \text{extend_env}(\rho[x \mapsto \text{Addr } a], \{\text{bind}_2; \dots; \text{bind}_n\}) \\
&\quad \text{where } a \text{ is a new heap address} \\
&\text{extend_env}(\rho, \{o_1, \dots, o_m\} = p\#\{i_1 \parallel \dots \parallel i_k\}; \text{bind}_2; \dots; \text{bind}_n) \\
&\quad = \text{extend_env}(\rho[o_1 \mapsto \text{Addr } a_1, \dots, o_m \mapsto \text{Addr } a_m], \{\text{bind}_2; \dots; \text{bind}_n\}) \\
&\quad \text{where } a_1, \dots, a_m \text{ are new heap addresses}
\end{aligned}$$

The corresponding heap closures are produced by *handle_bindings*. If the bindings contain process instantiations, this function also produces new threads, further entries in the import table and new machine instances.

$$\begin{aligned}
&\text{handle_bindings}(i, \{\text{bind}_1; \dots; \text{bind}_n\}, \rho, h_0, \sigma, \text{ipt}_0) \\
&\quad = (\text{threads}_1 \parallel \dots \parallel \text{threads}_n, h_n, \text{ipt}_n, \text{newInst}_1 ||| \dots ||| \text{newInst}_n) \\
&\quad \text{where for } 1 \leq j \leq n: \\
&\quad (\text{threads}_j, h_j, \text{ipt}_j, \text{newInst}_j) = \text{handle_binding}(i, \text{bind}_j, \rho, h_{j-1}, \sigma, \text{ipt}_{j-1})
\end{aligned}$$

Rule (P.1) specifies that the current thread continues with the evaluation of the body expression *e* in the extended environment. The heap and import table are adjusted accordingly. The new threads *ts* run concurrently and the newly generated machine instances *newInsts* run in parallel.

The auxiliary function *handle_binding* distinguishes between two cases. The simple case of binding is the heap allocation of a closure as in the sequential STGM. Only the heap is modified. In this case the effect of the transition exactly corresponds to the sequential rule. Let *val* $\rho \ \sigma \ x = \text{Addr } a$. Then:

$$\begin{aligned}
&\text{handle_binding}(i, x = \text{vars}_f \setminus \pi \text{ vars}_a \rightarrow e_{\text{rhs}}, \rho, h, \sigma, \text{ipt}) = (\varepsilon, h', \text{ipt}, \varepsilon) \\
&\quad \text{where } h' = h[a \mapsto \langle (\text{vars}_f \setminus \pi \text{ vars}_a \rightarrow e_{\text{rhs}}) (\rho \text{ vars}_f) \rangle]
\end{aligned}$$

A process instantiation leads to the creation of new input channels, new threads for the evaluation of outputs and a new machine instance for the newly created process. Let *val* $\rho \ \sigma \ p = \text{Addr } a$ and $h(a) = \langle (xs \setminus \mathbf{process} \text{ is} \rightarrow e_{\text{body}}) \ ws \rangle$. Then:

$$\begin{aligned}
&\text{handle_binding}(i, os = p\#vs, \rho, h, \sigma, \text{ipt}) = (\text{newThs}, h', \text{ipt}', \text{newInst}) \\
&\quad \text{where } (\text{new}_i, \text{newInst}, \text{outs}) \\
&\quad = \text{create_process}(\langle (xs \setminus \mathbf{process} \text{ is} \rightarrow e_{\text{body}}) \ ws \rangle, h_{ws}, i, \text{ins}, \sigma)
\end{aligned}$$

$$\begin{aligned}
(ins, h', ipt') &= \text{create_inports}(i, \text{new}_i, os, \rho, h, ipt) \\
h_{ws} &= h_0 \cup \text{subHeap}(h, ws) \\
\text{newThs} &= \text{spawn_threads}(vs, \rho, outs)
\end{aligned}$$

In the state of the current machine instance, input channels are created by *create_inports* to receive messages from the outputs of the new process. The identifiers of these channels together with the parent id are passed as arguments to the function *create_process* that yields a new machine instance for the process to be created. The function *spawn_threads* creates for each inport of the newly created process a new thread which evaluates the associated expression.

The heap h_{ws} contains the static closures of the global environment in h_0 and all the relevant closures needed to correctly bind the free variables xs of the process abstraction as well as all closures reachable from these closures via variable bindings computed by *subHeap*⁴. It is used as the initial heap for the new machine instance. The auxiliary functions *create_inports*, *create_process* and *spawn_threads* are defined next.

create_inports creates new input channels. It allocates **queueMe**-closures in the heap and extends the environment by mapping the input variables in_i to the addresses of the new closures. Finally, it extends the inport table by mappings of new channel identifiers to the **queueMe**-closures. The updated state components and the list of the new channel names are returned as result. Let $\text{val } \rho \ \sigma \ in_j = \text{Addr } ai_j$ for $1 \leq j \leq n$. Then:

$$\begin{aligned}
&\text{create_inports}(i_{\text{recv}}, i_{\text{send}}, [in_1, \dots, in_n], \rho, h, ipt) \\
&= (([i_{\text{recv}}, c_1], \dots, [i_{\text{recv}}, c_n]), h', ipt') \\
&\text{where } h' = h[ai_1 \mapsto \langle \text{queueMe}, [] \rangle, \dots, ai_n \mapsto \langle \text{queueMe}, [] \rangle] \\
&\quad ipt' = ipt[c_1 \mapsto (ai_1, i_{\text{send}}), \dots, c_n \mapsto (ai_n, i_{\text{send}})] \\
&\quad c_1, \dots, c_n \text{ are new channel identifiers}
\end{aligned}$$

in *create_process* is used to define a new machine instance with a new identifier and a list of output specifications. The state of this new machine contains the threads for its outputs, the heap passed as a parameter extended by the inports of the process, the global environment and the new inport table. The threads for the outputs are spawned using *spawn_threads*.

The addresses of the corresponding inports (of the parent process) are passed as a parameter. Inport channels are created by *create_inports* to which the new outputs of the parent process will be connected.

$$\begin{aligned}
&\text{create_process}(\langle (xs \setminus \text{process } is \rightarrow e_{\text{body}}) ws \rangle, h_{ws}, i, outs, \sigma) \\
&= (\text{new}_i, [\text{new}_i \mapsto \text{state}_i], ins) \\
&\text{where } \rho_0 = [xs \mapsto ws, i_1 \mapsto \text{Addr } a_1, \dots, i_n \mapsto \text{Addr } a_n] \\
&\quad \text{state}_i = ((\text{Eval } e_{\text{body}} \ \rho_0) \ \varepsilon \ \varepsilon \ \text{outs}) \ h_{in} \ \sigma \ ipt_{in} \ 0 \\
&\quad (ins, h_{in}, ipt_{in}) = \text{create_inports}(\text{new}_i, i, is, \rho_0, h_{ws}, []) \\
&\quad \text{new}_i \text{ is a new process identifier,} \\
&\quad a_1, \dots, a_n \text{ are new heap addresses.}
\end{aligned}$$

⁴ Note that this heap must not contain any **queueMe**-closure. Otherwise the process instantiation will be suspended until this closure is overwritten.

spawn_threads takes a parallel expression of the form $\{e_1 \parallel \dots \parallel e_k\}$, an environment ρ , and a list of output specifications and generates threads. These evaluate the expressions in the given environment and send the results via the respective outputs.

$$\begin{aligned} \text{spawn_threads } (\{e_1 \parallel \dots \parallel e_k\}, \rho, [out_1, \dots, out_k]) &= t_1 \parallel \dots \parallel t_k \\ \text{where } t_i &= ((Eval\ e_i\ \rho) \ \varepsilon \ \varepsilon \ out_i), \forall i \in \{1, \dots, k\} \end{aligned}$$

The evaluation of a process body leads to a parallel expression which defines the threads for the outputs to the generator process. The single control thread for the creation of the heap and top level subprocesses terminates. k threads for the evaluation of the outputs are started.

$$\begin{aligned} &\langle i \mapsto ((Eval\ \{e_1 \parallel \dots \parallel e_k\}\ \rho) \ \varepsilon \ \varepsilon \ outs) \parallel threads\ h\ \sigma\ ipt\ b \rangle \\ &\Rightarrow \\ &\langle i \mapsto threads' \parallel threads\ h\ \sigma\ ipt\ b \rangle \\ &\quad \text{where } threads' = \text{spawn_threads}(\{e_1 \parallel \dots \parallel e_k\}, \rho, outs) \end{aligned} \tag{P.2}$$

Termination: A machine instance without any threads (active or blocked) terminates immediately. The active or blocked threads in other instances which fill its imports are removed using the auxiliary function *remove_thread*.

$$\begin{aligned} &\langle i \mapsto \emptyset\ h\ \sigma\ [c_1 \mapsto (j_1, a_1), \dots, c_k \mapsto (j_k, a_k)]\ 0 \rangle \\ &\parallel_{l=1}^k \langle j_l \mapsto state_l \rangle \\ &\Rightarrow \\ &\parallel_{l=1}^k \langle j_l \mapsto \text{remove_thread}(i, c_l)\ state_l \rangle \end{aligned} \tag{P.3}$$

where

$$\begin{aligned} &\text{remove_thread}\ outp\ ((instr\ as\ rs\ us\ outp) \parallel threads\ h\ \sigma\ ipt\ b) \\ &= (threads\ h\ \sigma\ ipt\ b) \\ &\text{remove_thread}\ outp\ (threads\ h[a \mapsto \langle \text{queueMe}, (instr\ as\ rs\ us\ outp) \parallel threads' \rangle] \\ &\quad \sigma\ ipt\ b) \\ &= (threads\ h[a \mapsto \langle \text{queueMe}, threads' \rangle] \ \sigma\ ipt\ (b - 1)) \end{aligned}$$

Nondeterminism: The predefined **merge** process passes values from its incoming channels to its outgoing channel. We omit the straightforward definition of this primitive process.

Summary: This completes the specification of the DREAM machine. It adds 5 new transition rules for communication (C.1 – C.3, D.1, D.2), modifies the rules for entering an updatable closure (M.1) and for updating closures (M.2, M.3) and introduces a new rule for thread suspension (M.4), it extends the evaluation of *letrec*-expressions by process instantiations (P.1) and finally adds rules for thread spawning (P.2) and process termination (P.3). In summary, eight transition rules have been added and four rules have been modified in order to build up a parallel system on top of the sequential STGM [17].

5 Related Work

In the last decade parallel functional programming has been an active field of research [11]. Three main directions of research can be distinguished: the exploitation of implicit parallelism, the incorporation of parallel operations and the integration of functional and concurrent programming.

Implicit parallelism in functional programs is usually exploited by annotating expressions that are worthwhile to be executed remotely. Such annotations can be introduced by a parallelizing compiler or by the programmer. They are semantically transparent, i.e. the semantics of a program with annotations is identical to the semantics of the original program. Typical examples are para-functional programming[13], Concurrent Clean[15] and Glasgow Parallel Haskell[22].

Annotated expressions are handled by defining “sparks” (i.e. putting their heap addresses into a work pool), or by adding new threads for their evaluation to the task list. Sparked expressions and threads can be evaluated either locally or remotely, depending on the load balancing algorithm and the current work load. Accordingly, most of the parallel abstract machines for functional languages are based on a low level “fork and wait” mechanism: expressions are spawned for parallel evaluation. Threads which require the results of spawned expressions have to wait for the results to be sent back, see for example [22, 15, 8, 12]. They support a global address space and provide a virtual shared memory. In some implementations, e.g. [22], a request for the remote data must be sent, because results of parallel expressions are not automatically returned to their original location, as e.g. in [15]. While waiting for the data, the evaluation continues with another active thread or a new spark. If the whole evaluation is demand-driven, it is sufficient to use a non-preemptive scheduler.

DREAM differs substantially from these abstract machines, because there is no need for a virtual shared memory. Processes have no direct access to data of other processes. They are closed entities and global data must be explicitly communicated to them. In particular this implies that there is no need for a global memory management. Of course the process interface contains references to other processes, but these are explicit and set up a well-defined communication structure. As communication channels are one-to-one, it is known which data is received by which processor and consequently data can be directly sent there without waiting for a request. We expect this direct way of data exchange to reduce communication costs considerably. Before expressions are transmitted, they are evaluated to normal form. This simplifies the low level data transmission and supports granularity control as it is determined where expressions are evaluated. A fair scheduler is however mandatory for the thread management in Dreams.

Special parallel constructs are provided in data-parallel languages like NESL [5], SISAL [21] and pH [16] or skeleton-based coordination languages like SCL [9]. These languages encapsulate parallelism by providing distributed data structures and predefined parallel operations on these structures. A general process-oriented view of parallelism is not supported. NESL for instance supplies parallel sequences (arrays) and data-parallel operations like apply-to-each (map). The structured

coordination language SCL is based on skeletons, i.e. higher order functional forms with built-in parallel behaviour. These are naturally data-parallel and express data partitioning, placement and movement as well as control flow on a high level of abstraction. As in these languages parallelism is restricted to predefined operations and skeletons, they can be implemented by adding parallel primitives to a sequential implementation. The development of a specific parallel abstract machine is not required. NESL is compiled to VCode, a small stack-based intermediate language with highly optimized functions which operate on sequences of data [2] and SCL is compiled to Fortran plus MPI.

A coordination language which is closer to Eden is *Caliban*[14]. In correspondence with Eden, one can define networks of processes which communicate via head-strict lazy streams. The main idea in both approaches is to make the communication structure of a process system explicit in order to map it directly on a distributed memory multiprocessor. Unlike Eden, Caliban is restricted to static process systems which must be configured at compile-time. Only basic values can be exchanged between processes and each process has only one output stream. There are no special constructs for reactive systems.

Caliban provides annotations for partitioning a program into a process net. They can be defined by functions, which are partially evaluated during compile time until the annotations are in primitive form. The Caliban compiler is based on the sequential Haskell compiler of the FAST project[10]. Similarly to our approach, the specific Caliban extensions have been integrated into the base language (Haskell) in order to use as much of the machinery of the base language as possible. Due to the imposed restrictions the Caliban compiler can extract the whole process net information at compile time and translate it into a call to a special system primitive called `procnet` which implements the run-time parallelism. The resulting standard functional program is then compiled like any other for each processor element. The implementation of the `procnet` function which is called first on each processor element sets up the process network to start computing and afterwards controls interprocessor communication. Each process is evaluated eagerly by its processor element. Its evaluation can be blocked by an inter-processor data dependency or when the output buffer space is used up. Like in Eden, buffering is provided naturally by the heap.

Because Eden supports dynamic process creation and several outputs of processes, it is not possible to implement process creation and communication by a single primitive function with an interface to a purely sequential runtime system.

Most *concurrent* functional languages like Facile [23], Concurrent ML [20], Erlang[1] and Concurrent Haskell [19] provide low level parallel extensions to functional languages, which are implemented on a shared memory base using fair schedulers to switch between the concurrent activities. Distributed implementations are feasible, but require the implementation of a virtual shared memory as in the runtime systems of functional languages with implicit parallelism. The low level process model of these languages obstructs the abstraction of a communication structure which could easily be mapped on a distributed system.

6 Conclusions

We have developed an intermediate language PEARL and a parallel abstract machine DREAM. The intermediate language PEARL makes clear the kernel features of Eden's coordination model. More importantly, we provide a precise abstract view of Eden's parallel runtime system by giving a complete formal STG-level specification of the DREAM extensions to the STGM. This illustrates Eden's approach to parallel functional programming from an implementation point of view.

In comparison with other parallel abstract machine models for functional languages, substantial simplifications in the parallel runtime system are possible due to Eden's explicit way of expressing parallelism and process interaction. Since one of Eden's design goals is to be efficiently implemented on distributed memory machines, there is no global program graph and thus DREAM dispenses with virtual shared memory and global memory management.

In demand driven approaches, expressions are lazily evaluated to head normal form, and many communications may take place between different processors until an expression is fully evaluated to normal form. Moreover, there is no channel concept, therefore data transmission requires at least two communications: one for request and another one for reply.

In Eden, the receiver of some output is always known because channels in Eden are unidirectional and one-to-one. Consequently, communication costs can be kept down by transferring data to the respective receiver without explicit requests. In addition, outputs are only transmitted in normal form.

In being an abstract machine, DREAM does not address practical details, e.g. the details of message passing. In DREAM, buffer capacity is assumed to be unbounded, but in the implementation special system messages will throttle the sender if the receivers' buffer capacity is approaching its limit. This means that Eden's evaluation model forms a comfortable compromise between the efficiency of speculative parallelism and the flexibility and safety of lazy evaluation.

References

- [1] J.L. Armstrong, M.C. Williams, and S.R. Virding. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
- [2] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.
- [3] Silvia Breitinger, Ulrike Klusik, and Rita Loogen. An Implementation of Eden on Top of Concurrent Haskell. In W. Kluge, editor, *Implementation of Functional Languages, Bonn 1996*, LNCS 1268. Springer, 1997.
- [4] Silvia Breitinger and Rita Loogen. Channel Structures in the Parallel Functional Language Eden. In *Glasgow Workshop on Funct. Prg.*, 1997.
- [5] Guy E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM March 1996/Vol. 39, No. 3*, pages 85–97, 1996.

- [6] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. Eden — Language Definition and Operational Semantics. Technical Report 96-10, Philipps-Universität Marburg, 1996.
- [7] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. The Eden coordination model for distributed memory systems. In *High-Level Parallel Progr. Models and Supportive Env. (HIPS)*. IEEE Press, 1997.
- [8] Manuel M.T. Chakravarty. Integrating Multithreading into the Spineless Tagless G-machine. In D. N. Turner, editor, *Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer Verlag, 1995.
- [9] J. Darlington, Y.-K. Guo, H.W. To, and J. Yang. Functional skeletons for parallel coordination. In *Euro-Par*, LNCS 966. Springer, 1995.
- [10] H. Glaser, P. Hartel, and J. Wild. A pragmatic approach to the analysis and compilation of lazy functional languages. In *Proceedings of the 2nd Conf. on Parallel and Distributed Processing*, pages 169–184. North Holland, 1990.
- [11] Kevin Hammond. Parallel Functional Programming: an Introduction. In *PASCO, Linz, Austria*. World Scientific, 1994.
- [12] Matthias Horn. A Different Integration of Multithreading into the STGM. In W. Kluge, editor, *Impl. of Funct. Lang.* Kiel University, 1996.
- [13] Paul Hudak. Para-Functional Programming in Haskell. In B. Szymanski, editor, *Parallel Functional Languages and Compilers*. ACM Press, 1991.
- [14] Paul Kelly. *Functional Programming for Loosely Coupled Multiprocessors*. Pitman, 1989.
- [15] Marco Kessler. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*. PhD thesis, Katholieke Universiteit Nijmegen, 1996.
- [16] R.S. Nikhil, Arvind, J. Hicks, Sh. Aditya, L. Augustsson, J.-W. Maessen, and Y. Zhou. pH Language Reference Manual, Version 1.0 - preliminary. Technical report, Computation Structures Group Memo 369, Laboratory for Computer Science, MIT, 1995.
- [17] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *J. of Funct. Prog.*, 2(2), 1992.
- [18] John Peterson and Kevin Hammond (eds.). Report on the programming language Haskell: a non-strict, purely functional language, version 1.4. Technical report, Yale University, 1997.
- [19] Simon Peyton Jones, Andrew Gordon, and Sigbjørn Finne. Concurrent Haskell. In *ACM Symp. on Principles of Progr. Lang. (POPL)*, 1996.
- [20] J. Reppy. CML: A higher-order concurrent language. In *ACM PLDI*, 1991.
- [21] Stephen K. Skedzielewski. Sisal. In B. Szymanski, editor, *Parallel Functional Languages and Compilers*. ACM Press, 1991.
- [22] P. W. Trinder, K. Hammond, J. S. Mattson, Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: A portable parallel implementation of Haskell. In *ACM PLDI*, 1996.
- [23] B. Thomsen, L. Leth, and T.-M. Kuo. FACILE — From Toy to Tool. In F. Nielson, editor, *ML With Concurrency*. Springer, 1997.

Using Multicasting for Optimising Data-Parallelism

Pascal R. Serrarens

Computer Science Institute
University of Nijmegen, The Netherlands
`pascalrs@cs.kun.nl`

Abstract. Multicasting, where a message is sent from one processor to number of other processors, can optimise the communication for data-parallel functional computing. This paper introduces two multicast primitives in the functional language Concurrent Clean. A new implementation of distributed arrays using these primitives is described, which shows a speedup compared to a version without multicasting. The new implementation also pointed out that the implicit communication mechanism as it is currently implemented copies too much data in some cases.

In a data-parallel programming model, many instructions are sent to all processors. An efficient way to do this is broadcasting, which sends a message simultaneously to all processors available, or multicasting, where a subset of processes receives the message. Some interconnection schemes, like buses and stars, can support broadcasting directly, because the sender has a direct connection to all other processors. Distributed architectures based on other interconnection schemes often have special hardware to support broadcasting.

Broadcasting is a real optimisation technique: it does not introduce new functionality to a system because the same effect, albeit with lower performance, can be achieved by sending the same message multiple times.

In this paper we introduce two multicast primitives which use the broadcast or multicast hardware to achieve better performance. In Sect. 1 we describe the new multicast primitive. Section 2 describes a variant of the multicast primitive, which reuses heap nodes. In Sect. 3 the use of multicasting for optimising distributed arrays [17] is discussed. Section 4 describes which difficulties were met when implementing multi-dimensional distributed arrays. Some strange behaviour for communication with distributed arrays is described in Sect. 5. In Sect. 6 the performance is compared with normal distributed arrays is compared. At the end, Sect. 7 shows related work and Sect. 8 give a direction for some future work on array-comprehension annotations, while Sect. 9 concludes.

1 Using Multicasting

Currently, all parallelism in Concurrent Clean is introduced using annotations on expressions [15]. An annotated expression will be reduced in an interleaved or parallel way depending on the annotation:

- `{| I |}`: the expression is reduced interleaved on the same processor.
- `{| P |}`: the expression may be reduced on another processor, if there is no other processor available, it behaves like a `{| I |}`-annotation.
- `{| P at p |}`, with `p :: ProcId`: the expression is reduced on the processor identified by `p`. Every processor has a unique processor identifier of type `ProcId`, which can be obtained using one of the following system functions:
 - `currentProcId` returns the processor identifier of the processor reducing it. `{| P at currentProcId |}` behaves like `{| I |}`
 - `randomProcId` returns a random processor identifier.
 - `toProcId x` transforms a integer into a processor identifier. All processors have a unique number, making it possible to pick a certain processor.

When an expression with a P-annotation is reduced, it will be sent to another processor, which will reduce the expression to root normal form. The result of the expression is locally represented by a special indirection node, called a channel. When a channel is reduced a request is sent to the remote processor, which will return the root normal form of the expression.

A good example of a data-parallel function where broadcasting can be used is the parallel map on arrays, where a function is applied to all array elements in parallel. It can be defined using the P-annotation and array comprehensions. Array comprehensions are similar to list comprehensions, they are enclosed by curly brackets `{ }`, instead of square brackets `[]`. The symbol `<-:` is used to draw an element from an array, while lists use `<-`.

```
par_map :: (a -> b) { a } -> { b }
par_map f a = { {| P |} f x \ x <-: a }
```

(Clean's notation of type differs from that of Haskell, as it uses a Cartesian product for the specification of the function type)

In this implementation (`size a`) expressions will be sent to remote processors, namely `(f a.[0])`, `(f a.[1])`, ..., `(f a.[size a - 1])` (selecting the *i*-th element of array `a` is written as `a.[i]`). Each of these expressions will be packed and sent separately, which gives a relatively large communication overhead. Alternatively we could send the function `f` with the complete array `a` to all processors simultaneously. Each process then selects the correct element from the array before it applies `f` to it.

If we want to implement this behaviour, we need a new primitive function which uses the broadcast hardware available on many distributed computers. The new primitive is not a real broadcast, but a multicast, because it sends the expression to a limited set of processors. The set of processors which should receive the message, is represented as an array of processor identifiers (type `ProcId`) and is an argument of the function.

The remote expressions will be represented by channels in the local heap. When the result of a remote expression is needed, the channel will be evaluated and the result is copied to the local processor. The number of processors available is always finite, so the channels can be well stored in an array. This array should

be a lazy array [20], which has type $\{ a \}$. If we use a strict or unboxed array (with types $\{ !a \}$ and $\{ \#a \}$ respectively) parallelism is not used to its best. First we have a synchronisation barrier: as an strict or unboxed array cannot contain closures like channels, all array elements are copied to the local processor one by one before a next expression can be reduced. But it might also be the case that the array elements are not needed at the local processor, and communication is not needed. When one wants to use a strict array, the channel should then be protected for being reduced by a constructor, like is done in [17].

The problem with broadcasts and multicasts is that every processor receives the same expression. Evaluating the same expression on all processors is not very sensible, except in safety-critical systems, where hardware errors can be detected by replicating tasks.

If we assume that every process knows the index value of the array element it should evaluate, we can have different behaviour for every task, as the value is different for every process. An element of the result array can be evaluated by applying a certain function g to its index value. In the case of the parallel map, g could be:

```
g :: Int -> a
g i = f a.[i]
```

The index value for each process, can be deduced from the array with process identifiers determining the receiving set of processors. If, for a certain processor with processor identifier p , the i -th element of the array with processor identifiers equals p , it should apply the function g to i . So when the array with processor identifiers equals $\{P_0, P_1, P_0\}$, processor 0 will create processes evaluating $(g\ 0)$ and $(g\ 2)$, and processor 1 will evaluate $(g\ 1)$.

Our new multicast function has then the following type:

```
multicast :: { #ProcId } (Int -> a) -> { a }
```

The semantics of `multicast` can be described using an array comprehension and the P-at-annotation (of course the real implementation uses the broadcast hardware).

```
multicast ps g =
  { { | P at ps.[i] | } g i \ i <- [0 .. size ps - 1] }
```

The function g is send to all processors in ps and is applied to the appropriate index value.

Now we can implement a more efficient version of the parallel map. In this example the functions are distributed randomly over the processors.

```
par_map :: (a -> b) { a } -> { b }
par_map f a = multicast ps g
where
  ps = { randomProcId \ i <- [0 .. size a - 1] }
  g i = f a.[i]
```


Unfortunately, this function is hardly recognisable as being a map, because the `multicast` primitive gives it a different structure. For the time being, we will accept this, but we are already working on a better solution (see Sect. 8).

Another example where multicasting can be used is a parallel version of the `make_array` function. It is given a generator function and the length of the new array, evaluates the elements in parallel and returns an array with channels to the elements:

```
par_makeArray :: (Int -> a) Int -> { a }
par_makeArray f n = multicast ps f
where
  ps = { randomProcId \\ i <- [1 .. n] }
```

2 Updating Multicast

The `multicast` primitive from the previous section has one drawback: it always creates a new array for the result. This is not always necessary: you could have an array which can be recycled, using update-in-place. Therefore we give a alternative multicasting primitive, which does exactly that:

```
multicast_u :: { #ProcId } (a Int -> a) *{ a } -> *{ a }
```

It takes an extra argument: a lazy array with the uniqueness annotation `*`. This array can be updated destructively, because the uniqueness typing system ensures that the array will not be shared [1]. The function passed to `multicast_u` now also takes the element at each position, because when a selection is used, as in `par_map` above, the array will loose its unicity, which should not happen. We still pass the index of the element to the function, because some functions, like rotations, may need it.

Like the `multicast` primitive, we can define the semantics (again ignoring broadcast hardware) with a normal Clean function: (when two generators are separated by `&`, the elements are drawn simultaneously. In an array comprehension the construction `{ a & [i] = ... }` means that it will update array `a` at position `i`)

```
multicast_u ps g a =
  { a & [i] = { | P at ps.[i] | } g x i \\ x <-: a & i <- [0 .. ] }
```

An example of an updating parallel map, which reuses the array argument:

```
par_map_u :: (a -> a) *{ a } -> *{ a }
par_map_u f a = multicast_u ps g a'
where
  (n, a') = usize a
  ps      = { randomProcId \\ i <- [0 .. n - 1] }
  g x _   = f x
```

It could be possible to keep `multicast_u` as the only new primitive function, because the function `multicast` can be defined as:

```
multicast ps f = multicast_u ps g (createArray (size ps) (f 0))
where
  g x i = f i
```

However, this implementation of `multicast` can be much more inefficient than the primitive. The problem is that, in the code above, the array needs to be initialised with the result of the expression `(f 0)`, which may be expensive to reduce. The primitive `multicast` does not need the initialisation, but fills the array with the channels to the elements directly. So for efficiency reasons we introduce both `multicast` primitives.

3 Distributed Arrays

The arrays returned by the functions above are not very convenient. First, the locations of the elements are not clear. Second, the lazy evaluation scheme of Clean makes it hard to reason when a channel is needed. This results in communication where it is not expected or needed.

In [17] we introduced Remote Values, which are implemented with an ordinary datatype, which is abstract:

```
:: R a = Remote ProcId a
```

They combine data `a` with its location `ProcId`. The constructor prevents unwanted communication [9], while all functions on Remote Values use the stored location explicitly:

```
putRemote :: !ProcId a -> R a
putRemote p x = Remote p y
where
  y = { | P at p | } x

rap :: (a -> b) (R a) -> R b
rap f (Remote p x) = Remote p y
where
  y = { | P at p | } f x

getRemote :: R a -> a
getRemote (Remote p x) = x
```

It was shown that an efficient implementation of distributed arrays can be made in Clean without any extension, using an array of Remote Values. Multicasting could improve those results, because in many cases, the same function is sent to all processors.

To use multicasting to its best, we introduce an alternative implementation for distributed arrays. Instead of using an array of Remote Values, we define a new datatype:

```
:: DArray a = DA { #ProcId } { a }
```

Again, we use a ordinary constructor to prevent unnecessary communication. The first array argument of the datatype describes the locations of the elements of the second.

A distributed array can then be created using a function like `make_Dist`:

```
make_Dist :: { #ProcId } (Int -> a) -> DArray a
make_Dist ps g = DA ps (multicast ps g)
```

Functions like `map` and `zipwith` are straightforward, because they send the same function to the processors:

```
map_Dist :: (a -> b) (DArray a) -> DArray b
map_Dist f (DA ps a) = DA ps (multicast ps g)
where
  g i =          f a.[i]
```

Another, less obvious, example is a tree-shaped fold on arrays. It uses a multicast for every level in the tree.

```
fold_Dist :: (a a -> a) (DArray a) -> R a
fold_Dist f (DA ps a)
  | size a == 1          = rselect_Dist (DA ps a) 0
  | (size a) mod 2 == 0 = fold_Dist f (DA ps' (multicast ps' g))
  | otherwise           = fold_Dist f (DA ps' (multicast ps' h))
where
  ps' = { ps.[i] \\ i <- [0, 2 .. size ps - 1] }

  g i = f a.[i * 2] a.[i * 2 + 1]

  h 0 = a.[0]
  h i = f a.[i * 2 - 1] a.[i * 2]
```

For some other operations, we should have access to the individual elements of the distributed array. Therefore, we introduce the function `rselect_Dist`, which selects an element from an array and returns it as a Remote Value. Updating one element of the array can be done with `ap_Dist`, which does not use Remote Values, but updates the distributed array in-place with a channel to the new element being evaluated by a new process. (The construction `a={[i]=x}` is like a pattern match on an array: the *i*-th element of array *a* is selected and assigned to *x*. When the array *a* is not needed, the first part can be left out, as in `rselect_Dist`.)

```

rselect_Dist :: (DArray a) Int -> R a
rselect_Dist ( DA {[i]=p} {[i]=x} ) i = Remote x p

ap_Dist :: (a -> a) *(DArray a) Int -> *DArray a
ap_Dist f ( DA ps a={ [i]=x } ) i = DA ps (update a i x')
where
  x' = { | P at ps.[i] | } f x

```

The `rselect_Dist` function is convenient when operations should be applied on individual elements, for example in the `foldlr_Dist` from [17].

```

foldlr_Dist :: (a a -> a) (DArray a) -> R a
foldlr_Dist f dx = foldlr 1 (rselect_Dist dx 0)
where
  foldlr i r
    | i >= size_Dist dx = r
    | otherwise
      = foldlr (i + 1) (rap2Snd f r (rselect_Dist dx i))

```

4 Distributed Arrays with Higher Dimensions

Using the multicast primitive for distributed arrays with more than 1 dimension gives a problem. The easiest way is to use the 1-dimensional distributed arrays and make every row a distributed array:

```

::      DArray2 a ::= { DArray a }

```

The problem is that we need $O(n)$ time to send the messages for a map over a $n \times n$ distributed array, while it should be possible to do it in constant time, like with the 1-dimensional distributed arrays in the previous section. An alternative is to use nested distributed arrays:

```

::      DArray2 a ::= DArray (DArray a)

```

For a map-like function, the local processor first broadcasts the function to the processors in the first column, which then in their turn broadcast the function over the columns in parallel. While this in theory takes only 2 steps, it may take $O(n)$ time in reality. For example in a bus-based network, two or more messages cannot be sent at the same time, and therefore the “parallel” distribution of the function over the columns will happen sequentially. Of course we gain a little, because the packaging of the message can happen in parallel, but as we have an extra stage in broadcasting the function, we do not expect this to be effective.

The trouble with more dimensional distributed arrays is that the results and locations are stored in n -dimensional arrays, while `multicast` uses 1-dimensional arrays. We do not want to flatten and expand the structure with every function using `multicast`, because that introduces too much overhead.

A solution is to store the n -dimensional arrays in a flat, monolithic structure, like in APL [8] and SAC [16]. The data and locations are stored in 1-dimensional arrays and a so-called shape describes the structure of the data:

```
::      DArray_n a = DA_n { #ProcId } Shape { a }
```

```
:: Shape ::= { #Int }
```

An array with shape {3, 4} is a 2-dimensional array with 3 rows and 4 columns. With this datatype we can implement the map-like functions easily, using only one multicast:

```
map_DArray_n :: (a -> b) (DArray_n a) -> DArray_n b
map_DArray_n f (DA_n ps sh a) =
  DA_n ps sh (multicast ps g)
where
  g i = f a.[i]
```

The drawback is that it is a totally different approach to arrays, with a different behaviour than the standard arrays in Clean. Moreover, functions that do need the structure of the matrix are much harder to implement. However, it has the advantage that dimension-independent functions can be written and n -dimensional arrays are stored efficiently in one continuous memory area.

5 Normal Form Copying

We have two kinds of nodes in the heap: closures, which represent work, and data, which is in normal form. Lazy normal form copying copies all data eagerly, except when it can only be reached via a closure, while it stops at closures.

In data-parallelism the data on which functions are applied in parallel are often stored locally on each processor. Therefore one has to be sure that the lazy normal form copier cannot reach that data directly, because it will not be used at other processors most of the time. When we implemented the multicast-based distributed arrays, we noticed that much more data was copied than expected in some cases, especially for larger problems. So we suspected that the copier could reach data which should not have been reachable.

(We assume that `p0` is the processor identifier for the master processor, so the first element of the distributed array is placed on the master processor)

```
ps = { p0, p1, p2 }
```

```
g :: Int -> { #Real }
h :: { #Real } Int -> { #Real }
```

```
f :: { #ProcId } -> { #Real }
f ps = multicast ps (h a)
where
  a = multicast ps g
```

The program above is a simplified program construct which appears often in data-parallel programs, for example with two consecutive maps. In the example, function *g* is multicasted first, because *h* depends on *a*. After that first multicast, *a* can be represented by the graph in Fig. 1a. Notice that the zero-th element of the array is not a channel, but a normal link. Channels pointing to graphs at the same processor are always replaced by normal links. Now every processor can reduce its element to root normal form and we get Fig. 1b. The second multicast takes place next, copying *a* to all processors. But now we see that the subarray *a*. [0] is in normal form and reachable from the root of *a* and therefore it will be copied by the lazy normal form copier. In most cases this subarray is not needed on any other processor than *p*₀, as the most common functions are map-like and only use the local subarrays. In the case of distributed array, the subarrays are usually rather large and as the number of multicasts is quite high, the communication overhead introduced by this problem was rather high.

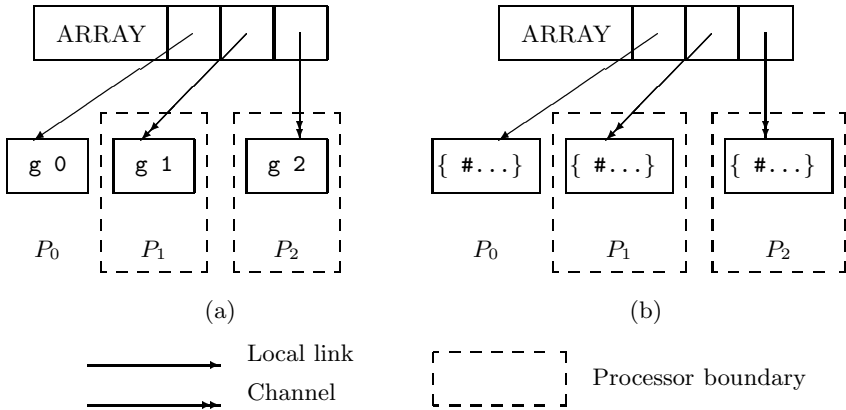


Fig. 1. A global graph after multicasting, before (a) and after (b) the local elements of the array have been reduced

One solution to this problem is to use $n + 1$ processors: n for the subarrays and 1 master processor, which sends the functions to all others. This has the big disadvantage that it needs one extra processor, which is not always available. In the case of a n^m network it gives us the inconvenient number of $n^m - 1$ slave processors.

Keeping channels to local graphs, instead of replacing them by normal links, is also a solution to the problem above, but it solves the problem only in this particular case: the problem also arises in other, harder, cases.

Another option was to defer [15] unboxed arrays by default, so that they will not be copied directly, but a channel pointing to them is created at copying time. When they are really needed the channel is evaluated and the array is copied.

This may introduce unnecessary communication in some other cases. When an array is packed into a datatype, like

```
:: D = C { #Real }
```

copying it to another processor takes two copies instead of one. First one to copy the constructor **C** and while a second message carries the array.

As it was rather clear in which cases the problem arises, we decided not to take a solution like limiting the number of bytes in a packet [13], but to adapt the copying mechanism. When we copy an array with some channel elements, we replace all non-channel elements, including normal forms, by channels in the copy. In this case, a subarray will not be copied directly, even when it is in normal form.

The problem behind this is that the lazy normal form copying strategy always copies all data up to closures eagerly, while sometimes a large part of that data will never be needed. It seems to be that we might want to reuse the defer attribute [15]. A defer attribute on a node means that the node will not be copied, but a channel pointing to it will be created instead. In our case we might want to use a defer annotation to denote that the subarrays should not be copied directly. The main difference between our intention and the semantics given in [15] is that we want to keep the defer attribute when the node has been reduced to normal form.

6 Performance Measurements

The **multicast** primitive has been included in a distributed implementation for a network of Macintosh computers. It involved a major redesign of the communication mechanism, because the Appletalk ADSP point-to-point protocol we were using did not support broadcasting. The Appletalk DDP protocol supporting broadcast is more low-level and gave us the opportunity to cut down the communication costs. The numbers presented in this paper are therefore not directly comparable with those presented in [17].

We tested a parallel implementation of the conjugate gradient algorithm [12] on a network of Apple Macintosh computers. The program was not optimised for absolute speed (see also [18]), as we were concentrating on the parallel speedups. The conjugate gradient algorithm approximates the solution of the equation $Ax = b$, where matrix A and vector b are given and vector x should be approximated. It uses a large number of map-like functions, so using multicasting should be appropriate here. Every time in the table is the average of 5 test-runs on 4 Apple Macintosh II computers connected by Ethernet.

Table 1 shows the various runtimes for different network sizes. We notice that using multicasts on a 2-processor network does not decrease the performance very much. The overhead introduced by the multicast primitive seems to be reasonable. In all cases the multicast version was less than 5% slower than the version with point-to-point communication. For the 4-processor network, we see a

Table 1. The execution times in seconds for the conjugate gradient algorithm. (mc) is with multicasting, (p2p) with normal point-to-point communication. Also included are the runtimes for a pure sequential implementation of the algorithm

matrix size	sequential	2 processors		4 processors	
		mc	p2p	mc	p2p
400^2	22.06	18.85	18.20	21.65	24.93
900^2	75.86	50.20	48.43	40.70	46.60
1600^2	175.90	105.23	102.31	71.08	78.53
2500^2	337.25	192.85	189.03	115.81	121.26
3600^2	590.85	323.00	318.33	184.90	188.05

speedup of 5-15%, so using multicasts increases the efficiency on larger networks. We expect that the performance gain on larger networks will be greater.

From Fig. 2, we notice that for bigger problem sizes the performance difference between the versions with and without multicasting decreases. The reason is that the multicast version creates many more channels. With many multicasts the distributed array is sent to all processors, while every element of that array is a channel (see also the previous section). The consequence of this is that the large number of channels introduce more overhead for the channel management.

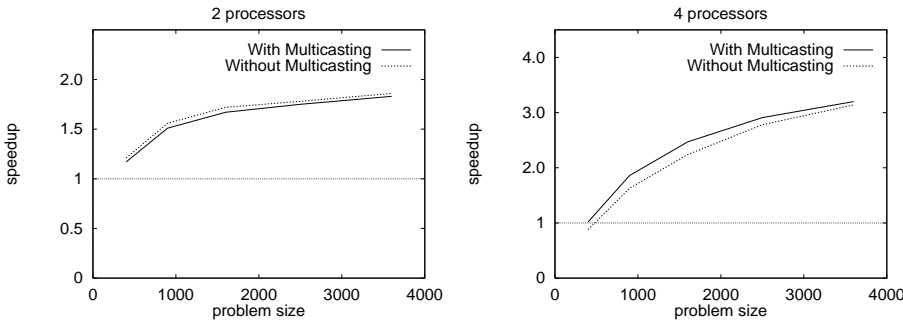


Fig. 2. Comparison between the multicasting and point-to-point version of the conjugate gradient algorithm for 2 (left) and 4 (right) processors

Apart from the fact that many channels are created, is that they become garbage very quickly. This introduces a lot of inter-processor garbage, which takes longer to collect than local garbage. It takes at least two garbage collections: first one for each processor containing a channel to see that the channel has become garbage. Then a garbage collection on the processor maintaining the graph to which the channel pointed, to detect that the graph has become garbage. This has the consequence that the amount of “live” data (all data in

the heap surviving a garbage collection) is larger, so the garbage collector will be called more often.

7 Related Work

In most programming languages multicasting is not a programming construction, but a language implementation issue. Languages like High Performance Fortran (HPF) [6] define constructs like the `FORALL`, which is a parallel version of a `DO`-loop with a “all-at once” semantics. Language implementations then use PVM’s `pvm_mcast` [4], MPI’s `mpi_bcast` [14] or machine-dependent communication primitives to implement these operations [5].

There has been a lot of work on parallel functional programming, but broadcasting or multicasting has not received much attention. This is partly because many ways of introducing parallelism in functional programs are not suited for it. Annotations [15] are put on function applications which will be sent to one processor only. The `par` primitive of GpH, an extension of Haskell [7], evaluates one expression at only one processor.

Apart from these rather low-level parallel primitives, many other higher-level (data-)parallel constructs are proposed. However, as with imperative languages, multicasting is part of the language implementation instead of the language specification.

Skeletons [2] could make use of multicasting. Take for example the DC skeleton, which represents the diver-and-conquer approach:

```
DC :: (a -> Bool) (a -> b) (a -> {a}) ({b} -> b) a -> b
DC triv solve divide combine x
  | triv x      = solve x
  | otherwise =
    let!
      xs = divide x
      ps = { randomProcId \\ x <-: xs }
    in
      combine (multicast ps (dc_ xs))
where
  dc_ {a} Int -> b
  dc_ a i = DC triv solve divide combine a.[i]
```

The idea behind skeletons is that they are provided as primitive higher-order functions, while the implementation is optimised for every machine platform. The function above can serve well as a basic implementation of the skeleton, which can be used to make specialised implementations. Optimising it for a certain machine platform consists mainly of finding a good set of processors for each multicast, instead of a random distribution as depicted here.

Multicasting can also be used for strategies [19]. For example, we can make a strategy for arrays using multicast, where the strategy is applied to all array elements simultaneously:

```

parList :: Strategy a -> Strategy [a]
parList strat a = multicast ps g
where
  ps = { randomProcId \\ i <- [0 .. size a - 1] }

  g i = strat a.[i]

```

In [11] Distributed Applicative Arrays (DAA's) are proposed. The operations on them are defined in terms of `gen`, `sel` and `fold`. These three functions are similar to our `make_Dist`, `rselect_Dist` with a `getRemote`, and `foldlr_Dist` respectively. However, in their paper Kuchen and Geiler suggest a more tree-like distribution technique of the DAA's, which is an dynamic adaptive method to ensure that the array is distributed efficiently. In our construction, this distribution has to be specified statically by the programmer, which can be more efficient.

In SCL [3] broadcasts are used to implement functions like `map` on parallel arrays (which are enclosed by `<<...>>`). A broadcasting function is also provided, but it is a real data broadcast. However it is possible to implement a function similar to our `multicast`. It takes an argument for the length of the parallel array instead of an array with processor identifiers. Furthermore, it uses a parallel `map` (which is implemented using a broadcast) over an parallel array with indexes:

```

map :: (a -> b) -> ParArray index a -> ParArray index b
map f << x0, ..., xn >> = << f x0, ..., f xn >>

multicast :: Int (Int -> a) -> ParArray Int a
multicast n f = map f is
where
  is = << ii := ii | ii <- [1..n] >>

```

8 Future Work

Multicasting can also be used to improve the performance when answering requests from channels. When a channel on one processor is copied to a group of processors, it is likely that the processor holding the data will receive multiple requests. When that data has been reduced to root normal form, it can be sent to all processors waiting for the data simultaneously using a multicast.

In Clean, the `I`- and `P`- annotations were the only way to introduce parallelism. They cannot change the outcome of a program, but only change the place where an expression is evaluated. Therefore, if they are left out, a normal sequential program remains. The primitives introduced in this paper cannot be left out easily because they are ordinary functions. An alternative for those primitives is a new annotation, which is only allowed on array comprehensions. The elements of the result of the array comprehension will be evaluated on the indicated processors. A parallel map, using the annotation, could look like:

```

y = { | PS at ps | } { f x \\ x <-: xs }

```

If the annotation is left out of the function above, we have a normal map over arrays. The new annotation can be used in all situations where a multicast can be used, including the implementation for distributed arrays, but gives clearer programs. It's also easier to use, as parallelising array comprehensions efficiently is now a matter of just putting an annotation in front of it.

An annotated array comprehension will be transformed using a set of rules which use the `multicast` and `multicast_u` primitives introduced in this paper to distribute the expressions over the processors. The two primitive functions should then be hidden: it is possible to achieve the same behaviour with the annotated array comprehensions:

```
multicast ps f =
  { | PS at ps | } { f i \ i <- [0 .. size ps - 1] }

multicast_u ps f a =
  { | PS at ps | } { a & [i] = f i \ i <- [0 .. size ps - 1] }
```

Creating such an annotation on list comprehensions is a harder, because it is hard to tell what the length of the result list will be. Moreover it is also possible to create infinite lists with list comprehensions. These two reasons complicate the use of multicasting, as the set of processors to which a function should be sent is hard to determine.

9 Conclusions

With the introduction of multicasting in Clean, a more efficient implementation of important parallel constructions can be made. We showed that the proposed primitives are particularly suited for data-parallel programming. However, multi-dimensional distributed arrays give some troubles if we want a single message for every map-like function. The best option here seems to be dimension-independent arrays.

We also noticed unwanted behaviour with respect to normal form copying. Copying a graph as far it is in normal form is not always desirable, especially when we want to use data-parallelism. When a large part of the data in normal form is not needed on the destination processor, copying that part gives unnecessary communication overhead. At the moment there is no possibility to influence this behaviour directly.

Using multicasting for data-parallel programming can be favourable. In a 2-processor network we had only a slight speeddown ($< 5\%$), while for a 4-processor network we had a speedup of up to 15%, compared to a version without multicasting. The garbage collection costs were higher, because the multicast version creates more channels, which take longer to collect.

References

- [1] E. Barendsen and J.E.W. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In R.K. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*, pages 41–51. Springer-Verlag, 1993. extended abstract.
- [2] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.
- [3] J. Darlington, Y. Guo, H.W. To, and J. Yang. Skeletons for structured parallel composition. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.
- [4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Menchek, and V. Sunderam. *PVM: parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. MIT press, 1994.
- [5] M. Gupta et al. An hpf compiler for the ibm sp2. *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, 1995.
- [6] High Performance Fortran Forum. *High Performance Fortran Language Specification, version 1.0*. Houston Texas, May 1993.
- [7] P. Hudak, S.L. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, Mar, 1992.
- [8] K.E. Iverson. *A Programming Language*. Wiley, New York, 1962.
- [9] M.H.G. Kessler. Constructing skeletons in Clean - the bare bones. In *Proceedings of High Performance Functional Computing*, pages 182–192, Denver, Colorado, 1995. CONF-9504126.
- [10] Werner Kluge, editor. *Implementation of Functional Languages, 8th International Workshop, Selected Papers*, volume 1268 of *LNCS*, 1997.
- [11] H. Kuchen and G. Geiler. Distributed applicative arrays. Technical Report AIB 91-5, RWTH Aachen, 1991.
- [12] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing, Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., California, 1994.
- [13] H.-W. Loidl and K. Hammond. Making a packet: cost-effective communication for a parallel graph reducer. In Kluge [10], pages 184–199.
- [14] Message Passing Interface Forum. *MPI: a Message Passing Interface Standard*, May 1994.
- [15] M.J. Plasmeijer and M.C.J.D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Publishers Ltd., 1993.
- [16] S.-B. Scholz. On programming scientific applications in SAC - a functional language extended by a subsystem for high-level array operations. In Kluge [10], pages 85–104.
- [17] P.R. Serrarens. Distributed arrays in the functional language Concurrent Clean. In *Proceedings of the 3rd international Euro-Par conference*, volume 1300 of *LNCS*, pages 1201–1208, Passau, Germany, August 1997.
- [18] P.R. Serrarens. Implementing the conjugate gradient algorithm in a functional language. In Kluge [10], pages 125–140.
- [19] P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8(1), Jan 1998. to appear.
- [20] J.H.G. van Groningen. The implementation and efficiency of arrays in Clean 1.1. In Kluge [10], pages 105–124.

Using Concurrent Haskell to Develop Views over an Active Repository

Einar W. Karlsen and Stefan Westmeier

Bremen Institute for Safe Systems (BISS)
FB3 Mathematik und Informatik
Universität Bremen, Germany
`ewk@informatik.uni-bremen.de`
`stefan.westmeier@uni-bremen.de`

Abstract. The UniForM WorkBench is an integration framework with Haskell features in support of data, control and presentation integration. Using the WorkBench, it is possible to implement the entire Software Development Environment for Haskell - using Haskell itself. The paper presents the higher order approach to event handling used within the WorkBench, as well as the persistent repository with version management support. It is then demonstrated how views over this repository are kept consistent, on the fly, in a multi-user environment using the Model-View-Controller paradigm. Interactors are set up to maintain consistency between a view and its underlying repository by coordinating database change notifications and user interactions. These events are represented as first class, composable event values.

1 Introduction

There is a strong tradition within the functional programming community to bootstrap the compiler for a functional language. Not only the compiler, but even the entire *Software Development Environment* (SDE), with integrated tool support, graphical user interfaces, and version and configuration management, could be implemented very elegantly using the functional language itself, under the provision of course, that the right kind of integration technology were available.

The *UniForM WorkBench*¹ [17] is such an integration framework offering services for *data integration* (DBMS), *presentation integration* (GUI), and *control integration* in Concurrent Haskell [23]. The WorkBench is set up along the imperative functional programming paradigm [24], extended with a higher order approach to event handling inspired by CML [25]. The main achievement over CML, however, is that internal channel events as well as tool events, such as user interactions and database change notifications, are uniformly represented in terms of composable and abstract event values that entirely hide the source of the event [13].

¹ This work has been supported by the German Ministry of Research (BMBF) on Project UniForM ("Universal Formal Methods WorkBench")

We have used this integration framework to develop a prototypical environment for Haskell program development - the *Hugs WorkBench* - that integrates Haskell tools such as the Hugs interpreter [11] around a persistent repository providing version and configuration management of Haskell modules. The first step in such an integration is to wrap Haskell interfaces around the pre-fabricated development tools (e.g. Hugs). Haskell is from then on used to glue the entire system together. In essence, the Hugs WorkBench turns out to be a reactive system, where events amount to user interactions, tool events, operating system events and database change notifications.

This paper is concerned with a central issue of such an integrated environment: the provision of consistent views over the repository. Other issues, related to control integration and tool encapsulation, are for example discussed in [12]. The intricate aspect of an SDE is that it is inherently multiuser. While one user is viewing a version graph, another may change it by creating or pruning versions. Views over the repository must therefore be maintained dynamically - on the fly. It shall be demonstrated how this can be done very elegantly by using a functional language extended with a higher order approach to event handling.

The remainder of this paper is organized as follows. The architecture of the WorkBench is introduced in the next section. First class synchronous events and tool events are discussed in section three and four - respectively. The fifth section presents the main features of the Repository Manager, in particular its support for attributed and versioned objects. The sixth section demonstrates the application of this technology on a larger example by defining a version graph over the repository. The results are then discussed.

2 Architecture

The UniForM WorkBench has been established on top of existing, public domain tools [15]. Data integration is provided by the *Repository Manager*, which takes its basis in a Haskell encapsulation of H-PCTE [3], the OMS Toolkit [29]. H-PCTE is a persistent, distributed and active object management system that implements an extended subset of the Portable Common Tool Environment [4].

Presentation integration is provided by the *User Interaction Manager* (UIM) [14], which consists of Haskell encapsulations of Tk [22] and the graph visualization system daVinci [7]. *Haskell-Tk* serves as a general purpose GUI for wrapping graphical user interfaces around non-graphic tools. The graph visualization system *daVinci* is, in turn, used to visualize graph structures of the Repository Manager such as version and configuration graphs.

Control integration is provided by a number of concurrently executing *interactors* that are set up to listen to the events of the integrated system. The response to an event is then defined in terms of a computation that usually calls one or more commands of the tools making up the environment.

The Hugs WorkBench is illustrated in Fig 1. A version graph for the Queue module is visualized using daVinci. New revisions are made using a text editor. Attributes of a revision, such as the informal description, are edited using an

By using the MVC paradigm throughout to maintain views over the repository, it does not matter, from the point of view of the implementation, whether the changes have been made (1) within the current view, (2) within another view running in the same WorkBench or (3) by a completely different user at a remote machine. Distribution is achieved, almost for free!

3 Synchronous Events

The *UniForm Concurrency ToolKit* [13] extends Concurrent Haskell [23] with a message passing model similar to the one of CML [25], where concurrently executing agents communicate over *typed channels*. Communication is expressed by letting agents synchronize on first class, composable *event values*.

A concurrent system is expressed using two kind of domains. Values of type `IO a` represent reactive computations that are executed for their effect, whereas values of type `EV a` represent events that will return a value of type `a`, or fail with an error, whenever the event occurs. The following computations, base events and event combinators are provided (Fig. 2):

- `channel` creates a new channel of type `Channel a`.
- `receive ch` denotes the event for reading a value over the channel `ch`.
- `send ch v` denotes the event for sending value `v` over channel `ch`.
- `inaction` denotes the empty set of events corresponding to the *null process* of process algebras.
- `e1 +> e2` denotes the *guarded choice* operator.
- `e >>>= c` is the *event-action* combinator that combines an event `e` with some reactive behaviour given in terms of a continuation function `c`.
- `sync e` is the operation that synchronises on the event `e`. Execution of `sync e` will suspend until one of the communications denoted by `e` occurs.

Communication is by handshake between two threads, which means that a sender and a receiver must perform a rendezvous in order to communicate. Several senders and receivers may attempt to communicate simultaneously over one and the same channel, but communication is ensured to be on a point-to-point basis: each message communicated involves exactly one sender and one receiver. Channels are, as in CML, parameterized over the type of message communicated over the channel. The message type can be an ordinary value, a channel value or even a computation of type `IO` which caters for higher order features.

The expressive power of the framework lies in the nature of the guarded choice and the event-action combinator, which are used to construct new event values from the basis of existing ones. The event-action combinator is to events what sequential composition is to computations, since it glues some additional reactive behaviour onto an existing event. Just as for computations, a combinator is provided that discards the value returned when an event occurs:

```
(>>>) :: EV a -> IO b -> EV b
e >>> c = e >>>= \_ -> c
```



```

data Channel a

channel      :: IO (Channel a)
receive     :: Channel a -> EV a
send        :: Channel a -> a -> EV ()

sync        :: EV a -> IO a

(>>>=)      :: EV a -> (a -> IO b) -> EV b
(>>>)       :: EV a -> IO b -> EV b
inaction    :: EV a
(>+)       :: EV a -> EV a -> EV a

tryEV       :: EV a -> EV (Either IOError a)

```

Fig. 2. Synchronous Events

The type `EV` is actually a functor. Additional reactive behaviour given in terms of a function, rather than a computation, is therefore provided by `map`:

```
instance Functor EV where map f e = e >>>= return . f
```

The guarded choice operator provides a concept of *generalised selective communication*. The guarded choice `e1 +> e2` denotes a choice between two (possibly composite) events, corresponding to the summation operator `+` of process algebras. The synchronisation over a sum of events will then choose among one of the `send` or `receive` events for which communication is possible.

A couple of derived events can be defined using guarded choice. The `choose e1` combinator turns a list of events `e1` into a single event value. The `select e1` command is a further refinement that suspends execution until one of the events denoted by the list `e1` occurs:

```

choose :: [EV a] -> EV a
choose = foldr (+>) inaction

select :: [EV a] -> IO a
select = sync . choose

```

The `inaction` event takes the role of the empty process in process algebras. Essentially, this means that deadlock can be modelled by `sync inaction`. Notice also, how `inaction` is used as the neutral element in the definition of `choose`: `e +> inaction` is the same as `e`.

The model is very similar to the one of CML with one important deviation. Unlike CML, we provide infix operators for composing events whose representation is entirely hidden to the user. Actually, what's behind a value of type `EV a` is a sequence of base events. Each base event is in turn represented in terms

of computations needed to implement generalised selective communication combined with a computation that specifies the reactive behaviour associated with the event [21].

3.1 Interactors

A limitation of `sync` is that it synchronises on an event at most once. Frequently, when developing reactive systems, an agent must be set up to react to an event repeatedly throughout its entire lifetime. The model for selective communication has therefore been extended with a concept of *interactors* providing *iterative choice* over an event.

The following computations are provided for interactors (see Fig 3):

- the computation `interactor e` creates a new interactor that repeatedly interacts as defined by `e`.
- the computation `become e` changes the behaviour of the current interactor so that the interactor will from now on interact as defined by `e`.
- the computation `stop` terminates the current interactor.

The `stop` command is actually a short hand feature: it is defined as a call to `become inaction`, which changes the behaviour of the interactor to that of the empty process.

```
data InterActor

interactor :: EV () -> IO InterActor
become     :: EV () -> IO a
stop       :: IO a
```

Fig. 3. Interactors

Interactors provide a refinement of the *Actor model* of Gul Agha [1]. However, rather than defining the response to an event in terms of a continuation function and an event dispatching case statement, it is defined in terms of an event value that hides the actual dispatching being done.

Channels and threads are garbage collected whenever they are not needed by the system anymore. However, when developing reactive programs, such as SDE's, we are interfacing to external components of the Haskell program such as windows and operating system processes. An inevitable property of these entities is that their lifetime is limited.

The WorkBench uses Haskell classes to structure the code, and to achieve a number of lean and standardised interfaces to its underlying services. One such class is the class `Destructible`, which offers a `destroy` computation for

```

class Destructible o where
  destroy    :: o -> IO ()
  destroyed  :: o -> EV ()

```

Fig. 4. Destructible Objects

destroying objects together with a **destroyed** event for listening to such events (see Fig. 4).

One advantage of using first class event values and interactors is that we can define new interaction idioms, such as for example a controller for a destructible object. This controller extends a basic interactor with a destruction event, as to ensure that the interactor stops executing whenever the destructible object, to which it is attached, is destroyed:

```

controller :: Destructible o
           => o -> EV a -> IO InterActor
controller o e = interactor (
  tryEV e      >>>= either openErrWin (const done)
  +> destroyed o >>> stop
)

```

Interactors are not robust with respect to errors, meaning that, if a reaction fails with an error, then the interactor will stop, unless of course, an error handler has been wrapped around the event. This is exactly, what the **tryEV** event does. This operation is to events, what **try** is to computations, and gives the application a chance to react to errors. In the case of the controller, an error dialogue is opened that displays the error message, and asks the user whether the interactor should go on serving events or whether it should be terminated.

4 External Tool Events

In a framework to event handling based on synchronous events, we would like to represent tool events, such as user interactions and repository change notifications, as first class composable event values. All a listening thread should do in order to receive an event **e** from an external source would be to call **sync e**. Such tool events could then be combined to form new composite events, and one could freely mix internal and external events using guarded choice and the event-action combinator, thus having a uniform and composable framework to event handling that is independent of the actual source of the event.

The WorkBench is based on a concept of event *sources*, *adaptors* and *interactors* to achieve this degree of abstraction (Fig. 5). The main component is the adaptor which is interposed between the physical event source (i.e. a GUI or a DBMS) and the interactors of the application. It is the role of the adaptor to turn a tool event into a first class composable value of type **EV**, and to delegate

such an event, whenever it occurs, to the interactors awaiting the event. It is, on the other hand, the role of the interactors to carry out the reaction to such an event, i.e. to provide the logic of the application.

The architecture is illustrated in Fig. 5, where a counter widget is used to view and update some integer attribute of a persistent object. The counter widget consists of an entry widget displaying the current count and 2 button widgets for incrementing and decrementing the count. The count attribute of the persistent object can be changed, either when the user interacts with the counter widget, or when some other application sets the count attribute of the persistent object calling operations of the Repository Manager. User interactions and repository change notifications are represented as values of type `EV`, and a single controller has been set up to coordinate changes to the attribute value and changes to the counter widget.

Tk and H-PCTE are foreign tools, implemented in Tcl and C. A Haskell API has then been wrapped around the tools, which defines the types, computations and events of relevance. It is the role of the Tk adaptor and the H-PCTE adaptor to turn physical events of the two foreign tools, whatever representation they may have, into first class composable events.

4.1 Listen Event

Tool events (Fig. 6) are communicated from the event adaptor to the set of listening interactors in terms of tuples $(\mathbf{eid}, \mathbf{d})$, where \mathbf{eid} denotes the unique *event designator* of the event and \mathbf{d} the *event descriptor*.

An interactor is set up to receive a tool event \mathbf{eid} by synchronizing on the `listen eid r dr` event, where `r` and `dr` are computations that register, respectively de-register, the interactor with the adaptor mediating the event denoted by \mathbf{eid} . All the *registration command* does, is actually to inform the adaptor that "when this event occurs, please forward it to me". The de-registration command has the opposite effect. Both commands are executed behind the scene when `sync` is called.

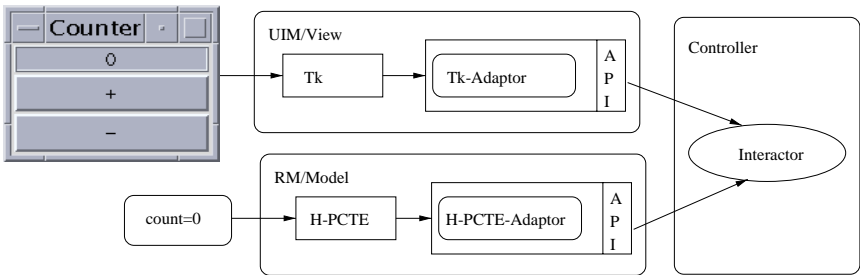


Fig. 5. Model-View-Controller Paradigm in Action

```

class EventDesignator e where
  toEventID  :: e -> EventID

class Typeable t where
  toDyn      :: t -> Dyn
  fromDyn    :: Dyn -> Maybe t

type Register = InterActor -> IO ()

listen       :: (EventDesignator e, Typeable a)
              => e -> Register -> Deregister -> EV a

```

Fig. 6. External Tool Events

All tool events are required to be uniquely identifiable in terms of event designators of type `EventID`. The class `EventDesignator` abstracts over the kind of types that can be used as event designators.

The event descriptor carries relevant information about the event in terms of a value of the *dynamic* type (`Dyn`). The *injection* (`toDyn`) and *projection* (`fromDyn`) functions are provided through class `Typeable` [11], and are called by the adaptor and the event listener - respectively.

The registration command is used during synchronization to inform the adaptor about the presence of a new listener. The adaptor can then delegate an incoming event from the event source onwards to the interactor. Having received an event (`eid,d`), the event listener determines the reaction associated with the event designator `eid`, passes it the received event descriptor `d` as actual argument, and executes the resulting computation. The reaction to such an event is, of course, defined by using the event-action combinator.

Adaptors can be developed on need. The `Tk` adaptor provides for example two communication paradigms: button clicks are *synchronously delegated* onwards to exactly one of the interactors having registered interest in the event, whereas change notifications, such as the destruction events, are *asynchronously broadcasted* to all interested interactors.

4.2 User Interactions

When a Haskell interface is wrapped around a reactive tool such as a GUI, specializations of the `listen` event are usually defined that hide the registration commands of the adaptor. *Haskell-Tk* defines for example all synchronous user interactions in terms of the following specialization:

```

interaction :: (GUIObject o, GUIEvent e) =>
  o -> e -> EV GUIEventInfo
interaction w e = listen eid (regGUI eid) (deregGUI eid)
  where eid = toEventID (w,e)
clicked bt    = interaction bt Clicked >>> return ()

```

Each **interaction** is uniquely defined in terms of its origin (a graphical object) and its (Tk) event pattern. The class **GUIObject** defines the class of graphical objects such as widgets, whereas the class **GUIEvent** defines the class of event patterns (e.g. a key press or a mouse motion). The type **GUIEventInfo** represents the event descriptor returned when a Tk event occurs, which amounts to mouse positioning or key press information. The registration command (**regGUIeid**) registers the event with the Tk adaptor, and informs the adaptor that the event is to be synchronously delegated to exactly one listener. The events associated with active widgets, such as button clicks, are then defined by further specializations of the interaction event.

4.3 Example

We can now define a controller for the counter widget and the persistent object of Fig. 5. It may sound as a simple and artificial example, but it actually demonstrates the principles used to construct complex views of the repository such as version and configuration graphs.

The controller is defined as a composition of 3 events:

```
controller cnt (
  updated cnt          >>>= (\v -> objAttrSet anm v)
  +> objModified obj anm >>>= (\v -> setValue cnt v)
  +> destroyed obj     >>> do {destroy cnt; stop}
) where anm = countAttr obj
```

The first event-action pair ensures that the count attribute associated with the persistent object **obj** is set every time that the user changes the count by interacting with the counter widget **cnt**. The second event-action pair is set up to react to change notifications coming from the Repository Manager. The event **objModified obj anm** occurs whenever the attribute **anm** associated with the persistent object **obj** is changed. The reaction to this event is to change the view by updating the count. The last event-action pair is set up to terminate the interactor should the persistent object be destroyed. The interactor will automatically deregister all registered events in such a case.

The **updated cnt** event is a composite event that listens to the button clicks, and returns the new count when triggered. It also updates the entry widget to display the new count behind the scene:

```
updated (Counter ent bi bd) =
  clicked bi >>> updValue ent succ
  +> clicked bd >>> updValue ent pred
```

5 Repository Manager

The UniForM *Repository Manager* serves as persistent store for software objects like specifications, proofs, programs, test runs, manuals etc. These objects are

linked together by *development links* (revision, refinement) and various *dependency links*. e.g. import links, test-spec links, documentation links etc. Thus from an abstract point of view, the repository is a persistent graph machine.

The Repository Manager is based on a Haskell encapsulation of H-PCTE. Basically, H-PCTE is a persistent machine for attributed graphs, which extends a traditional file system with the following features:

- attributed objects and links,
- object and link types,
- long field (i.e. bulk) attributes,
- fine grained links,
- transactions and
- change notifications.

What makes H-PCTE attractive to a functional programming environment is that it is based on the garbage collection paradigm. One object of H-PCTE is designated as the persistent root. Links between objects can be explicitly removed, whereas objects are garbage collected whenever they can't be reached from the persistent root any more.

The Haskell API abstracts over the underlying functionality provided by H-PCTE in terms of classes that are set up to provide generalized functionality of relevance to objects, links and attributes. Change notifications from H-PCTE are furthermore represented as first class composable event values by interposing an adaptor between H-PCTE and the interactors of the application.

We shall briefly introduce some of the commands of the Repository Manager, and demonstrate how they can be used to define some of the computations needed for managing versions of Haskell modules. Essentially, the version graph is a directed acyclic graph, where the nodes denote module revisions and the edges revision links (see Fig. 1).

5.1 Objects

Persistent objects are instances of the class `OMSObjectC`. Most persistent objects are uniquely identified by a *persistent object identifier*. This identifier can be retrieved by the `getPOId` method of class `OMSIdObjectC` (see Fig. 7). The reason why we have two classes, `OMSObjectC` and `OMSIdObjectC`, is that predefined objects of H-PCTE, such as the persistent root, have no associated object identifier.

All objects can have attributes and the class `OMSAttributedObjectC` provides operations for updating and retrieving the attributes associated with a persistent object. The definition as multi-parameter type class constrains an attribute to be used only for specific objects and attribute value types. The computation `objAttrGet (anm o)`² retrieves the value of the attribute `anm`

² We use in several places the abbreviations `obj` for `object`, `lnk` for `link` and `attr` for `attribute`.

```

class OMSObjectC o

class OMSObjectC o => OMSIdObjectC o where
  getPOId    :: o -> IO POId

class (OMSObjectC o,OMSObjectAttributeC a,OMSAttributeValueC av) =>
  OMSAttributedObjectC o a av where
  objAttrGet  :: IO (a o) -> IO av
  objAttrSet  :: IO (a o) -> av -> IO ()

```

Fig. 7. Persistent Object Classes

associated with the persistent object `o`, whereas the computation `objAttrSet (anm o) v` sets the attribute `anm` of object `o` to the value `v`.

The classes `OMSObjectAttributeC` and `OMSAttributeValueC` abstract over *attribute designators* and *attribute values* - respectively. The definition of these classes is actually not essential for the exposition here and so will be omitted.

Haskell modules are persistent objects of type `HaskellObj`. The following instantiations provide the computations for accessing the attributes and the global identification of the object:

```

data HaskellObj
data HaskellModuleNameObjAttr o

instance OMSObjectC HaskellObj
instance OMSIdObjectC HaskellObj

instance OMSAttributedObjectC HaskellObj
  HaskellModuleNameObjAttr String

```

The computation `getModuleName` that retrieves the name of a Haskell module can now be defined by the following function:

```

getModuleName :: HaskellObj -> IO String
getModuleName hobj = objAttrGet (moduleNameObjAttr hobj)

```

The function `moduleNameObjAttr` is just an attribute designator that denotes the name of the Haskell module. Other attributes are then used to hold the content, description, version number and creation date. These attributes are accessed in a similar way.

Attribute values of the Repository Manager amount to predefined and scalar H-PCTE attribute types as well as Haskell values. The Haskell values that can be stored on the level of H-PCTE must be instances of class `Read` and `Show`. In other words, we cannot store functions on the level of the persistent store.

5.2 Links

Links are binary relationships parameterized over the type of the source and target object. Again, we use Haskell classes to structure the application interface. The class of links is defined by the class `OMSLinkC` - a multi-parameter type class parameterized over the type of the link `l`, the type of the source object `s` and the type of the destination object `d` (see Fig. 8). The class provides two methods: `lnkSource l` returns the source object of the link `l`, and `lnkDestination l` returns the destination object of the link `l`.

```
class (OMLObjectC s,OMLObjectC d) => OMSLinkC l s d where
  lnkSource      :: (l s d) -> IO s
  lnkDestination :: (l s d) -> IO d

  objGetLinks    :: s -> IO [(l s d)]
```

Fig. 8. Links

A revision link for Haskell modules is for example defined by the following type and instance declaration:

```
data RevisionLnk r1 r2

instance OMSLinkC RevisionLnk HaskellObj HaskellObj
instance Destructible (RevisionLnk HaskellObj HaskellObj)
```

Deletion works for links by invoking the computation `destroy l` provided by class `Destructible`, whereas objects are garbage collected whenever they can't be reached from the persistent root any more. Object deletion can however be emulated by an operation that removes all links having the given object as destination. This approach of H-PCTE fits very well with the paradigms of functional programming.

When generating a version graph we need to get hold on the revisions of a given object. The computation `objGetLinks o` retrieves all links of a specific type originating from object `o`. The operation `getHaskellRevisions o`, which returns the revisions of a given Haskell object by retrieving the destinations of all `RevisionLnk` links emerging from `o`, is therefore quite easily defined as:

```
getHaskellRevisions :: HaskellObj -> IO [HaskellObj]
getHaskellRevisions o = do
  lnks <- objGetLinks o ::
    IO [RevisionLnk HaskellObj HaskellObj]
  mapM lnkDestination lnks
```

5.3 Change Notifications

H-PCTE is an *active database* that allows an application to receive notifications whenever the repository has changed. The kind of events reported by the Repository Manager amount to object "deletion" events (instance `Destructible`), link deletion events (instance `Destructible`), link append events (`lnkAppended`) and attribute modification events (`objModified`).

The notifications are implemented in terms of a single `listenOMSObj` event, that is parameterized over the class of repository event designators `OMSObjectEventC` (Fig. 9). The union type `OMSEventInfo` represents the event descriptor returned when a notification event occurs. The event is a simple specialization of the `listen` event.

```
listenOMSObj :: (OMSIDObjectC o,OMSObjectEventC e) =>
  o -> e -> EV OMSEventInfo
```

Fig. 9. Change Notification Event

The events provided to the casual application developer are straightforward refinements of the `listenOMSObj` event. For example, the event that occurs when a Haskell revision is garbage collected is defined as:

```
instance Destructible HaskellObj where
  destroyed o = listenOMSObj o ev >>> return ()
    where ev = OMSObjEvent HPcteObjDelete
```

5.4 Versioned Objects

So far we have introduced some of the generic features of the Repository Manager. What we are really interested in is to use this basis to represent the version (and configuration) graphs of the Hugs WorkBench.

```
class (OMSIDObjectC vo) => OMSTVersionedObjectC vo where
  revise      :: vo -> IO vo
  merge       :: vo -> vo -> IO vo
  getRevisions :: vo -> IO [vo]
  revised     :: vo -> EV vo
```

Fig. 10. Versioned Objects

Versioned objects are instances of class `OMSTVersionedObjectC`, i.e. the version model is parameterized. The following base computations and events are relevant to version management (see Fig. 10):

- **revise** *o* creates a new revision of object *o*.
- **merge** *o1 o2* creates a new revision *o* which is a merge of *o1* and *o2*.
- **getRevisions** *o* returns the immediate revisions of object *o*.
- **revised** *o* occurs whenever a new revision has been made to object *o*.

It is quite straightforward to define the operations in terms of the notification event provided by the Repository Manager. For example, the **revised** event is defined as:

```
instance OMSVersionedObjectC HaskellObj where
  getRevisions o = getHaskellRevisions o
  revised o =
    (lnkAppended o ltRevision ::
     EV (RevisionLnk HaskellObj HaskellObj)) >>>=
    lnkDestination
```

The **lnkAppended** event occurs when a new revision of the given object appears. The result of this event is to return the identity of the revision link. With this link as a handle, **lnkDestination** is called to retrieve the identity of the new revision, which is then returned as the result of the **revised** event.

6 Version Graph

So far we have seen fragments of code defining the interfaces to the repository and the approach to event handling. Time has come to glue everything together by defining a view over the repository that can be maintained consistently on the fly in a multiuser setting. The view is constructed by using the services of the *User Interaction Manager*. Interactors are then set up that respond to user interactions and repository change notifications. The principles shall be demonstrated by building and maintaining a *version graph*.

Graphs are created and visualized using a Haskell interface to the graph visualization system daVinci. The following computation creates a new window that displays the version graph associated with a given Haskell module:

```
newVersionGraph :: HaskellObj -> IO Graph
newVersionGraph hobj = do
  g <- graph [title "Development"]
  showNode g hobj
  buildGraph g hobj
  displayGraph g
  return g
```

First an empty graph is created and the root of the version graph is visualized calling **showNode**. The entire version graph is then built and displayed. The computation **showNode** retrieves the relevant attributes from the repository, e.g. the name (calling **getModuleName**) and the version number, and displays the version as a daVinci node.

A version graph is an acyclic directed graph. The view is transitively constructed by retrieving all revisions associated with the versioned Haskell module:

```
buildGraph :: Graph -> HaskellObj -> IO ()
buildGraph g hobj = do
  revs <- getRevisions hobj
  foreach revs (\rev -> do
    t <- isVisualized g rev
    unless t ( do
      showRevision g hobj rev
      registerEvents g rev
      buildGraph g rev
    )
  )
```

A revision is visualized in terms of a revision edge pointing from the source to the target object. The new node and the new edge are added to the version graph by calling the operations `showNode` and `showEdge`, respectively:

```
showRevision :: Graph -> HaskellObj -> HaskellObj -> IO Edge
showRevision g hobj rev = do
  showNode g rev
  showEdge g (RevisionLnk hobj rev)
```

Objects and links of relevance to a version graph may come and go during the lifetime of the view. The view is kept consistent with the underlying repository by a bunch of interactors - one for each node of the version graph. Each controller is set up to listen to a `revised` and a `destroyed` event:

```
registerEvents :: Graph -> HaskellObj -> IO InterActor
registerEvents g hobj = controller g (
  revised hobj >>>= (\rev -> do
    showRevision g hobj rev
    registerEvents g rev
    redrawGraph g)
  +> destroyed hobj >>> do {
    destroy (g,hobj);redrawGraph g; stop}
)
```

When a notification arrives that a new revision has been made, the revision is visualized and a new controller for the graph is created to monitor changes to it. The graph is then redrawn. Similar, when a notification arrives that the revision has been destroyed, the *daVinci* node is destroyed and the interactor stops. The controller will furthermore stop interaction if the graph is deleted, i.e. if the user closes the window displaying the graph.

7 Related Work

The provision of integrated program environments for functional languages have previously been addressed within the Napier project [5], but in a homogeneous and persistent language setting. The integration of foreign tools has not been taken into account. Nowadays, SDEs are constructed by integrating off-the-shelf tools in order to reduce development costs [26]. The architecture of the UniForM WorkBench reflects this trend, by providing generic integration services for data, control and presentation integration.

ToolBus [2] is a similar system using adaptors and a network of communicating agents to integrate tools. The integration is expressed in an extremely simple process and term language, which does not compare to Haskell in power and elegance. The idea of associating registration and deregistration commands with tool events can be traced back to the JavaBeans architecture [19]. The WorkBench improves on JavaBeans by providing automatic registration of events, and by representing events in terms of first class, composable values. A functional language is a prerequisite in coming up with these abstractions.

The provision of graphical user interfaces in functional languages has been a vivid track of research over the past years [10] [20] [6] [8] [18] [28]. The UniForM WorkBench goes one step further by providing a uniform approach to event handling that is independent of the actual source of the event. User interactions, repository change notifications, operating system events and tool events are all represented as first class composable event values.

The UniForM Concurrency Toolkit [21, 13] has found inspiration in CML, but extends CML with infix operators and external events of the reactive system. Facile [9] is another, and similar system, but unlike Facile, all threads are running within the same process. The focus of Facile is on homogenous distribution within one language framework, whereas the WorkBench supports the integration of foreign tools. There are some similarities though: Facile's two-sorted syntax of behaviour expressions and pure expression is to some extent similar to the domains of computations (IO) and event values (EV).

8 Future Work

An interesting track of future research is to extend the WorkBench with tools for cooperative work and development management. This requires that development rules can be expressed, which in turn suggest that the framework must be extended with "rules", i.e. events triggered by changes to object collections (types) rather than individual instances.

The repository manager could be made more elegant in the presence of orthogonal persistence [27]. Having persistent functions at hand, meta information - such as for example the computation for visualizing Haskell modules - would be stored on the level of the persistent store. A more flexible environment, that could be instantiated on the fly rather than being precompiled, could then be achieved.

9 Conclusion

The UniForM WorkBench, which provides generic services for data, presentation and control integration in Concurrent Haskell, has proved itself to be a viable vehicle for developing integrated systems given prefabricated components. A key feature, beyond the usual advantage of using a functional programming language, is the higher order model to event handling, which extends CML with first class, fully composable tool events such as user interactions and database change notifications.

With this technological basis, it is possible to take the discipline of implementing functional languages one step further, since not only the compiler, but the entire SDE can be developed using the functional language itself. The paper demonstrates how one aspect of such a development task, namely that of maintaining consistent views over a distributed multiuser repository, can be achieved very elegantly without loss of abstraction.

Acknowledgement

We would like to thank Walter Norzel for implementing parts of the *UniForM Concurrency Toolkit*, Carla Blanck Purper for implementing some of the underlying features of the *daVinci* encapsulation, and last but not least, Michael Fröhlich and Mattias Werner for providing *daVinci*.

References

- [1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, Massachusetts, 1986.
- [2] J. A. Bergstra and P. Klint. The Toolbus - a Component Interconnection Architecture. Technical Report P9408, University of Amsterdam, 1994.
- [3] The H-PCTE Crew. H-PCTE vs. PCTE, version 2.8. Technical report, Universität Siegen, June 1996.
- [4] ECMA. *Portable Common Tool Environment (PCTE) — Abstract Specification*. European Computer Manufacturers Association, 3 edition, December 1994. Standard ECMA-149.
- [5] A. M. Farkas, A. Dearle, G. N. C. Kirby, Q. I. Cutts, R. Morrison, and R. C. H. Connor. Persistent Program Construction through Browsing and User Gesture with some Typing. In *Proc. 5th International Workshop on Persistent Object Systems*. San Miniato, Italy, 1992.
- [6] S. Finne and S. Peyton Jones. Composing Haggis. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms for Computer Graphics*. Springer, 1995.
- [7] M. Föhlich and M. Werner. daVinci V2.0.3 Online Documentation. Universität Bremen, <http://www.informatik.uni-bremen.de/~davinci>, 1997.
- [8] T. Frauenstein, W. Grieskamp, P. Pepper, and M. Südholt. Communicating Functional Agents and their Application to Graphical User Interfaces. In *Proceedings of the 2nd International Conference on Perspectives of System Informatics, Novosibirsk*, Lecture Notes in Computer Science. Springer, 1996.

- [9] A. Giacalone, P. Mishra, and P. Sanjiva. Facile: A Symmetric Integration of Concurrent and Functional Programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.
- [10] M. Hallgren and M. Carlsson. Programming with Fudgets. In *First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*. Springer, May 1995.
- [11] M. P. Jones and J. C. Peterson. *Hugs 1.4, The Nottingham and Yale Haskell User's System*, April 1997.
- [12] E. W. Karlsen. Integrating Interactive Tools using Concurrent Haskell and Synchronous Events. In *CLaPF'97: 2nd Latin-American Conference on Functional Programming*, September 1997.
- [13] E. W. Karlsen. The UniForM Concurrency Toolkit and its Extensions to Concurrent Haskell. In *GWFP'97: Glasgow Workshop on Functional Programming*, September 1997.
- [14] E. W. Karlsen. The UniForM User Interaction Manager. Technical report, FB 3, Universität Bremen, Germany, 1998.
- [15] E. W. Karlsen. The UniForM WorkBench - a Higher Order Tool Integration Framework. Technical report, FB 3, Universität Bremen, Germany, 1998.
- [16] G. Krasner and S. Pope. A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.
- [17] B. Krieg-Brückner, J. Peleska, E. R. Olderog, D. Balzer, and A. Baer. Universal Formal Methods Workbench. In U. Grote and G. Wolf, editors, *Statusseminar des BMBF: Softwaretechnologie*. Deutsche Forschungsanstalt für Luft- und Raumfahrt, Berlin, 1996. Available at <http://www.informatik.uni-bremen.de/~uniform>.
- [18] C. Lüth, S. Westmeier, and B. Wolff. sml.tk: Functional Programming for Graphical User Interfaces. Technical Report 8/96, FB 3, Universität Bremen, 1996.
- [19] Sun Microsystems. *JavaBeans 1.0*. JavaSoft, December 1996.
- [20] R. Noble and C. Runciman. Gadgets: Lazy Functional Components for Graphical User Interfaces. In *PLILP'95: Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, volume 982 of *LNCS*, pages 321–340. Springer, September 1995.
- [21] W. Norzel. Building Abstractions for Concurrent Programming in Concurrent Haskell. Master thesis (in german), FB 3, Universität Bremen, Germany, April 1997.
- [22] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.
- [23] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Principles of Programming Languages '96 (POPL'96)*, Florida, 1996.
- [24] S. Peyton Jones and P. Wadler. Imperative Functional Programming. In *Proc. 20th ACM Symposium on Principles of Functional Programming*, 1993.
- [25] J. H. Reppy. *Higher-Order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, 1992.
- [26] D. Schefström and G. van den Broek. *Tool Integration*. Wiley, 1993.
- [27] P. Trinder. *A Functional Database*. PhD thesis, Department of Computer Science, University of Glasgow, 1990.
- [28] T. Vullings and W. Schulte. The Design of a Functional GUI Library using Constructor Classes. In *Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*. Springer, 1996.
- [29] S. Westmeier. Verwaltung versionierter persistenter Objekte in der UniForM WorkBench (UniForM OMS Toolkit). Master thesis, Bremen University, Germany, January 1998.

Interactive Functional Objects in Clean

Peter Achten* and Rinus Plasmeijer

Computing Science Institute, University of Nijmegen, 1 Toernooiveld, 6525 ED,
Nijmegen, The Netherlands (peter88@cs.kun.nl, rinus@cs.kun.nl)

Abstract. The functional programming language Clean has a high level I/O system (version 0.8) in which complex yet efficient interactive programs can be created. In this paper we present its successor (version 1.0), the *object I/O system*. We consider some of the design considerations that have influenced the design of the new I/O system greatly. Key issues are *compositionality*, *orthogonality*, and *extensibility*. Apart from design, the object I/O system improves on its predecessor by two major contributions: programmers can introduce *polymorphic local state* at every (collection of) user interface component(s) and programmers can create *interactive processes* in a flexible way. All interface components can communicate with each other by sharing state but also using powerful message passing primitives in both synchronous, asynchronous, and uni- or bi-directional way. Using the message passing mechanism *remote procedure calling* can be added easily. The result is an *object oriented* I/O system. As in the previous system the uniqueness type system of Clean, offering the possibility to do destructive updates in a pure functional framework, plays a crucial role. Furthermore, the object I/O system makes extensive use of new type system facilities, namely *type constructor classes* and *existential types*.

1 Introduction

In the pure, lazy, functional programming language Clean [5] [16] [17] one can develop real world applications using a high level I/O system, released as version 0.8 [3], [2]. It is available for programmers in the form of a library. The major part of the library is written completely in Clean, and only a small part interfaces with the operating system. The whole I/O system is defined in a pure functional framework. The I/O system has continued to evolve and various new advanced features have been added. The two major contributions are the ability to introduce *polymorphic local state* at every user interface element, and to create *interactive processes* dynamically that communicate by means of *message passing*. These new features allow programmers to construct programs in an *object oriented* way. The results of these research activities can be found in [18]. While incorporating these ideas in the Clean I/O system we have further refined and changed these concepts. This labour has resulted in the new version 1.0 of

* Supported by STW under grant NIF33.3059: The Concurrent Clean System.

the I/O system, released recently. In this paper we discuss the concepts that distinguish this new I/O system, the *object I/O system*, from its predecessor.

There are many solutions to handle I/O in a functional system (see Section 6 for a discussion of other systems). There are a number of features that makes the old I/O system special. In a nutshell these are:

- The Clean programming language employs the type system, the *uniqueness type system*, to guarantee that functions have single threaded access to arguments [20] [4]. This makes it possible to deal with side-effects in a flexible way without sacrificing the purity of the language.
- Based on the uniqueness type system the I/O system of Clean can use the *world as value* paradigm. This paradigm is explained in detail in [18]. Briefly, every interactive Clean program is a function of type `*World -> *World`. `World` is an abstract type which represents the complete environment of the program. The `*` in front of a type constructor states that that type is unique, enabling the destructive update of a value of that type. The `World` itself is composed of sub worlds (which may also be unique), for instance the file system and event stream. The file system is again composed of unique sub worlds, the individual files. This paradigm has two major advantages over other paradigms:
 - Real world resources are *first class citizens*: they can be passed along explicitly by functions. There is no need to encapsulate them in one single abstraction, like for instance the monadic approach [21]. The type system guarantees that the program never duplicates unique resources.
 - Programs can access an arbitrary number of real world resources without fixing a specific order of evaluation because each of these resources can be manipulated independently, and, in principle, even in parallel.
- Graphical user interfaces are specified on a high level of abstraction. Specifications are given by defining appropriate instances of predefined *algebraic data types* and *higher order functions*. With the algebraic data types a programmer describes what user interface elements should be created by the system (for instance the definition of a menu along with its items). These definitions are parameterised with functions that define what action should occur (for instance in case a menu item is selected). These functions are the callback routines of the user interface. The only other thing the programmer needs to do is to specify an initial program state of arbitrary type which holds the data required during evaluation of the program. Because the specification elements are all values they can be interpreted easily by a special library function `startIO` which does all the complicated and low level event handling. The semantics of a program is that of a simple *state transition system*. The state consists of the program state and an abstract data type, the `IOSt` (pronounce ‘I/O state’) in which the information of the concrete interface elements is maintained. The callback functions are of course the state transition functions. The program terminates as soon as one of the callback functions has applied the library function `quitIO` to the `IOSt`. `quitIO` closes all currently open user interface elements and returns a special empty `IOSt` value. This causes termination of `startIO`.

- Because of the high level of abstraction the system is portable to a wide variety of very different and incompatible platforms. Currently ports exist for Macintosh, X Windows, Linux, Microsoft Windows(95 and NT), and OS/2. Each port takes care that the high level specification is mapped to its corresponding platform maintaining the required look and feel.

As an example consider the following (subset of) the algebraic data types that are used in the old I/O system to define menus:

```

:: MenuDef      s io
= PullDownMenu Id MenuTitle      SelectState [MenuElement s io]
:: MenuElement s io
= MenuItem      Id ItemTitle KeyShortcut SelectState (MenuFunction s io)
| MenuSeparator


```

A possible instance of a menu definition is `menu`, shown below together with a picture of its mapped look on a Macintosh platform.

```

menu :: MenuDef s (IOSt s)
menu = PullDownMenu menuId "File" Able
      [ MenuItem id1 "Open..." NoKey Able   open
      , MenuItem id2 "Close"      NoKey Unable close
      , MenuSeparator
      , MenuItem id3 "Quit"       NoKey Able   quit
      ]

```



This data structure is interpreted by the I/O system which will generate a concrete menu. This menu will be a pulldown menu with the title “File”. The `Able` attribute denotes that the menu is selectable. To change the menu afterwards the program uses the `Id menuId`. The “File” menu has four elements, three of which are commands (`MenuItem`) and one is a visual separator (`MenuSeparator`). If the user selects in one way or another the “Quit” menu item then the callback function of “Quit”, `quit`, will be evaluated. The type of a callback function is in general: `s -> IOSt s -> (s, IOSt s)`, with `s` the program state. A typical definition of `quit` is the following:

```

quit :: s -> IOSt s -> (s, IOSt s)
quit programstate iostate = (programstate, quitIO iostate)

```

For any program state and `IOSt quit` will terminate the application because it applies `quitIO` to its `IOSt` argument.

The release of the old I/O system has enabled us to gather feedback from ourselves and external users. On our own part we investigated the expressive power and efficiency of the system. Two examples of large applications that we have created in Nijmegen are the Clean programming environment and a functional spreadsheet [8]. The first application is interesting because it demonstrates that the efficiency of Clean and the I/O system is so good that full screen editors and the like can be created and used in daily practice. The Clean programming environment comes with the standard distribution of the Clean system. The latter application is interesting because it consists of three independently developed components: a spreadsheet, a symbolic evaluator and small proof tool, and a

text editor that have been composed afterwards into one application. A thorough early elaborate external examination is [14]. From the implementation of various ports we were able to evaluate the level of abstraction.

Not only the I/O system has been subjected to reflection and feedback. The Clean language itself has also been improved, partially due to this research. The most important changes to Clean with respect to this project concerns its type system which has been extended with *record types*, *existential types* [13] and *type constructor classes* [12].

These experiences enabled us to improve the design of the I/O system considerably with respect to *compositionality*, *orthogonality*, and *extensibility*. See Section 2.

Apart from improvements on the design of the I/O system two more fundamental issues required thorough investigation:

- Interactive programs use the program state to store information they need during evaluation. For small programs this is usually fairly trivial and causes no problems, but as programs grow things get complicated quickly. Because the program state is in principle accessible by every callback function it is difficult to guarantee the integrity of data. For this reason the new I/O system should allow programmers to add *polymorphic local state* at every (collection of) interface component(s). This is discussed in Section 3.
- The functional spreadsheet case study [8] illustrates the need for a flexible set of combinators that allow programmers to merge independently developed interactive programs into a larger one. In the old I/O system this composition is limited to (arbitrarily deep) nested composition. The ultimate goal is a set of combinators in which interactive programs can be created and destroyed as independent *interactive processes*. *Message passing* takes care of the interprocess communication. This is discussed in Section 4.

Being able to encapsulate polymorphic state in the form of interactive processes and having a flexible means of message passing turns the new I/O system in an *object oriented* I/O system. This issue is surveyed in Section 5. We end this paper with a brief summary of related work in Section 6 and present conclusions and future work in Section 7.

2 Design Improvement

One of the factors that helped us to improve the design of the I/O system was the addition of many new language facilities in Clean.

The introduction of record types helps a lot when programming interactive applications. Before the introduction of record types the program state was usually defined by an algebraic data type. Access to program state components was done by pattern matching. When changing the definition of the program state (a frequently occurring action while developing programs) all functions that pattern match the program state needed to be rewritten. Of course, using a disciplined style of introducing (abstract) access functions right away should be used, but

this also requires work. Records make the definition of functions more independent of changes to the record type.

Another change to Clean is a syntactic feature which improves the readability and comprehension of interactive functions. Local expressions when preceded by the keyword `#` introduce a new lexical scope. Right hand side identifiers belong to the scope preceding the expression in textual order. Left hand side identifiers belong to the new scope. In this way identifiers can be reused. Comprehension is aided because evaluation order corresponds with the textual order. This is particularly rewarding when passing around environments. Example 1 illustrates the use of `#` syntax using two imaginative functions `getchar` and `putchar` to implement an `echo` function. In example 2 the semantically equivalent program will be generated by the compiler after eliminating these scopes.

```
echo :: *World -> *World
echo world
#  (c,world) = getchar world
#  world      = putchar c world
|  c=='\n'    = world
|  otherwise  = echo world
```

```
echo :: *World -> *World
echo world
|  c=='\n'    = world2
|  otherwise  = echo world2
where
    (c,world1) = getchar world
    world2     = putchar c world1
```

Example 1: echo with `#` syntax

Example 2: echo without `#` syntax

The design of the I/O system has been improved greatly by sticking to the following relatively simple design decisions:

- The program state and `IOWt` values are now collected in a new record type `PSt` (pronounce ‘process state’). Its type definition is `*PSt p = {ps::p, io::*IOWt p}`. All callback functions have the same basic type, `(PSt p) -> (PSt p)`, the identity type on process state. This eases function composition.
- All attributes for which sensible default values can be chosen have been made optional. For instance, for a menu item definition its title must be provided by the programmer whereas its id, selection and mark state, short key access, and even callback function are optional. Definitions become more concise and it is easier for the library designer to introduce new attributes without changing existing programs. One nasty property of the old I/O system is that every interface component needs to be identified by an `Id` value even if the component will never be changed. Making the `Id` attribute optional frees programmers from this pain.
- All interface components can be created and destroyed dynamically. This gives the programmer the ability to change the layout and content of the user interface of a program easily.

Another major improvement in the design of the I/O system concerns windows and dialogues. In the object I/O system the definitions of these ubiquitous user interface elements have been unified. Now both windows and dialogues can contain arbitrary complex and recursive controls. The ability to regard a composition of controls as a new single control, the *compound control*, increases the expressive power of the layout language considerably. Controls can be positioned

relative to the frame of its parent component (window, dialogue, or compound control), but also relative to other controls or at absolute locations.

Finally, because we have gained a lot of implementation experience due to the ports to many different platforms, we have learned how to arrange the object I/O system in such a way that it anticipates differences between these platforms.

3 Polymorphic Local State

In this section we present the techniques that we have applied to incorporate polymorphic local state into the object I/O system. We will first illustrate our intention by means of a small example of a counter in a system without local state. For this purpose we use the following algebraic data types to define controls:

```

::Control ps      = TextControl      String      [ControlAttribute ps]
                  | ButtonControl    String      [ControlAttribute ps]
                  | CompoundControl [Control ps] [ControlAttribute ps]
::ControlAttribute ps = ControlId      Id
                  | ControlPos        ItemPos
                  | ControlFunction (ps -> ps)

```

The example counter consists of a display that shows the current value of an integer. This value can be decremented and incremented by the user by pressing two buttons labeled "-" and "+" respectively. Let `init` be the initial value of the counter. For conciseness we assume that the program state consists of the count value only. The counter is defined by the following expression:

```

counter :: Id -> Control (PSt Int)
counter dispid
= CompoundControl
  [ TextControl (toString init)
    [ ControlId dispid, ControlPos Center ]
  , ButtonControl "-"
    [ ControlFunction (upd dispid (-1))
      , ControlPos Center
    ]
  , ButtonControl "+"
    [ ControlFunction (upd dispid 1)
    ]
  ] []

```

counter

The function `toString` is an overloaded function that converts its argument to a string. A text control is used to represent the integer display. It is identified by the `Id` value `dispid`. The text of a text control can be changed afterwards by the function `setControlText` which expects this `Id`. In the object I/O system `Ids` are abstract values that can be generated only using a suitable environment (`World` and `IOSt`). The callback function of the decrement and increment button is defined as follows:

```

upd :: Id -> Int -> PSt Int -> (PSt Int)
upd dispid dx {ps=count,io}
= {ps=newcount,io=setControlText dispid (toString newcount) io}
where newcount = count+dx

```

So selection of the "-" ("+") button adds the value -1 (1) to the current count value that is located in the program state and updates the display.

For a small program using one program state for information that is maintained by a part of the program does not cause any problems. As discussed in the introduction, when programs get more complex it becomes increasingly difficult to guarantee the integrity of the program state.

What we really want is to change the I/O system in such a way that the program state contains only the *global information*, and that where needed, interface components can have their private *local state*. The I/O system has to prevent other parts of the program from having access to this local state. In case of the counter we want to associate the integer counter explicitly with the counter. Therefore, if the callback function of one of the counter controls is evaluated it should not be a function of type $(\text{PSt Int}) \rightarrow (\text{PSt Int})$ but of type $(\text{Int}, \text{PSt p}) \rightarrow (\text{Int}, \text{PSt p})$. In this way we can define `upd` by:

```
upd :: Id -> Int -> (Int, PSt p) -> (Int, PSt p)
upd dispid dx (count, ps)
= (newcount, appPIO (setControlText dispid (toString newcount)) ps)
where newcount = count+dx
```

(The library function `appPIO` updates the `io` component of a (PSt p) record using an (IOSt p) transition function argument.)

There are two obstacles when attempting to redefine the algebraic type definition `Control` as given in the beginning of this section to accomodate arbitrary compositions of controls with local state. Firstly, `Control` has only one parameter `ps` which represents the process state. We need to extend it with an alternative in which we associate a local state of type `ls` with a recursive control definition that expects a local state of type `ls` and a process state of type `ps`. This can only be a pair, and so the type of the recursive controls will be `Control (ls, ps)`. This continues recursively for each additional local state. This is of course not the kind of compositional behaviour that we are looking for. For instance, one can not add controls that have a different local state to such recursive controls. Secondly, lists can not be used to construct recursive collections of controls with arbitrary local state because their elements have to have the same type.

More powerful glue is required to construct a set of types in which interface elements can be composed. This glue is provided by Clean *type constructor classes*, similar to Gofer [12]. The type constructor classes restrict what elements can be glued, while their instances define what the elements are. We first define the instance elements. This is done by promoting the alternatives of the set of algebraic types to individual type constructors. At positions where lists are used to create recursive elements a type constructor variable is introduced. The local state of an element is identified by the `ls` type parameter, while the process state of an element is identified by the `ps` type parameter. So, the algebraic type `Control` above is changed into:

```
:: TextControl      ls ps
=   TextControl     String  [ControlAttribute (ls, ps)]
:: ButtonControl    ls ps
```

```

=   ButtonControl   String   [ControlAttribute (ls,ps)]
::   CompoundControl c ls ps
=   CompoundControl (c ls ps) [ControlAttribute (ls,ps)]

```

The type constructor class `Controls` governs which elements can be glued to form `Controls` instances:

```

class Controls c where
openControls :: Id -> ls -> c ls (PSt p) -> PSt p -> PSt p

instance Controls TextControl
instance Controls ButtonControl
instance Controls (CompoundControl c) | Controls c

```

This definition states that a control is a `TextControl`, or a `ButtonControl`, or a `CompoundControl` of `c` provided that `c` is also a `Controls` instance.

To glue elements that operate on the same local state of type `ls` and process state of type `ps` the infix type constructor combinator `:+:` is predefined:

```

::   :+: t1 t2 ls ps = (:+:) infixr 9 (t1 ls ps) (t2 ls ps)

```

Given two expressions `e1::(t1 ls ps)` and `e2::(t2 ls ps)` for appropriate actual types `t1` and `t2`, one can glue them as `(e1 :+: e2)::(:+: t1 t2 ls ps)`. For instance, using `:+:` and the new type constructors for controls we can rewrite the original definition of `counter` as follows:

```

counter dispid
= CompoundControl
  (   TextControl (toString init) [ControlId dispid,   ControlPos Center]
  :+: ButtonControl "-" [ControlFunction (upd dispid (-1)), ControlPos Center]
  :+: ButtonControl "+" [ControlFunction (upd dispid 1)]
  )   []

```

This definition not only differs in the use of the `:+:` constructor instead of list constructors, it also has a more elaborate type than its original, concise, type `Id -> Control (PSt Int)`:

```

counter :: Id -> CompoundControl (( :+: ) TextControl
                                   (( :+: ) ButtonControl
                                       ButtonControl
                                   )) Int (PSt p)

```

It is interesting to observe the structural equivalence between the type of `counter` and its definition. In our approach the types contain a lot of information about their ‘inhabitants’. Although types grow proportionally with the number of type constructors involved in the definition, programmers do not have to worry because the Clean compiler can derive these types automatically.

Using the new glue we can compose elements that operate on the same local state. To allow us to switch between different local states, the object I/O system provides two additional type constructor combinators. These combinators either introduce a *new* local state (`NewLS`) or *add* a local state (`AddLS`). In both cases a new value of arbitrary type is created. To accomodate this, the two new type

constructor combinators must introduce a type variable on the right hand side of their definition. In Clean this can be done by *existentially quantifying* these type variables. One (or more) type variable(s) can be introduced by existential quantification by placing them behind the keyword `E.` which appears immediately before the rest of the type.

```
:: NewLS t ls ps = E.new: {newLS::new,newDef::t new ps}
:: AddLS t ls ps = E.add: {addLS::add,addDef::t (add,ls) ps}
```

The type constructor combinators should be read as follows:

- To every element definition we can add an element definition `e` with a new local state of value `x` by `{newLS=x,newDef=e}`.
- To every element definition we can add an element definition `e` with an extended local state of value `x` by `{addLS=x,addDef=e}`.

We can now easily encapsulate the integer local state of `counter`:

```
counter dispid
= { newLS = init
  , newDef
    = CompoundControl
      (   TextControl (toString init)
          [ControlPos Center,ControlId dispid]
      :+ ButtonControl "-" [ControlPos Center
          ,ControlFunction (upd dispid (-1)) ]
      :+ ButtonControl "+" [ControlFunction (upd dispid 1)   ]
      ) []
  }
```

The type of `counter` effectively hides the type of the local integer state `init`:

```
counter :: Id -> NewLS (CompoundControl ((:,:) TextControl
                                          ((:,:) ButtonControl
                                              ButtonControl
                                          ))) ls (PSt p)
```

4 Interactive Processes

In this section we present the tools that are offered in the object I/O system to create and handle *interactive processes*. To illustrate the concepts we will construct a small interactive *talk* program. To clarify the account, we start with a simplified version of the object I/O system.

The example talk program creates two interactive processes. Each interactive process has a simple dialogue with four elements:

- An input field in which the text is typed that has to be *sent* to the other talk process.
- An output field in which the most recently *received* text from the other talk process is displayed.
- A button that, when selected, sends the current input text to the other talk process.
- A button that, when selected, terminates the program.

talk

As this example suggests, when dealing with processes, one also needs to communicate. The object I/O system provides functions to do *message passing*. Messages can be arbitrary values (data structures and functions). Messages are received by *receivers* which can be opened and closed dynamically.

Basic message passing will be explained step by step as we construct our example program (see Section 4.2 for more details). For the time being we assume that we have some function `send :: RId m -> m -> PSt p -> PSt p`. The abstract type `RId m` uniquely identifies a receiver that accepts messages of type `m`. When applied to such a special identifier, and a message of type `m`, `send` will send the message to the indicated receiver.

Interactive processes are defined in the same way as user interface components in the previous section. Again we simplify (see Section 4.1 for more details). Algebraic data types are applied to define interactive processes. An interactive process is a state transition system. So one component of the definition should be the initial program state, of type `p`. Initially, an interactive process has an empty user interface. It is up to the *process initialisation* attribute, of type `[(PSt p) -> (PSt p)]`, to create the user interface. The functions in the initialisation attribute are evaluated in order of appearance. The algebraic type that defines an interactive process can therefore be defined as follows:

```

:: Process          =   E.p:Process p (ProcessInit (PSt p))
:: ProcessInit ps == [ps -> ps]

```

The talk program consists of two identical interactive processes that are distinguished only by their `RId`. So they can be defined by the same function `talk`. The messages will either be a line of text (a `String`) or a message to request termination. This is given with the algebraic data type `Message = Line String | Bye`. For each `talk` process `me` identifies its own receiver, while `you` identifies the receiver of the other `talk` process, both of type `RId Message`. Because `talk` is very simple it will have no need of global nor local data, so the initial program state can be any arbitrary value. Since we have to choose a value we will use the simple type `Nil = Nil`. The initialisation actions first create the two `Ids` `inId` and `outId` that will identify the input control and the output control respectively. Then the dialogue described above is opened, followed by the receiver. (The library function `accPIO` accesses the `io` component of a `(PSt p)` record using an `(IOSt p)` access function argument.)

```

talk :: RId Message -> RId Message -> Process
talk me you = Process Nil [ initialise ]
where initialise ps
    # ([inId,outId:_],ps) = accPIO (openIds 2) ps
    talkdialog             = ...           // see below
    # ps                   = openDialog talkdialog ps
    talkreceiver           = ...           // see below
    = openReceiver talkreceiver ps

```

To define the talk dialogue we use the controls class that we defined at the end of Section 3. For the two buttons we can conveniently use the `ButtonControls`. The input and output controls will be represented with a library control that we have not introduced yet, the `EditControl`. An `EditControl` can be used to display an arbitrary large text within a frame of a certain width (in pixel units) and a fixed number of lines. Its library definition is:

```

:: EditControl ls ps
= EditControl String Width NrLines [ControlAttribute (ls,ps)]

```

We apply the `EditControl` for both the input and the output control of the dialogue. To prevent users from typing text in the output control, its `SelectState` attribute is set to `Unable`. In the I/O system controls must always be part of a window or dialogue. Now `talkdialog` can be defined as follows:

```

talkdialog
= Window "Talk"
    ( EditControl "" 200 5 [ ControlId inId, ControlPos Center ]
    :+: EditControl "" 200 5 [ ControlId outId,ControlPos Center
                              , ControlSelectState Unable          ]
    :+: ButtonControl "Send" [ ControlFunction talksend
                              , ControlPos Center                  ]
    :+: ButtonControl "Quit" [ ControlFunction talkquit
                              , ControlPos Center                  ]
    ) []
talksend ... // see below
talkquit ... // see below

```

In the object I/O system receivers are user interface elements that handle messages of some type `m`. The algebraic data type that defines receivers is:

```

:: Receiver m ls ps
= Receiver (RId m) (ReceiverFunction m (ls,ps))
               [ReceiverAttribute (ls,ps)]
:: ReceiverFunction m ps ::= m -> ps -> ps

```

The event handler of `talkreceiver`, `receive`, has a straightforward definition: if the message is `Line line`, it should set the text of the output control to `line`. If the message is `Bye`, it should terminate its parent process. The library function `closeProcess` is the new name of the `quitIO` function of the old I/O system. (The library function `noLS1` is useful to ‘lift’ a `(PSt p)` transition function to a `(ls,PSt p)` transition function.)

```

talkreceiver = Receiver me (noLS1 receive) []
where receive (Line line) ps = appPIO (setControlText outId line) ps
      receive Bye           ps = appPIO closeProcess ps

```

The only parts that remain to be defined are the two callback functions of the button controls, `talksend` and `talkquit`. The callback function of “Send”, when selected, first retrieves the current content, `line`, from the input control (using the library function `getControlText`). It then resets the content of the input control to the empty string, and sends the `Line line` message to the other talk process. The callback function of “Quit”, when selected, sends the `Bye` message to the other talk process and terminates the interactive process.

```
talksend ps
# (line,ps) = accPIO (getControlText inId) ps
# ps       = appPIO (setControlText inId "") ps
= send you (Line line) ps
talkquit ps
= appPIO closeProcess (send you Bye ps)
```

This finishes the definition of a talk process. All we need to do is to actually open two process instances. The function `startProcesses` takes a list of process definitions and creates the processes by applying their initialisation functions. `startProcesses` terminates only if there are no more interactive processes to evaluate. As for all identification values, receiver identification values are created by the system, using `openRId(s)` to obtain a (list of) value(s) of type `RId`. So the main function that creates the talk application can be defined as follows:

```
Start :: *World -> *World
Start world
# ([a,b:_,world) = openRIds 2 world
= startProcesses [talk a b, talk b a] world
```

4.1 More About Interactive Processes

In the `talk` example we have used a simplified version of the set of functions and type definitions by which one can define and create interactive processes in the object I/O system. In this section we will discuss the system in more detail.

We have seen how to construct an application that consists of two independent interactive processes that communicate by means of message passing. This is not the only way to construct programs. Consider for instance the situation in which one has created an interactive process that implements a structural text editor for some programming language, and a compiling interactive process for that source language. When glueing these processes into a programming environment application it is natural that they *share* the program text that has to be edited and compiled. Another example of state sharing is the *clipboard* that is shared by applications on graphical user interface platforms.

The object I/O system allows interactive processes to share the program state, provided of course that the types of the program states are equal. A collection of interactive processes that share a program state is called a *process group*. However, the restriction to equal types of program state is not a practical one. One would rather have a part `p` of the program state to have equal type and another part `l` to have different, arbitrary type. This is reflected in the real library `PSt` and `IOSt` types:

```
:: *PSt l p = { ls :: l, ps :: p, io :: *IOSt l p }
```

The function that creates a number of shared interactive processes is:

```
class shareProcesses pdef :: pdef p -> PSt 1 p -> PSt 1 p
```

The instances of `shareProcesses` are either individual interactive process definitions or interactive process compositions, using the new type constructor combinators `ListCS` and `:~:`. `ListCS` allows the convenient definition of interactive processes of the same type. With `:~:` arbitrary processes can be glued.

```
:: ListCS t p = ListCS [t p]
:: :~: t1 t2 p = (:~:) infixr 9 (t1 p) (t2 p)
```

Because the program state is split, the definition of an individual interactive process also has a small change. For a given shared program state `p`, each interactive process only needs to introduce a local program state `l`:

```
:: Process p = E.l:Process 1 (ProcessInit (PSt 1 p))
```

Expressed in Clean, the instances of `shareProcesses` are therefore:

```
instance shareProcesses Process
instance shareProcesses (ListCS pdef) | shareProcesses pdef
instance shareProcesses (:~: pdef1 pdef2) | shareProcesses pdef1
                                         & shareProcesses pdef2
```

In the `talk` example the function `startProcesses` opened the two talk processes. With `startProcesses` one creates an initial interactive process structure.

```
class startProcesses pdef :: pdef -> *World -> *World
```

The instances of `startProcesses` are either individual process groups or process group compositions, using lists (as we have seen in the `talk` example) and the new type constructor combinator `:^:` that glues arbitrary process groups:

```
:: :^: t1 t2 = (:^:) infixr 9 t1 t2
```

A process group is defined by the algebraic type `ProcessGroup`. It introduces a value for the shared program state component of type `p`, and some composition `pdef` of interactive processes that share `p`:

```
:: ProcessGroup pdef = E.p: ProcessGroup p (pdef p)
```

Of course, one also wants to create process groups within an interactive process. This is done with the function `openProcesses`. It has the same type as `startProcesses` except that the `World` argument and result should be of type `(PSt 1 p)`. Therefore `startProcesses` and `openProcesses` are member of the same class `Processes`. Its definition and instances are:

```
class Processes pdef where
startProcesses :: pdef -> *World -> *World
openProcesses  :: pdef -> PSt 1 p -> PSt 1 p

instance Processes (ProcessGroup pdef) | shareProcesses pdef
instance Processes [pdef] | Processes pdef
instance Processes (:~: pdef1 pdef2) | Processes pdef1 & Processes pdef2
```

4.2 More About Message Passing

In the construction of the `talk` example, we used a function `send` to send messages. In this section we present the means to do message passing in the object I/O system. Messages can be sent *asynchronous* or *synchronous*, but also *uni-directional* or *bi-directional*. Although one would assume that this results in four combinations, there are only three. The combination *asynchronous/bi-directional* is not provided because such a function can be defined in terms of two asynchronous/uni-directional message passing calls making use of the fact that receivers can be created and destroyed dynamically. We start with the simplest message passing function, the combination *asynchronous/uni-directional*:

```
asyncSend :: RId m -> m -> PSt l p -> (SendReport,PSt l p)
```

When applied to the identification of a receiver that accepts only messages of type `m` and a message of that type, `asyncSend` adds the message to the message queue of the indicated receiver. The location of the receiver is immaterial, it may well be part of the same interactive process but also of another interactive process. However, it is possible that some exceptional situation occurs. For this purpose all message passing functions return a `SendReport` which gives information about the action.

```
:: SendReport
= SendOk | SendUnknownReceiver | SendUnableReceiver | SendDeadlock
```

The only exception in case of `asyncSend` is `SendUnknownReceiver` in case the indicated receiver could not be found in any of the currently open interactive processes. If it was found then `SendOk` is returned. If the indicated receiver remains `Able` and is not closed before its message queue is empty it will at some point in time after `asyncSend` retrieve the message from the top of its message queue and apply its receiver function to the message.

Synchronous/uni-directional is the next message passing function:

```
syncSend :: RId m -> m -> PSt l p -> (SendReport,PSt l p)
```

When applied to the identification of a receiver that accepts only messages of type `m` and a message of that type, `syncSend` *blocks* its parent process, locates the indicated receiver if it exists, and if the receiver is `Able` applies the receiver function to the message, and then returns to and *unblocks* its parent process.

As this description suggests, synchronous message passing is more complicated than asynchronous message passing because it involves a *context switch*. Also more exceptions may occur. `SendUnknownReceiver` is returned in the same situation as with `asyncSend`. `SendUnableReceiver` is returned in case the receiver could be found, but is currently `Unable`. `SendDeadlock` is returned in case the receiver could be found, is currently `Able`, but has a blocked parent process that is involved in a synchronous, cyclic communication with the current process. In all exceptional cases message passing is halted, and `syncSend` returns to the current process which becomes unblocked.

Synchronous/bi-directional is the last message passing function:

```
syncSend2 :: R2Id m r -> m -> PSt l p -> ((SendReport,Maybe r),PSt l p)
```

Synchronous/bi-directional message passing is the same as synchronous/unidirectional message passing except for the fact that the involved receiver accepts not only a message of type `m`, but also returns a response of type `r`. For this reason a new identification type (`R2Id m r`) for bi-directional receivers is introduced that is parameterised with both the message type `m` and the response type `r`. Also the callback function of a bi-directional receiver is now not a function of type `m -> ps -> ps`, but of type `m -> ps -> (r,ps)`. This results in the following type definitions for bi-directional receivers:

```
:: Receiver2 m r ls ps
=   Receiver2 (R2Id m r) (Receiver2Function m r (ls,ps))
      [ReceiverAttribute      (ls,ps)]
:: Receiver2Function m r ps := m -> ps -> (r,ps)
```

The programmer uses the function `openR2Id(s)` to obtain a (list of) value(s) of type `R2Id`. For `SyncSend2` the same exception results apply as for `syncSend`. In case the communication was succesful, `SendOk` is returned, but also a response value (`Just r`). In case of an exception, the response value is `Nothing`.

5 Object Orientation

In Section 3 we have explained how polymorphic local state can be added at will by the programmer to arbitrary interface elements. In Section 4 we added interactive processes and message passing. Using these concepts it becomes possible to construct programs in an *object oriented* way. Except for inheritance, the concept of *objects* as defined in [10] with respect to Smalltalk coincides exactly with interactive processes in the object I/O system: interactive processes have local memory (the local state), inherent processing ability (obviously), and the capability to communicate with other objects (message passing between processes). *Methods* correspond closely with bi-directional receivers in the object I/O system, while invoking a method corresponds with sending the appropriate message by means of the synchronous/bi-directional `syncSend2` primitive. In principle, inheritance can be added to the Clean language.

In the object I/O system we can use the power of functional programming languages to abstract away the fact that interactive processes use message passing to invoke actions from each other. This approach has been explained in detail in [1]. It transforms a message passing protocol into a *remote procedure calling* protocol. This means that we can choose to use either a message passing style or a function application style. In the object I/O system we have applied this technique to rearrange the model of the `World`. In the object I/O system all interactive processes in principle should have access to the file system. Modeling the file system as a unique sub environment of type `Files` of the `World` causes unwanted sequentialisation. Instead, we let `Files` be contained in the program state of an interactive process, the *file server* that is always part of the `World`. The file server consists of one bi-directional receiver only. Its `Id` is globally known in the library but hidden from the programmer. The file system functions of the old I/O system are used locally by the file server. For the programmer opening

and closing files is now done in a functional style using remote procedure calls as sketched above. This new model of the world is not only more realistic, it also allows process groups to be evaluated concurrently.

6 Related Work

In Section 1 we have extensively discussed the unique features of the Clean I/O approach. In the area of functional languages a lot of solutions for graphical user interface I/O have emerged, and it is beyond the scope of this paper to compare them all. Therefore we will discuss only a few characteristic exponents.

FUDGETS [6] and *Gadgets* [15] are examples of stream processing I/O solutions. Every graphical user interface element is a stream processor. Elements have input streams and output streams (one for FUDGETS, many for Gadgets). Communication occurs by placing messages on these streams. State is local to each of the elements. ‘Global knowledge’ must be achieved by message passing.

A large class of solutions relies on monads [21] to thread one external environment. The three examples that we discuss are *Haggis* [9], *Pidgets* [19], and *TkGofer* [7]. In *Haggis* a lot of attention is paid to compositionality and extensibility. *Pidgets* has looked at the definition of user interface elements at the functional level and extends this model with an automatic hit detection mechanism. *TkGofer* has used type constructor classes to structure the graphical user interface element hierarchy.

In contrast with the object I/O system (and also the old Clean I/O system), in each of these systems the definition of a graphical user interface is an *action*. As we have illustrated extensively, in the object I/O system (and also the old Clean I/O system) graphical user interface elements are defined as data type instances. Separating definition from creation has the advantage that it is possible to inspect and modify descriptions by means of functions.

These systems also have in common that concurrency has been added to the functional language to obtain a flexible I/O model. It must be observed however that in these systems the a-priori use of concurrency in combination with monads destroys the deterministic behaviour of these systems. As an alternative to repair these effects the Brisk project [11] pays effort to introduce only deterministic concurrency.

7 Conclusions and Future Work

In this paper we have presented a pure functional I/O system that allows the definition of interactive programs on high level of abstraction. It builds on the unique characteristics of its predecessor, improving on compositionality, orthogonality, and extensibility. The capability to introduce polymorphic local state everywhere and create interactive processes that communicate by means of message passing provide the programmer with powerful tools to construct programs in an object oriented style.

Thanks to the uniqueness type system Clean can exploit the benefits of the world as value paradigm. The I/O system can be arranged in such a way that process groups can, in principle, be evaluated in parallel without loosing determinism. Shared processes inside one group have an interleaved behaviour. One of the leads to future work is to see how we can employ our knowledge of concurrency to obtain a truly concurrent implementation of the I/O system.

Acknowledgements

The authors thank all users of Clean who have contributed to the design of the object I/O system with their constructive criticism. Marko van Eekelen commented on the structure of this paper. Pieter Koopman and Arjan van IJzenendoorn helped in simplifying the system of local state object definitions.

References

- [1] P.M. Achten and M.J Plasmeijer. Concurrent interactive processes in a pure functional language. In J.C. van Vliet, editor, *Proceedings Computing Science in the Netherlands (CSN'95)*, Jaarbeurs Utrecht, The Netherlands, Stichting Mathematisch Centrum, Amsterdam, pages 10–21, 27–28 November 1995.
- [2] P.M. Achten and M.J Plasmeijer. The ins and outs of clean i/o. *Journal of Functional Programming*, 5(1):81–110, January 1995.
- [3] P.M. Achten, J.H.G. van Groningen, and M.J Plasmeijer. High level specification of i/o in functional languages. In J. Launchbury and P. Sansom, editors, *Proceedings Glasgow Workshop on Functional Programming (Ayr, Scotland)*, Workshops in Computing, pages 1–17. Springer-Verlag, 6–8 July 1993.
- [4] E. Barendsen and J.E.W Smeters. Uniqueness type inference. In M. Hermenegildo and S.D. Swierstra, editors, *Proceedings of Seventh International Symposium on Programming Languages: Implementations, Logics and Programs, Utrecht, The Netherlands*, number 982 in LNCS, pages 189–206. Springer-Verlag, 20–22 September 1995.
- [5] T. Brus, M.C.J.D. van Eekelen, M.O. van Leer, and M.J Plasmeijer. Clean: A language for functional graph rewriting. In Kahn. G., editor, *Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon, USA*, number 274 in LNCS, pages 364–384. Springer-Verlag, 1987.
- [6] M. Carlsson and Th Hallgren. sc Fudgets - a graphical user interface in a lazy functional language. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 321–330. ACM Press, 9–11 June 1993.
- [7] K. Claessen, T. Vullingsh, and E Meijer. Structuring graphical paradigms in tkgofer. In *Proceedings of the ACM SIGPLAN International Conference of Functional Programming (ICFP'97)*, Amsterdam, The Netherlands, pages 251–262. ACM Press, 9–11 June 1997.
- [8] W.A.C.A.J. de Hoon, L.M.W.J. Rutten, and M.C.J.D. van Eekelen. Implementing a functional spreadsheet in clean. *Journal of Functional Programming*, 5(3):383–414, July 1995.

- [9] S. Finne and S Peyton Jones. Composing haggis. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms in Computer Graphics, Maas-tricht, The Netherlands*. Springer-Verlag, September 1995.
- [10] A Goldberg. *Object-Oriented Programming Languages*, pages 570–607. Addison-Wesley Publishing Company, 3rd edition, 1995.
- [11] I. Holyer, N. Davies, and C. Dornan. The brisk project: Concurrent and distributed functional systems. In *Proceedings Glasgow Workshop on Functional Programming, Ullapool, Scotland*, 10–12 July 1995.
- [12] M.P Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 52–61. ACM Press, 9–11 June 1993.
- [13] J.C. Mitchell and G.D Plotkin. Abstract types have existential type. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 37–51, 1985.
- [14] R. Noble and C. Runciman. Functional languages and graphical user interfaces - a review and a case study. Technical report, Department of Computer Science, University of York, England, 3 February 1994.
- [15] R. Noble and C Runciman. Gadgets: Lazy functional components for graphical user interfaces. In M. Hermenegildo and S.D. Swierstra, editors, *Proceedings of Seventh International Symposium on Programming Languages: Implementations, Logics and Programs, Utrecht, The Netherlands*, number 982 in LNCS, pages 321–340. Springer-Verlag, 20–22 September 1995.
- [16] E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J Plasmeijer. Concurrent clean. In E.H.L. Aarts, J. van Leeuwen, and M. Rem, editors, *Proceedings of Parallel Architectures and Languages Europe, June, Eindhoven, The Netherlands*, number 506 in LNCS, pages 202–219. Springer-Verlag, 1991.
- [17] M.J. Plasmeijer and M.C.J.D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Publishing Company, 1993.
- [18] Achten P.M. *Interactive Functional Programs - Models, Methods, and Implementation*. PhD thesis, University of Nijmegen, 1996.
- [19] E Scholz. sc Pidgets - unifying pictures and widgets in a constraint-based framework for concurrent gui programming. In H. Kuchen and S.D. Swierstra, editors, *Proceedings of eighth International Symposium on Programming Languages: Implementations, Logics, and Programs, Aachen, Germany*, number 1140 in LNCS, pages 363–377. Springer-Verlag, September 1996.
- [20] J.E.W. Smetsers, E. Barendsen, M.C.J.D. van Eekelen, and M.J Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In H.J. Schneider and H. Ehrig, editors, *Proceedings Workshop Graph Transformations in Computer Science, Dagstuhl Castle, Germany*, number 776 in LNCS, pages 358–379. Springer-Verlag, 4–8 January 1993.
- [21] P. Wadler. Comprehending monads. In *Proceedings of the ACM Conference on Lisp and Functional Programming, Nice*, pages 61–78. ACM Press, 1990.

Programming in the Large: The Algebraic-Functional Language Opal 2 α

Klaus Didrich, Wolfgang Grieskamp, Christian Maeder, and Peter Pepper

Technische Universität Berlin, Fachbereich 13 - Informatik,
`{kd,wg,maeder,pepper}@cs.tu-berlin.de`

Abstract. We present design aspects of the algebraic-functional language OPAL 2 α , which features flexible modularization to support programming in the large. OPAL 2 α provides parameterized *structures* and *theories* as part of *packages*. Parameterized theories serve to specify properties of functions and are shown — along with *partial instantiations* — to play a similar role to *type classes* in HASKELL. *Structures* can be viewed as (mere) interfaces or as concrete implementations. A *realization* relation between structures tells us which implementation realizes which interface. A *package* is a kind of folder for structures and theories. Packages establish flexible name spaces via an *import* relation. Name spaces, overloading and instance derivation are based on a well-established *annotation* concept.

1 Introduction

“Programming in the large”, that is, the production of medium- to large-sized software systems, is generally considered to be the true challenge in current programming research. In contrast to the area of “programming in the small”, it appears that the solutions are expected to be found less in programming language concepts than in methodological approaches and tools — the ubiquitous CASE tools — based solely on software engineering principles.

Yet, as once pointed out by David Gries, the principles used in programming in the large will not differ significantly from those we so successfully employ in programming in the small. (This claim is quite plausible because all programming paradigms, be they “small” or “large”, are ultimately based on fundamental mathematical constructions — of which there are but few.)

When we look at modern software methodology, Gries’ conjecture is clearly confirmed. One does indeed need such principles as clustering of related items, hierarchical structuring, parameterization and substitution, encapsulated name spaces, refinement relations and inheritance. But of these only some can be provided by CASE tools — the others are intrinsically interwoven with the underlying programming language and thus must be realized as part of the language.

In this paper we present and discuss aspects of the algebraic-functional programming language OPAL 2 α , which is an experimental successor of the language

OPAL 1. OPAL is a strongly typed, higher-order, pure and strict applicative language, which has been used since around 1989 in teaching and research at the TU Berlin and elsewhere. The current state of OPAL 1 is described in [3] and a discussion of the language is given in [2]. The experiences gained with OPAL 1 over the years have led us to a number of conclusions about the appropriateness of certain language design decisions. In particular, the modularization features of OPAL 1 are relatively inflexible. Thus, the main focus of the redesign in OPAL 2 α concerns constructs for *programming in the large*.

For flexible modularization OPAL 2 α uses *structures*, *theories* and *packages*. Structures and theories — collectively referred to as *units* — may be parameterized by types and functions and are best compared to the specifications in Extended ML [11], ASL [18] or to the objects and theories in OBJ [6]. Parameterized units may be viewed in the following ways: fully instantiated, partially instantiated or uninstantiated (supplying polymorphic names). Structures, theories and packages may be related in different ways, not just via the ordinary import relation.

Section 2 presents an overview of the modularization features of the language; this is mainly done by way of simple illustrating examples. In Section 3 we consider the means by which program elements can be named, because this is an issue of high and traditionally underestimated importance.

2 Modularization Concepts

Our experiences with OPAL 1 taught us that modularization should be supported on two levels of granularity. We call these *packages* and *units*: packages are clusters of units, while units are either *structures* or *theories*, of which the former are *program* components and the latter *specification* components. Both structures and theories constitute modules in the classical sense, that is, they consist of type and function declarations, definitions and axioms.

2.1 Packages

A real-world software project nowadays comprises a large number of individual software components, usually called “modules” in software engineering. For example, the compiler for the OPAL 1 language consists of about 250 modules, which introduce around 180 types and 6500 functions. The OPAL library (which is also utilized by the compiler) comprises around 300 modules with 120 exported types and 2700 exported functions.

In order to manage projects of such size one needs structuring mechanisms that go beyond mere collections of modules. In OPAL 2 α , we have decided to introduce the concept of *packages* for this purpose. (They are similar to the — independently developed — idea of packages in JAVA.) A *package* basically is a collection of *structures* (which are the “modules” of OPAL), but it offers a number of additional features and capabilities (see Program 1):

- Packages are fully integrated into the language. Therefore they provide — by contrast to a mere CASE-tool-based or file-based management of modules — an additional level of name spaces with facilities for information-hiding and overload resolution. (This will be discussed in more detail in Section 3.)
- The usage of parameterized structures — which are usually too fine-grained to count as true software-modules — is made more convenient, because less notational overhead is needed.
- Structures within a package can exhibit *cyclic dependencies*. For example, the abstract syntax of OPAL itself entails types for expressions and for names, which mutually depend on each other. These types can now be defined in two different structures, provided they are within one package.
- A package may contain *global* imports, which is equivalent to repeating the import in each individual structure of the package.
- Also, a package can be imported as a whole, which is equivalent to importing all structures of the package.
- Structures within one package share the same name space. It is not necessary to supply imports for objects originating from a structure in the same package, thus saving a lot of trivial imports (that in practice tend to clutter the program texts).

We demand that packages must be compilable incrementally. Therefore dependencies between packages must be acyclic (in contrast to the packages in JAVA).

Example: Program 1 shows the package **Sets**. (The various syntactic features will be explained in the following sections.)

Program 1 Package for sets

PACKAGE Sets	-- <i>sets package</i>
IMPORT Int ONLY ...	-- <i>global import</i>
STRUCTURE Set $[\alpha, <]$	-- <i>parameterized structure</i>
ASSUME TotalOrder $[\alpha, <]$	-- <i>parameter spec.</i>
TYPE Set	-- <i>type of sets</i>
...	
PRIVATE STRUCTURE SetImpl $[\alpha]$	-- <i>implementation of sets</i>
REALIZES Set $[\alpha]$	
TYPE Set == Seq $[\alpha]$	-- <i>hidden representation</i>
...	
STRUCTURE SetMap $[\alpha, \beta, <_\alpha, <_\beta]$	-- <i>higher-order functions on sets</i>
FUN _ * _ : $(\alpha \rightarrow \beta) \rightarrow \mathbf{Set}[\alpha] \rightarrow \mathbf{Set}[\beta]$	-- <i>map function</i>
...	

This package collects all structures that are related to sets. The basic structure **Set** introduces the type of sets together with its elementary operations. It is parameterized by an element type α , which has to carry a total order. This structure is implemented (“realized”) by the structure **SetImpl**, which is hidden inside the package. In order to provide the usual higher-order polymorphic functions on sets another structure, **SetMap**, is introduced.

2.2 Structures

The main units of an OPAL package are so-called *structures*. They correspond to what other languages call modules, clusters, classes and the like. A simple example is given in Program 2. This (unparameterized) structure declares the type of the integers together with some basic functions. In a function declaration, a notation like ‘ $_ + _$ ’ introduces an infix operator. Note that the name `Int` is used both for the structure and for the type defined by the structure. In addition to this (harmless) overloading, there is another form of overloading, namely subtraction and negation.

The structure `Int` also *asserts* the theory `TotalOrder`, which will be discussed below.

Program 2 Structure of the integers

```

STRUCTURE Int                -- the integers
  TYPE Int                  -- the underlying carrier set
  FUN 0: Int                -- the constant zero
  FUN _ + _ : Int  $\times$  Int  $\rightarrow$  Int -- addition (infix)
  FUN _ - _ : Int  $\times$  Int  $\rightarrow$  Int -- subtraction (infix)
  FUN - : Int  $\rightarrow$  Int        -- negation (prefix)
  ...
  FUN _ < _ : Int  $\times$  Int  $\rightarrow$  Bool -- ordering (infix)
  ASSERT TotalOrder[Int, <]    -- assertion of theory
  ...

```

Structures can *import* other structures either completely or selectively (see Program 1 or Program 4).

```

IMPORT Int COMPLETELY
IMPORT Int ONLY Int 0 =

```

In the first case all types and functions declared in the structure `Int` are made available, whereas in the second case only the type `Int`, the constant `0` and the equality test can be used.

2.3 Theories

In OPAL 2 α we express specifications by way of *theories*, which are similar in intent, but more flexible and expressive than type class declarations in HASKELL [15] or signatures in ML [8]. Theories are syntactically similar to parameterized structures, but semantically different. The relationship between structures and theories will be discussed in Section 2.4.

For example, the theory of total orderings is given in Program 3. This theory not only defines the ordering axioms on its parameters $[\alpha, <]$ but also introduces the function ‘ $=$ ’, which can be derived from a total ordering.

Specification facilities like these theories are useful not only in the context of program specification and verification, but also in connection with executable software components:

Program 3 Theory of total orders

```

THEORY TotalOrder[ $\alpha$ , <]
  TYPE  $\alpha$                                 -- parameter
  FUN  $_ < _ : \alpha \times \alpha \rightarrow \text{Bool}$  -- parameter

  AXM ALL  $x1$  .  $\neg (x1 < x1)$                 -- irreflexivity
  AXM ALL  $x1\ x2\ x3$  .
     $(x1 < x2) \wedge (x2 < x3) \implies (x1 < x3)$  -- transitivity
  AXM ALL  $x1\ x2$  .  $(x1 < x2) \vee (x2 < x1) \vee (x1 = x2)$  -- trichotomy

  FUN  $_ = _ : \alpha \times \alpha \rightarrow \text{Bool}$  -- additional function
  AXM ALL  $x1\ x2$  .
     $(x1 = x2) \equiv \neg (x1 < x2) \wedge \neg (x2 < x1)$ 
  ASSERT Equality[ $\alpha$ , =]

```

- Theories serve to characterize the parameters of generic structures rigorously and compactly. (*Assumptions* will be discussed in Section 2.4.)
- Theories enhance capabilities for automatic inference of instantiations of parameterized structures. (Assumptions must be matched by assertions.)
- Theories are a shorthand notation for expressing signatures and properties and thus provide a concise and readable specification and documentation facility. (*Assertions* correspond to HASKELL’s instance declarations.)

On the language level, one application of theories is to *assert* that the pertinent properties hold for some given types and functions. This is expressed by an ASSERT statement, as is illustrated in Program 2. Note that assertions can also be used within theories, as can be seen in Program 3. Another application is to *assume* theories for parameters, an application which will be discussed in the next section.

We can also introduce derived operations in theories, such as ‘=’ in **TotalOrder**, which are then made automatically available wherever the theory is asserted. For instance, the assertion in Program 2 also introduces an equality on the integers. Of course, the compiler does not check whether assertions actually hold, because this would entail theorem-proving facilities. However, we envisage language extensions and a verification tool for proving such claims, as is described in greater detail in [4].

The relationship between structures and theories will be discussed at the end of the next section.

2.4 Parameterization

Flexible parameterization mechanisms must be available on all levels of programming, not only for programming in the small, because they are our most powerful abstraction mechanism.

Units in OPAL can be parameterized by types and functions to support universal polymorphism in the tradition of algebraic specification languages, such as, for

example, ASL [18]. OBJ [6] even claims that this parameterization compensates the lack of higher-order functions. It is in contrast to the classical functional style of polymorphism known from ML [8] or HASKELL [15] that parameterizes only by *type* variables.

Program 4 sketches the structure of sets over elements from a type with a total ordering. (Note that such an ordering is mandatory for implementing operations like “pick an element” or “test equality of sets” constructively *and* efficiently.)

Program 4 Parameterized structure for sets

```

STRUCTURE Set[ $\alpha$ , <]
  ASSUME TotalOrder[ $\alpha$ , <]                -- parameter spec.
  IMPORT Int ONLY Int                      -- import integer type
  TYPE Set                                -- type of sets
  FUN {} : Set                            -- empty set
  FUN _ + _ : Set  $\times$  Set  $\rightarrow$  Set      -- union
  FUN card : Set  $\rightarrow$  Int                  -- cardinality
  ...
  FUN _  $\triangleright$  _ : ( $\alpha \rightarrow$  Bool)  $\times$  Set  $\rightarrow$  Set -- filter
  FUN _ < _ : Set  $\times$  Set  $\rightarrow$  Bool         -- lexicographical order
  ASSERT TotalOrder[Set, <]

```

If we instantiate and import such a structure, e.g. by writing:

```
IMPORT Set[Int, <] ONLY ...
```

then the compiler actually checks whether the arguments [Int, <] indeed constitute a total order. That is, the ASSUME statement in the definition of the parameterized structure has to match a corresponding ASSERT statement for the actual arguments.

For convenience, it is possible in OPAL 2 α to omit the last argument and simply write:

```
IMPORT Set[Int] ONLY ...
```

Such a *partial* instantiation is completed automatically by searching for a matching assertion. (To avoid ambiguities, unique default assertions may be designated.)

In practice, however, the above import requires too much notational overhead when a plethora of differently instantiated versions of a parameterized structure like Set are needed.

For a truly polymorphic usage of functions like *map* (see Program 1), OPAL supports *uninstantiated* imports. (The usefulness of uninstantiated imports was already apparent in OPAL 1.) Suppose that we want to convert a set of real numbers into a set of integers by rounding each individual element. In order to do this we can simply write (note also the overloading of *round*):

```

IMPORT Real    ONLY Real Int round...    -- unparameterized
      Set      ONLY Set                  -- uninstantiated sets
      SetMap   ONLY *                    -- uninstantiated map
FUN round: Set[Real] → Set[Int]
DEF round(A) == round * A

```

Note that partially instantiated and uninstantiated imports only increase the work for the compiler, but they do *not* change the underlying concept: the compiler collects all applications of the imported functions and types and determines which instantiations are needed in order to make them correct. In the above example the compiler thus generates

```

IMPORT Real                                ONLY Real Int round...
      Set[Real, <Real]                    ONLY Set
      Set[Int, <Int]                      ONLY Set
      SetMap[Real, Int, <Real, <Int] ONLY *
FUN round: Set[Real] → Set[Int]
DEF round(A) == round * A

```

Discussion: Even though it is beyond the capabilities of a compiler to actually prove that the given definitions of a structure fulfil the claimed axioms and assertions, we have gained additional programming safety: the required pairing of assumed and asserted theories has similar impacts on the proper use of modules as typing has on the proper use of functions. (The same observation applies to ML functors, but our parameterized structures and theories go further by also including specification features; moreover, they provide implicit renaming morphisms.)

One might consider the amalgamation of the concepts of (parameterized) structures and theories into one concept. Even though such a reduction of different language features appears interesting from a theoretical point of view, we have decided to keep them apart for three reasons: First, structures and theories serve different methodological purposes and therefore it is probably less confusing if they look syntactically different. Second, structures ultimately have to be constructively implemented, whereas theories collect operations and properties that need not be computable. Third, there are actual semantic differences concerning parameterization (see Section 2.6).

2.5 Variants of Inheritance: Interface and Realization

In the object-oriented world, the notion of “inheritance” is at the focus of an intensive debate; in the algebraic-specification world similar questions are discussed using catchwords like “enrichment” and “extension”; and in the type-theory world, one ponders these problems under the heading “subtyping” and “existential types”. These debates have, among other things, led to the clarification of an important trichotomy:

- There is the concept of *specialization*: B is a *subtype* (*subclass*) of A if B has at least all the properties of A .
- Another related but not identical idea is that of *code reuse*: when writing a *subclass* B , one wants to “inherit” some of the code of a superclass A (perhaps selectively overwriting other parts of the code of A).
- Also related is the idea of providing a single interface for different implementations of semantically similar or equivalent data types.

OPAL 2 α currently supports the second and the third of these aspects of inheritance. Code reuse is indeed possible by importing structures and asserting theories, as has been illustrated in the previous section. *Overwriting* definitions (as it is usually done in object-oriented languages) is *not* allowed in these cases. Interfaces with different implementations are provided by the *realization concept*, which makes the relation between interface and implementation very flexible (in contrast to the design of OPAL 1, where they were in a strict one-to-one correspondence):

- “n:1-correspondence”: A given structure may have many *different interfaces* providing different *views* of the structure.
- “1:n-correspondence”: A given interface may be *implemented in different ways*. Classical examples are **Set** and **Bag**, which may be implemented as unordered sequences, ordered sequences, various kinds of trees, hash tables and so forth.
- Finally, the idea of *refinement* makes it necessary that the realization relation is *transitive*: when A is realized by B , it must be possible to realize B in turn by C .

Note, that interfaces and implementations are both structures. Specifically interfaces are not theories. (In ML a single *signature* — that roughly correspond to our theories — may be implemented by different *structures*.)

2.6 Notes on the Semantics

There is evidently not enough space here to give a detailed account of the semantics of the language. But fortunately the realms of Scott domains and of parameterized algebraic specifications have been elaborated so well in the literature (see e.g. [12, 19, 5]) that we can convey the basic design principles by referring to standard terminology.

Since we are talking about programming, the defined functions and types have to be computable. Hence, we employ the category of Scott domains as the basic model.

For *theories* we have a loose semantics in the classical mathematical sense (like lattices, groups or matroids): Any Σ' -algebra, for which there is a forgetful functor to the signature Σ of the theory such that all axioms are fulfilled, is a model. (We once have baptized this, not too seriously, as “hyper-loose” semantics [9] in

answer to the notion of “ultra-loose” semantics in [1], which turned out to be too constrained.) Renaming morphisms evidently pose no problems, and asserting other theories within theories simply unites the axioms.

Assuming theories for parameters constrains the set of possible models that may occur as arguments.

Structures are slightly more complex and need to be considered on two levels. On the first level, we consider a structure by itself, without realization relations in the environment. Again, we assume a loose semantics of Σ -algebras here, where Σ is the signature of the structure. But now we have additional constraints:

- Definitions not only establish the corresponding equations (like axioms), but in addition require the functions to be the least fixpoints of these equations.
- Imports are protected. That is, a structure is a conservative and faithful extension of its imports.
- A model for a structure must also be a model of each of the *asserted* theories.
- Parameterization of structures is also protected in the following sense. The parameters (including the assumed theories) determine the class of algebras that are admissible as arguments.

Since OPAL 2α does not require a structure to provide a definition for each of its functions (which is why a loose semantics makes sense), we need a second level of semantics, which defines how an executable program is constituted. Ultimately every function and type of a program must be constructively defined. This is, where the realization relation has to be considered:

- If a structure **B** realizes a structure **A**, then the model class of **B** is contained in that of **A**. (Note that **B** inherits **A** completely.)
- By taking the transitive closure over its realized-by relations a structure must be completely defined, that is, there must be at least one constructive definition for each of its functions and types.

3 Naming Concept

Naming disciplines vary considerably across different programming languages, ranging from the extremely rigid (every identifier has to be unique throughout the entire program text) to the extremely loose (any identifier can be arbitrarily overloaded as long as the compiler can figure out its meaning). The designs are mostly motivated by technical considerations about (the efficiency of) the envisaged analysis algorithms in the compiler. OPAL has a highly flexible and liberal naming concept, which is based on the following principles:

1. Software is modularized into *units*, which form separate and independent name spaces.
2. Good names are rare, so it should be possible to *reuse* them freely, also *within* the name space of a given unit.

3. The compiler should utilize *all* available information to resolve overloaded names.
4. The programmer can provide all kinds of annotations to the program in order to facilitate — or even enable — the analysis task of the compiler.

There is an ongoing debate as to what extent unrestricted overloading is sensible and where its abuse starts (because it obfuscates the program). We feel, however, that this is a methodological issue that should be decided individually for each software project rather than being enforced universally by a programming language. (After all, almost every generally accepted language feature can be abused.)

From a technical point of view, the trade-off between language flexibility on the one hand and the efficiency of the analysis algorithms on the other should be resolved in favour of flexibility. This entails, for instance, that overload resolution has to be based on both argument *and* result types, even though a purely argument-based resolution algorithm (like that of JAVA) is much simpler to implement.

A liberal discipline of overloading and polymorphism may lead to NP-hard or even undecidable problems. However, the pertinent situations usually only occur in rare and quite pathological programs, so it would be an over-reaction to universally prohibit flexibility just because of problems with a few borderline cases. We prefer to conceive of a mechanism whereby the *programmer* can resolve such pathological cases individually. This way the compiler can be equipped with a “too hard for me” reaction, which is triggered whenever an analysis gets too long.

3.1 Names

Names designate entities like packages, structures, theories, functions and types. Since we are only concerned with overall name spaces, we ignore names for axioms and also local names, that is, bound variables occurring within definitions and formulae.

The principles presented above and the language design outlined in Section 2 necessitate a *name concept* that goes beyond simple qualification. Overloading of identifiers in OPAL 2 α is almost unrestricted; the only constraints are:

1. All package identifiers within a project context must be different.
2. All unit identifiers within a package must be different.
3. All type identifiers within a unit must be different.
4. All functions within a unit having the same identifier must have different functionalities *for all possible instances of that unit*.

The last restriction rules out pathological situations such as

```
STRUCTURE Foo[ $\alpha, \beta$ ]
  FUN foo:  $\alpha \rightarrow \beta$ 
  FUN foo:  $\beta \rightarrow \alpha$ 
```

which would have to identify the two `foo`'s for `Foo[Int, Int]` and distinguish them for `Foo[Int, Real]`. (Such a situation can be easily detected by *unification*.)

The principal idea of overload resolution is that a *name* consists of several components, of which the *identifier* is only one part (though the most important). Such *full names* are not only values managed internally by the compiler, but can also be written down explicitly by the programmer. However, full names are nearly always incomprehensible. Thus, the language allows the programmer to also write *partial names*. The definition of these partial names is simple: any component of the full name, except for the mandatory identifier, can be omitted. An identifier may be *annotated* by its *origin*, that is, a unit or package name, by its *instantiation* (in the case of parameterized units) and by its *kind*. The following examples demonstrate different possibilities for *annotating* the function `card` over integer sets.

```
card           -- no annotations
card`Set       -- annotated by structure
card`Set`Sets  -- annotated by structure and package
card`Set[Int, <] -- annotated by structure and instantiation
card[Int]      -- annotated by partial instantiation
card: Set → Int -- annotated by functionality
card[Int]: Set → Int -- ann. by partial instantiation and functionality
```

In practice we found that simply writing down the identifier is sufficient in “99% of all cases”. The compiler can deduce the rest of the name from the context. In the few remaining situations it is usually sufficient to add just one further component to remove the ambiguity.

The almost fully annotated name of our function `card` — the parameter instances are still only partially annotated — looks as follows. (This clearly demonstrates why nobody would actually want to write full names into program texts.)

```
card`Set`Sets[Int: TYPE, <: Int × Int → Bool]
: Set`Set`Sets[Int: TYPE, <: Int × Int → Bool]
→ Int`Int: TYPE
```

Full names can be internally represented, as illustrated in Table 1 for the aforementioned function `card`. (This table already anticipates some aspects of the algorithm outlined in Section 3.2.) The function `card` and its argument `Set` both come from the instantiated structure `Set`. All three names are instantiated with the type `Int`. The structure `Set` — and all its instances — is a part of the package `Sets`.

The following important properties hold:

1. Full names uniquely identify semantic *entities* — specifically the types and functions — of the language. Any two given full names are equal *if and only if* their identifier, origin, instance and kind are identical.

	<i>Identifier</i>	<i>Origin</i>	<i>Instance</i>	<i>Kind</i>	<i>Comment</i>
$\delta_1 =$	card	δ_3	$[\delta_5, \delta_4]$	$\delta_2 \rightarrow \delta_5$	card : Set \rightarrow Int
$\delta_2 =$	Set	δ_3	$[\delta_5, \delta_4]$	TYPE	Set 'Set' Sets [Int]
$\delta_3 =$	Set	δ_8	$[\delta_5, \delta_4]$	STRUCTURE	Set 'Set' [Int]
$\delta_4 =$	<	δ_6		$\delta_5 \times \delta_5 \rightarrow \delta_7$	Int \times Int \rightarrow Bool
$\delta_5 =$	Int	δ_6		TYPE	Int 'Int'
$\delta_6 =$	Int			STRUCTURE	Int
$\delta_7 =$	Bool			TYPE	predefined Bool
$\delta_8 =$	Sets			PACKAGE	Sets

Table 1. Table of a full name

- Full names are strictly hierarchical, that is, no cyclic names exist. (The rows of Table 1 have been ordered top-down to reflect this hierarchy.)

3.2 Name Resolution at the Module Level

Name resolution is required on two levels in OPAL 2 α :

- The first level concerns the names of packages, structures, theories, types and functions that occur in imports, TYPE- and FUN-declarations and the like. The outcome of this analysis constitutes the *global name space* (of a unit).
- The second level concerns the recognition of names used in expressions. That is, every expression must be formulated using the names from the global name space of the unit under consideration (plus locally defined names).

The latter activity, that is, name resolution for expressions, is an adaptation of the usual Hindley-Milner algorithm for polymorphic type analysis which supports ad-hoc polymorphism (overloading). Note that this task may lead to complex problems (see [7, 13]).

The more interesting aspect for us is name resolution at the module level. Program 5 illustrates overloading and some intricate interdependencies between the various components of a structure.

- The declaration of **match** (line 6) is based on the imported type of **Set** (line 2); the instantiation of this import is in turn based on types and functions declared by the current structure (lines 4 and 5).
- The function '**<**' is imported from **String** (line 3) and also newly declared for **Name** (line 5).
- The identifier **Name** is used to denote a structure, a type and a function (lines 1, 4 and 7, respectively)

Because OPAL does not fix the order of (dependent) declarations and imports, incremental treatment is not possible. Instead we employ a global-search algorithm with constraint propagation in the style of [10, 14], where constraint calculation is performed by a fixpoint iteration.

Program 5 Intricate dependencies of instantiations

STRUCTURE Name		1
IMPORT Set [Name , <] ONLY Set	-- <i>apply new type Name</i>	2
IMPORT String ONLY String <	-- <i>note the overloading</i>	3
TYPE Name	-- <i>new type</i>	4
FUN _ < _ : Name × Name → Bool		5
FUN match : String × Set [Name] → Set [Name]		6
FUN Name : String → Name	-- <i>only for illustration</i>	7
ASSERT TotalOrder [Name , <]		8

The basic principle is that the partial names occurring in a structure are distinguished as being either *defining* or *applied* occurrences. For example, line 6

FUN **match** : **String** × **Set**[**Name**] → **Set**[**Name**]

in Program 5 induces one name definition, namely that of **match**, and five name applications, namely those of **String**, **Set**, **Name**, **Set** and **Name**. (Note that there is no a-priori guarantee that the two occurrences of the identifiers **Set** and **Name** actually denote the same name. Names in program texts are usually *partial*, as described in Section 3.1)

The name space \mathcal{N} of the unit under consideration is partitioned into two disjoint sets, the defined names \mathcal{D} and the applied names \mathcal{A} :

$$\mathcal{N} = \mathcal{D} \cup \mathcal{A} \quad \text{with } \mathcal{D} \cap \mathcal{A} = \emptyset$$

Our name resolution can then be simply reformulated as the global-search problem:

“Find all *total mappings* $\mathcal{A} \rightarrow \mathcal{D}$ of applied names to defined names which are *consistent*.”

If there is exactly one such mapping, then we have a valid name space \mathcal{N} . Otherwise our name space is *inconsistent* (no consistent mapping) or *ambiguous* (more than one consistent mapping).

In order to solve this search efficiently, a fixpoint iteration is used to compute the propagation of *constraints* (see [10]). We cannot present the algorithm here in detail, but we will at least illustrate it using Program 5:

1. We can deduce initial sets \mathcal{D}_0 and \mathcal{A}_0 from the parse tree (and known units).
2. Then we use our global-search algorithm with constraint propagation to complete the information in these sets and to find for each applied occurrence α_i of a name its corresponding definition δ_j .

The initial sets \mathcal{D}_0 and \mathcal{A}_0 for Program 5 are shown (in tabular form) in Table 2. Question marks ‘?’ indicate missing information (to be derived in subsequent phases of the algorithm), while empty slots indicate that the corresponding attribute is not applicable.

\mathcal{D}_0 :	Identifier	Origin	Instance	Kind	Comment
$\delta_0 =$	Name			STRUCTURE	current structure
$\delta_1 =$	Set		$[\alpha_1, \alpha_2]$	STRUCTURE	instantiated IMPORT
$\delta_2 =$	Set	δ_1	$[\alpha_1, \alpha_2]$	TYPE	imported
$\delta_3 =$	String			STRUCTURE	simple IMPORT
$\delta_4 =$	String	δ_3		TYPE	imported
$\delta_5 =$	<	δ_3		$\delta_4 \times \delta_4 \rightarrow \delta_{10}$	imported
$\delta_6 =$	Name	δ_0		TYPE	TYPE declaration
$\delta_7 =$	<	δ_0		$\alpha_3 \times \alpha_4 \rightarrow \alpha_5$	FUN declaration
$\delta_8 =$	match	δ_0		$\alpha_6 \times \alpha_7 \rightarrow \alpha_8$	FUN declaration
$\delta_9 =$	Name	δ_0		$\alpha_9 \rightarrow \alpha_{10}$	FUN declaration
$\delta_{10} =$	Bool			TYPE	predefined name

\mathcal{A}_0 :	Identifier	Origin	Instance	Kind	Comment
$\alpha_1 =$	Name	?	?	TYPE	actual parameters
$\alpha_2 =$	<	?	?	$\alpha_1 \times \alpha_1 \rightarrow \delta_{10}$	of δ_1
$\alpha_3 =$	Name	?	?	TYPE	functionality
$\alpha_4 =$	Name	?	?	TYPE	of δ_7
$\alpha_5 =$	Bool	?	?	TYPE	
$\alpha_6 =$	String	?	?	TYPE	functionality
$\alpha_7 =$	Set	?	$[\alpha_{11}]$	TYPE	of δ_8
$\alpha_8 =$	Set	?	$[\alpha_{12}]$	TYPE	
$\alpha_9 =$	String	?	?	TYPE	functionality
$\alpha_{10} =$	Name	?	?	TYPE	of δ_9
$\alpha_{11} =$	Name	?	?	?	instantiation of α_7
$\alpha_{12} =$	Name	?	?	?	instantiation of α_8
$\alpha_{13} =$	Name	?	?	TYPE	actual parameters
$\alpha_{14} =$	<	?	?	$\alpha_{13} \times \alpha_{13} \rightarrow \delta_{10}$	of TotalOrder

Table 2. Initial name space of Program 5

The propagation of constraints is based on a predicate **matches**, which tests whether an application and a definition are compatible. This predicate basically checks whether both identifiers are equal and whether origins, instances and kinds are compatible. (The comparison of type terms boils down to *unification*.) For a given application α_i , all possibly matching definitions form the *candidate set* of α_i . In general, we must consider the following situations:

- Empty candidate sets immediately indicate an error.
 - Singleton candidate sets indicate unique resolution and allow the substitution of applications by their corresponding definitions.
- After such a substitution, constraint propagation can take place, thus possibly enabling further unique filterings. An *occurrence check* must guarantee that no definition becomes a part of itself.
- Only candidate sets with two or more remaining elements require actual branching in the course of global search.

(In our almost trivial example, Program 5, the constraint propagation resolves all ambiguities. *Actually, this is quite typical: in our experience, “95% of all practical situations” are resolved after the first constraint propagation.*)

This sketch of the name-resolution algorithm is all we have room for. It should, however, be mentioned that some subtle and intricate problems remain, which we have not addressed here. For example, the introduction of theories into the language leads to a new quality of inference, since the problem of finding an assertion for an assumption is generally undecidable. (The reasoning behind this is similar to that concerning *predicate types* given in [16, 17].)

4 Conclusion

OPAL 2 α is the result of discussions in our group based on the experiences (good as well as bad) we had with OPAL 1. Several extensions of OPAL 1 were introduced during recent years, but further progress was not possible without losing compatibility with the original language. So we decided to define a new language, OPAL 2 α , which is still similar to OPAL 1 in look and feel, but at the same time leaves us scope to introduce new concepts for formal software development.

OPAL 2 α is being implemented on the basis of the existing environment of OPAL 1. In order to achieve results with limited resources, our plan is to gradually build the compiler for OPAL 2 α starting at the front end. The back end will be taken from the existing OPAL 1 compiler, thus enabling us to translate a subset of OPAL 2 α . We expect that the established environment of OPAL 1, as sketched in the introduction, can be re-used by following a similar approach.

Acknowledgment The design of the language OPAL has been an ongoing project to which many people have contributed. The earlier name-resolution algorithm for OPAL 1, from which our algorithm is derived, was devised by Andreas Fett and Michael Jatzeck. Niamh Warde was very helpful in formulating the paper.

References

- [1] Manfred Broy and Martin Wirsing. Ultra-loose algebraic specification. *Bulletin of the EATCS*, 53:117–128, June 1988.
- [2] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In J. Gutknecht, editor, *Programming Languages and System Architectures*, LNCS 782, pages 228–244. Springer, March 1994.
- [3] K. Didrich, C. Gerke, W. Grieskamp, C. Maeder, and P. Pepper. Towards Integrating Algebraic Programming and Functional Programming: the Opal System. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, LNCS 1101, pages 559–562. Springer, July 1996.

- [4] Klaus Didrich. Compiler support for correctness proofs. In *Automated Theorem Proving in Software Engineering (CADE-14 workshop)*, 1997.
- [5] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1990.
- [6] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, October 1993.
- [7] Fritz Henglein. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, December 1991.
- [8] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.
- [9] Peter Pepper. Transforming Algebraic Specifications – Lessons learnt from an Example. In Bernhard Möller, editor, *Constructing Programs From Specifications*, pages 399–425. North-Holland, 1991.
- [10] Peter Pepper and Douglas R. Smith. A High-Level Derivation of Global Search Algorithms (with Constraint Propagation). *Science of Computer Programming*, 28(2–3):247–271, 1997.
- [11] Donald T. Sannella and Andrzej Tarlecki. Extended ML: An Institution-Independent Framework for Formal Program Development. In *Proc. Workshop on Category Theory and Computer Programming*, LNCS 240, pages 364–389. Springer, September 1986.
- [12] David A. Schmidt. *Denotational Semantics. A Methodology for Language Development*. W. C. Brown Publishers, 1988.
- [13] Michael I. Schwartzbach. Polymorphic Type Inference. Lecture Series LS-95-3, BRICS, DAIMI, June 1995.
- [14] Douglas R. Smith and Eduardo A. Parra. Transformational Approach to Transportation Scheduling. In *Proc. 8th Knowledge-Based Software Engineering Conf., Chicago, IL*, pages 60–68, 1993.
- [15] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 1996.
- [16] Dennis M. Volpano and Geoffrey S. Smith. On the Complexity of ML Typability with Overloading. Technical Report TR91-1210, Cornell University, May 1991.
- [17] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages*, pages 60–76. ACM Press, January 1989.
- [18] Martin Wirsing. Structured algebraic specifications: a kernel language. *Theoretical Computer Science*, 42:123–249, 1986.
- [19] Martin Wirsing. Algebraic Specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. North-Holland, 1990.

Context Patterns, Part II

Markus Mohren

Lehrstuhl für Informatik II, RWTH Aachen, Germany
mohren@informatik.rwth-aachen.de

Abstract. Functional languages allow the definition of functions by pattern matching, which performs an analysis of the *structure of values*. However, the structures which can be described by such patterns are restricted to a fixed portion from the root of the value.

Context patterns are a new non-local form of patterns, which allow the matching of subterms without fixed distance from the root of the value. Typical applications of context patterns are functions which *search* a data structure for patterns and *transform* it by replacing the pattern. In this paper we introduce a new construct called *extended context*, which allows the definition of transformational functions without superfluous repetition of the recursive search. Furthermore, we present an inference system which formalises the type conditions for context patterns.

1 Introduction

This paper is the successor to “Context Patterns in Haskell” [13]. That work described a new non-local kind of pattern which allows matching of subterms without fixed distance from the root of the value. The underlying observation is that standard patterns allow only the matching of a fixed region near the root of the structure. Consequently, the resulting bindings are substructures adjacent to the region (see Figure 1(a)). It is neither possible to specify patterns at a non-fixed distance (possibly far) from the root, nor to bind the context of such a pattern to a variable (see Figure 1(b)).

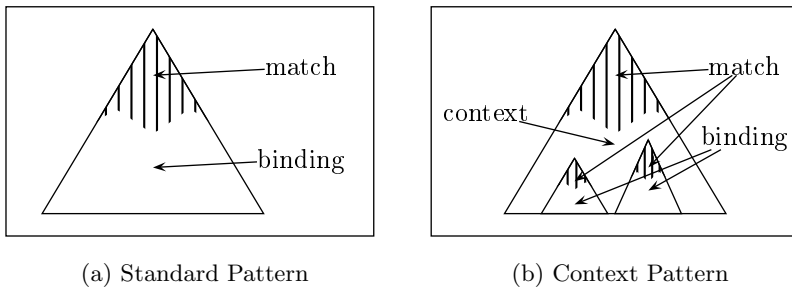


Fig. 1. Pattern Matching

Context patterns are a flexible and elegant extension of traditional patterns. Typical examples of functions using the increased expressive power are functions which search and transform data structures.

Transformations often replace all occurrences of a pattern without introducing new occurrences. Such functions can be programmed manually such that the data structure is traversed only once. However, the use of context patterns in combination with recursion imposed an additional repetition of the recursive search for those parts before the last match.

In this paper, we extend context patterns to overcome this deficiency: We introduce a new construct called *extended context*, which gives access to the unvisited part of a data structure. Recursive calls of a transformation can be placed to visit these parts only without revisiting the parts before the last match.

In addition, we formalise the type conditions for context patterns in terms of an inference system. In the previous paper, the type conditions were only stated informally.

This paper is organised as follows. In Section 2 we introduce context patterns by example programs and discuss the necessary modification to the approach in [13]. Section 3 defines the syntax of context patterns. Moreover, the static semantics of context patterns is defined by an inference system and we discuss the relation of context patterns and the type system of Haskell. The implementation is described by a translation of context patterns to standard Haskell which is presented in Section 4. Some of the previous work in pattern matching and topics related to our approach is reviewed in Section 5. Section 6 concludes.

2 Context Patterns by Examples

To demonstrate the basic idea, we start with a toy example. Consider a function `initlast :: [a] -> ([a], a)` which splits a list into its initial part and its last element. Informally, we can describe an implementation as a single match:

Take everything up to but not including a one-element list as initial part and the element of the list as last element.

However, we cannot express this match by using the standard patterns. Instead, the recursive search must be programmed explicitly:

```
initlast :: [a] -> ([a], a)
initlast []      = error "Empty list"
initlast [x]     = ([], x)
initlast (x:xs) = let (ys, l) = initlast xs in (x:ys, l)
```

Our extension consists of a single additional pattern called *context pattern*, with the syntax:

$$cpat \quad \rightarrow \quad var \, pat_1 \, \dots \, pat_k$$

A context pattern matches a value v if there exists a function f and values v_1, \dots, v_k such that pat_i matches v_i and $f \, v_1 \, \dots \, v_k$ is equal to v . Furthermore,

this function f is a representation of a *constructor context* [1], i.e. a constructor term with “holes”. The representation consists of modelling the “hole” by the function arguments:

$$f = \lambda h_1 \dots \lambda h_k. C[h_1, \dots, h_k]$$

where C is a constructor context with k “holes”, which imitates the shape of the value v . If the pattern matches the value, the function f is bound to the variable var .

In our example, we can reformulate `initlast` using context patterns in the following way:

```
initlast :: [a] -> ([a], a)
initlast []      = error "Empty list"
initlast (c [x]) = ((c []), x)
```

Applying `initlast` to a list $[a_1, \dots, a_{n-1}, a_n]$ gives us the following bindings:

$$[x/a_n, c/\lambda l \rightarrow (a_1 : \dots (a_{n-1} : l) \dots)]$$

Hence, evaluation of the application `(c [])` on the right hand side yields the initial part $[a_1, \dots, a_{n-1}]$.

2.1 The Problem of Replacing All Occurrences

Transformations typically replace *all* occurrences of a pattern. We can easily do this by combining context patterns and (tail-)recursion. For instance, replacing all occurrences of the character `a` in a string can simply be done by the following function

```
rplc_a :: Char -> String -> String
rplc_a x (c 'a') = rplc_a x (c x)
rplc_a x s = s
```

The first rule searches for the first occurrence of the character `'a'` in the string. By the application `(c x)` a new string is created where this occurrence is replaced by the value of `x`. The recursive call of `rplc_a` then replaces all other occurrences until the context patterns fails and the result is returned in the second rule.

Although this technique is applicable in general, it imposes an additional overhead: The portion of the string before the first occurrence of `a` is searched again by all recursive calls to `rplc_a`. However, this is not necessary at all. For this example, this results in quadratic complexity of `rplc_a`, whereas only linear complexity is needed.

To overcome this deficiency, we extend the approach. We assume that we have a purely transformational function, where the result has the same type as the input. To avoid the searching of the already searched part, we simply have to ensure that the recursive call is only applied to the *unvisited subvalues* of

same type in the context. Since the context function does not allow access to these parts, we introduce another variant of the context function: Assume that we have a context pattern $c \text{ pat}_1 \dots \text{pat}_k$. In addition to the context c with type $t_1 \rightarrow \dots \rightarrow t_k \rightarrow t$ an *extended context* c_e of type¹ $[t] \rightarrow t_1 \rightarrow \dots \rightarrow t_k \rightarrow t$ is available on the right hand side. The difference is that c_e accepts a list of subvalues which are used as replacement for the unvisited subvalues. In addition, the list of unvisited subvalues is accessible as list c_l : We have $c = (c_e \ c_l)$. By using this extension, we can reformulate `rplc_a` in the following way:

```
rplc_a :: Char -> String -> String
rplc_a x (c 'a') = c_e (map (rplc_a x) c_l) x
rplc_a x s = s
```

Given the string "snafu", we obtain the following bindings for the context and the extended context:

```
c = λx.'s': 'n': x++"fu"
c_l = ["fu"]
c_e = λ[l].λx.'s': 'n': x++l
```

Hence, the recursive call applies `rplc_a` to the string "fu" only.

2.2 A More Challenging Example

All previous example were functions operating on lists. Of course, they all can be written without effort in the traditional Haskell way either by explicit recursive programs or list comprehensions. We now give a more challenging example, where the advantages of context patterns become clear.

We consider the task of writing an optimising compiler for a functional language. For simplicity, we consider only λ -expression without data structures, patterns, or `case`-constructs. For the internal representation of expressions, we use the following data structure, assuming that identifiers are represented by strings:

```
data Expr = Var String
          | Lit String
          | Lam String Expr
          | App Expr Expr
          | Let [(String,Expr)] Expr
          | If Expr Expr Expr
```

¹ In the original workshop presentation, the extended context had type $(t \rightarrow t) \rightarrow t_1 \rightarrow \dots \rightarrow t_k \rightarrow t$, and c_e applied its functional argument to the unvisited subvalues: $c = (c_e \ id)$. The discussion showed that this was too restrictive.

The optimisation we want to program is simply β -reduction of all applications of λ -abstractions. We assume a function `subst :: String -> Expr -> Expr -> Expr` performing the replacement of all occurrences of the variable in the first expression by the second expression. Hence, we have to search for the pattern `(App (Lam x e1) e2)` and replace it by `subst x e1 e2`. In the traditional way, we program the recursive search explicitly:

```

beta :: Expr -> Expr
beta e = fst (until (not.snd) (beta'.fst) (beta' e))

beta' :: Expr -> (Expr,Bool)
beta' (Var id)    = (Var id, False)
beta' (Lit id)    = (Lit id, False)
beta' (Lam id e)  = let (e',b) = beta' e
                    in (Lam id e', b)
beta' (App (Lam x e1) e2)
        = let (e',_) = beta' (subst x e1 e2)
          in (e', True)
beta' (App e1 e2) = let (e1',b1) = beta' e1
                    (e2',b2) = beta' e2
                    in (App e1' e2', b1||b2)
beta' (Let bnd e) = let (e',b) = beta' e
                    (xs,ebs) = unzip (map betal' bnd)
                    (es,bs)  = unzip ebs
                    bnd'     = zip xs es
                    in (Let bnd' e', or (b:bs))
    where betal' :: (String,Expr) -> (String,(Expr,Bool))
          betal' (x,e) = (x,beta' e)
beta' (If ec et ef) = let (ec',bc) = beta' ec
                      (et',bt) = beta' et
                      (ef',bf) = beta' ef
                      in (If ec' et' ef', bc||bt||bf)

```

In the fourth case, the actual replacement takes place. However, it is hardly recognisable since we have to ensure that the resulting expression does not contain newly created redexes: The function `beta'` returns an additional boolean value indicating whether at least one replacement has taken place. The function `beta` iterates until no more replacements take place.

Apart from this case, the program is just another recursive traversal of the data structure `Expr`. However, using context patterns, the same function can be written much conciser in the following way:

```

beta :: Expr -> Expr
beta (c (App (Lam x e1) e2)) = beta (c (subst x e1 e2))
beta e = e

```

This program is a straightforward implementation of the original idea. In addition, there is no need for an additional boolean indicating whether a replacement has taken place.

However, the price to pay is an unnecessary overhead: After finding one redex, the complete data structure is searched again. For instance, consider the expression

```
If (Var "x") (App (Lam "y" (Var "y")) (Lit "0"))
    (App (Lam "z" (Var "z")) (Lit "1"))
```

The first check of the context pattern succeeds with the binding of the context function $c = \lambda e.(\text{If } (\text{Var } "x") \ e \ (\text{App } (\text{Lam } "z" \ (\text{Var } "z")) \ (\text{Lit } "1")))$. In the recursive call of **beta'** the second check of the context pattern traverses the whole expression again, although it is clear that there cannot be any matches before reaching the position where the last match was. But since there are no further matches there, the context pattern finally succeeds with the binding $c = \lambda e.(\text{If } (\text{Var } "x") \ (\text{Lit } "0") \ e)$. The third check of the context pattern traverses the term again and fails, resulting in the expression $(\text{If } (\text{Var } "x") \ (\text{Lit } "0") \ (\text{Lit } "1"))$. In total, the top-level constructor **If** is visited three times.

Extended contexts help to deal with this situation more efficiently. Consider a version of **beta** written by using the extended context:

```
beta :: Expr -> Expr
beta e = fst (until (not.snd) (beta'.fst) (beta' e))

beta' :: Expr -> (Expr, Bool)
beta' (c (App (Lam x e1) e2))
    = let (es', bs) = unzip (map beta' c_l)
        (e', b)    = beta' (subst x e1 e2)
        in (c_e es' e', or (b:bs))
beta' e = (e, False)
```

The context pattern has moved to the auxiliary function **beta'** which replaces all occurrences of the redex in one traversal of the expression. Analogous to the first version of this program, **beta'** returns a boolean result indicating whether at least a reduction was performed. However, the computation of this value is restricted to the place where the actual replacement takes place. In contrast, the computation in the traditional version is scattered throughout the whole program.

To clarify the extended bindings we consider the slightly more complicated example expression

```
Let [("z1", (App (Lam "u" (Var "u")) (Lit "2"))),
    ("z2", (Lit "3"))]
    (If (Var "x") (App (Lam "y" (Var "y")) (Lit "0"))
        (App (Lam "z" (Var "z")) (Lit "1")))
```

Matching of the context pattern $(c \text{ (App (Lam } x \text{ e1) e2))}$ succeeds and yields the extended bindings $c_e = \lambda[e_1, e_2] e. (\text{Let } [(\text{"z1"}, e), (\text{"z2"}, e_1)] \text{ e}_2)$ and $c_l = [(\text{Lit "3"}), (\text{If ...})]$.

Application of `map beta'` to c_l causes two matchings of the context patterns: The first fails, resulting in $(\text{Lit "3"}, \text{False})$. The second succeeds with the bindings are: $c_e = \lambda[e_1] e. (\text{If (Var "x")} e \text{ e}_1)$ for the context function and $c_l = [(\text{App (Lam "z" (Var "z")) (Lit "1"))]$ for the list of unvisited occurrences of expressions. In turn, this causes one other recursive call of `bind'` with a succeeding context pattern match and the bindings $c_e = \lambda[] e.e$ and $c_l = []$. Combining all intermediate results yields the expression

```
Let [("z1", (Lit "2")), ("z2", (Lit "3"))]
    (If (Var "x") (Lit "0") (Lit "1"))
```

We can identify another place for context patterns in this setting: If we assume that a variable renaming has taken place, which ensures that no inner binding uses the same variable names as an outer one, then we can program the function `subst :: String -> Expr -> Expr -> Expr` also using context patterns:

```
subst :: String -> Expr -> Expr -> Expr
subst x (c (Var y) if x==y) e = c e
```

For instance, the Glasgow Haskell Compiler performs such a renaming phase, since it helps to simplify all subsequent phases.

2.3 Additional Features

In addition to the basic syntax of context patterns there are three extras features which are discussed in more detail in [13]:

- *Context wildcards* $(_)$ can be used to match contexts which are not used on the right hand side. For instance, to obtain the last element of a list, we can define:

```
last :: [a] -> a
last []      = error "Empty list"
last (_ [x]) = x
```

- *Guards* introduce additional context-sensitive (or non-free) conditions during the recursive search. An example which uses this feature is an implementation of `takeWhile`

```
takeWhile :: (a->Bool) -> [a] -> [a]
takeWhile p (c (x:_ ) if (not (p x))) = c []
takeWhile p l = l
```


The first rule matches a context where the head of the remaining list is the first element which does not satisfy p . By replacing the remaining list by the empty list in the context, we obtain the longest prefix of elements satisfying p . The second rule is to handle those lists where all elements belong to this prefix.

- *Explicit types* at the patterns can be used to avoid underspecified context patterns.

3 Syntax and Static Semantics

Since context patterns do not interfere with the class system including constructor classes, we present context patterns in the context of **Haskell** 1.2.

The syntax of **Haskell**'s patterns is shown in Figure 2(a) (taken from [8, pp. 17–18]). For simplicity, we omit as-patterns, irrefutable patterns, infix patterns, tuple patterns, unit patterns, and $n + k$ patterns.

3.1 Syntax of Context Patterns

Our extension is in Figure 2(b). There is one small conflict which arises with this extension: The definition

$$\text{let } x \ (y:ys) = e_1 \text{ in } e_2$$

can be either a function definition for x using the pattern $y:ys$, or a context pattern. In these cases, function definitions are preferred.

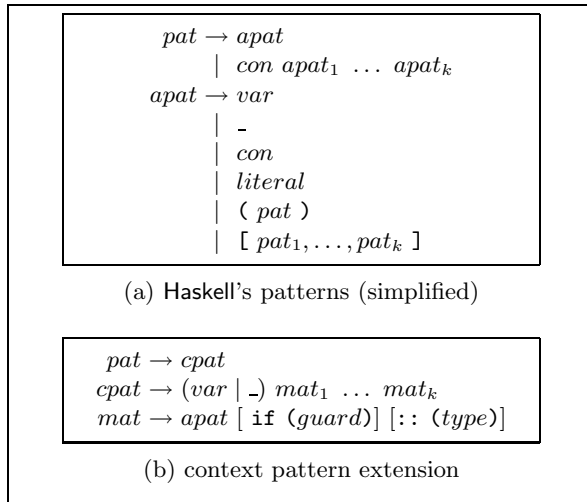


Fig. 2. Extended Haskell patterns

3.2 Pattern Type Inference

Type inference of patterns is done by the inference system in Figure 3, which yields sentences of the form

$$A \vdash pat \Rightarrow (A', \tau)$$

where A is an environment for variables and constructors, pat is a pattern, A' is the environment for the variables bound by pat , and τ is the type of pat . The combination of environments $A \cup A'$ is defined iff A and A' are identical on shared entries. For the rule MAT we assume that $guard = \mathbf{True}$ if no guard is present and similarly $type = \mathbf{a}$. To obtain a simpler system, the rules APPL, CP1, and CP2 could be merged into one.

Axioms:	
$\frac{}{A \vdash var \Rightarrow ([var/\tau], \tau)}$	VAR
$\frac{}{A \vdash _ \Rightarrow ([], \tau)}$	WILD
Standard Rules:	
$\frac{A(con) \succ \tau}{A \vdash con \Rightarrow ([], \tau)}$	CON
$\frac{A(literal) \succ \tau}{A \vdash literal \Rightarrow ([], \tau)}$	LIT
$\frac{A \vdash pat_i \Rightarrow (A_i, \tau) \ (1 \leq i \leq n)}{A \vdash [pat_1, \dots, pat_k] \Rightarrow (A_1 \cup \dots \cup A_n, [\tau])}$	LIST
$\frac{A(con) \succ \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \quad A \vdash apat_i \Rightarrow (A_i, \tau_i) \ (1 \leq i \leq n)}{A \vdash con \ apat_1 \ \dots \ apat_k \Rightarrow (A_1 \cup \dots \cup A_n, \tau)}$	APPL
Context Pattern Rules:	
$\frac{A(var) \succ \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \quad A \vdash mat_i \Rightarrow (A_i, \tau_i) \ (1 \leq i \leq n)}{A \vdash var \ mat_1 \ \dots \ mat_k \Rightarrow (A_1 \cup \dots \cup A_n, \tau)}$	CP1
$\frac{A \vdash mat_i \Rightarrow (A_i, \tau_i) \ (1 \leq i \leq n)}{A \vdash _ \ mat_1 \ \dots \ mat_k \Rightarrow (A_1 \cup \dots \cup A_n, \tau)}$	CP2
$\frac{A \vdash apat \Rightarrow (A', \tau) \quad A \vdash guard \Rightarrow \mathbf{Bool} \quad \tau \succ type}{A \vdash apat \ \mathbf{if} \ guard :: type \Rightarrow (A', type)}$	MAT

Fig. 3. Inference Rules for Patterns

3.3 Non-type Restrictions

Besides type correctness, there are two additional context-sensitive restrictions for Haskell's patterns:

CS1: All patterns must be linear, i.e. no repeated variable occurs in the pattern.

CS2: The arity of a constructor must match the number of sub-patterns associated with it, i.e. no partially applied constructors occur in the pattern.

However, not all context patterns which fulfill these conditions are to be considered well-formed: If there is no value which can possibly match the context pattern, then we consider this context pattern to be malformed. Consider the following (malformed) function definition

```
foo :: [a] -> [a]
foo (c (x:xs) (y:ys)) = exp
```

Here, we try to find two non-overlapping lists matching $(x:xs)$ and $(y:ys)$ with a list. However, the matching can never succeed: Obviously, a list cannot contain two non-overlapping sublists.

To avoid such malformed context pattern, we require that a context pattern $var\ mat_1 \dots mat_k$ satisfies the following context-sensitive condition:

CS3: If the context variable var has type $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$, then there must exist a value v of type τ and values v_i of type τ_i such that all v_i are independent subexpressions of v and occur in the sequence v_1, \dots, v_n in a top-down, left-to-right traversal of v .

To check this condition, we have to test if it is possible to construct such a value v . This checking is done by the inference system in Figure 4. For a type expression τ the sentences have the form

$$A \vdash \tau \rightarrow (\tau_1, \dots, \tau_k)$$

where A is an environment for (data) constructors and (τ_1, \dots, τ_k) is a list of types. The τ_i are types of independent subvalues of a value of type τ and occur in a top-down, left-to-right traversal. If we can derive the sentence $A \vdash \tau \rightarrow (\tau_1, \dots, \tau_n, \tau_{n+1}, \dots, \tau_{n+l})$ for a context variable of type $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$ then this is equivalent to the admissibility of the context pattern according to condition CS3.

$\frac{}{A \vdash \tau \rightarrow (\tau)} \text{ REF}$ $\frac{A(con) = \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau}{A \vdash \tau \rightarrow (\tau_1, \dots, \tau_n)} \text{ TYCON}$ $\frac{A \vdash \tau \rightarrow (\tau_1, \dots, \tau_n) \quad A \vdash \tau_i \rightarrow (\tau_{i,1}, \dots, \tau_{i,m_i}) \ (1 \leq i \leq n)}{A \vdash \tau \rightarrow (\tau_{1,1}, \dots, \tau_{1,m_1}, \dots, \tau_{n,1}, \dots, \tau_{n,m_n})} \text{ COMB}$
--

Fig. 4. Inference Rules for Condition CS3

Type variables and function types are solely handled by the axiom REF. Type variables are not expanded which means that no bindings can occur. Functional types are not decomposed, since no search can go into the components either. For an algebraic types with k constructors, we have $k + 1$ possibilities: use rule

TYCON for each constructor and use rule REF. Finally, rule COMB is to combine the results of adjacent components of constructors. An implementation of the inference system is described in [14].

Example: We use the context pattern $(c \ x \ y)$ with type $[a]$ and the environment $A = [\text{Cons}/a \rightarrow [a] \rightarrow [a], \text{Nil}/[a]]$.

$$\frac{\frac{A(\text{Cons})=a \rightarrow [a] \rightarrow [a]}{A \vdash [a] \rightarrow (a, [a])} \quad \frac{}{A \vdash a \rightarrow (a)} \quad \frac{A(\text{Cons}) = a \rightarrow [a] \rightarrow [a]}{A \vdash [a] \rightarrow (a, [a])}}{A \vdash [a] \rightarrow (a, a, [a])}$$

Hence, the variable x must have type a . Possible types for the variable y are $[a]$ or a .

3.4 Context Patterns vs. Type Inference

In contrast to the description in [13], the check of condition *CS3* does *not* introduce new bindings of type variables. Consequently, the type condition of context patterns is only checked and, besides the normal type inference, no extra inference is performed. Since the construction of a value according to *CS3* is non-deterministic, the introduction of new bindings would also be non-deterministic: Potential type errors caused by the choice of bindings are completely inexplicable to the programmer.

Instead, we consider context-patterns where additional choices for type variables would have to be made as *underspecified*. In such cases additional type information must be provided by the programmer, using a type signature for the function or explicit types at the patterns. At first sight, this seems to be a severe restricting: Consider the function definition

$$f \ (c \ [x]) = x$$

Without further information, this context pattern is underspecified, since the type of f is $a \rightarrow b$ and the sentence $A \vdash a \rightarrow ([b])$ does not hold in the inference system from Figure 4. However, one might argue that this function can be applied to objects of different types:

- lists of elements
- lists of lists of elements
- tree of lists of elements
- ...

But in contrast, it is obvious that the application of f to objects of the types

- elements
- trees of elements
- tuples of trees of elements

is nonsense since the match will never succeed. Hence, the type of \mathbf{f} must be restricted in a way such that \mathbf{f} can only applied to objects which may contain a list: $\mathbf{f} :: \mathbf{a} \text{ contains } \mathbf{b} \Rightarrow \mathbf{a} \rightarrow \mathbf{b}$. However, this is a new kind of polymorphism which cannot be expressed in the current type system.

Moreover, this kind of polymorphism would be misleading, since it seems to be a variation of parametric polymorphism: A single function which can be applied to objects of different type. However, although the definition of \mathbf{f} does not change with the types, the semantics does:

- For a lists of elements search for the last element.
- For a lists of lists of elements search for the last element of the first list of elements.
- For a tree of lists of elements search for the last element of the leftmost list of elements.
- ...

Hence, it is clearly a variation of ad-hoc polymorphism. Consequently, a compiler must provide different translations of this single definition, according to all possible types. In fact, this is not possible since at all since there a infinitely many possible types containing a list.

4 Translation into Haskell

In this section we define the semantics of context patterns in terms of a translation into a Haskell program without context patterns. The idea is to implement context patterns by functions that perform the recursive search. The actual matching is done by applications of these functions.

All pattern matching constructs may appear in several places: lambda abstractions, function definitions, pattern bindings, list comprehensions, and **case** expressions. However, the first four of these can be translated into **case** expressions, so we only consider patterns in **case** expressions. Furthermore, we follow [8] and assume that we have a context pattern in a *simple case expression*:

$$\text{case } e_0 \{ c \ p_1 \text{ if } g_1 \ \dots \ p_n \text{ if } g_n \rightarrow e \ ; \ - \rightarrow e' \}$$

For the top-down left-to-right traversal of a value e_0 , we have to visit every sub-expression of e_0 which may contain one of the patterns p_i . Assume that t_0, \dots, t_m are the types of all sub-expressions of e_0 such that t_0 is the type of e_0 and t_{p_i} is the type of the pattern p_i ($1 \leq i \leq n$). To traverse all of e_0 , we provide functions

$$chk_{t_j} :: t_{args} \rightarrow t_j \rightarrow t_{res-j}$$

for each t_j . Each of these functions traverses one type of sub-expression. At top level, we use chk_{t_0} to perform the complete search. Hence, a first approximation of the top-level structure of the translation looks like this:

$$\begin{aligned} & \text{case } e_0 \text{ of } \{c \ p_1 \text{ if } g_1 \dots p_n \text{ if } g_n \rightarrow e ; - \rightarrow e'\} \\ & \rightsquigarrow \text{let } \{ \langle \text{chk}_{t_0}\text{-decl} \rangle ; \dots ; \langle \text{chk}_{t_m}\text{-decl} \rangle \} \\ & \quad \text{in case } (\text{chk}_{t_0} \langle \text{args} \rangle e_0) \text{ of } \{ \dots \} \end{aligned}$$

Example: In this section, we use the following program as running example. It implements a function `rplc`, which is a generalisation of the function `rplc_a` from Section 2. Here, the application `(rplc x y l)` yields a list where all occurrences of `x` in `l` are replaced by `y`.

```
rplc :: Eq a => a -> a -> [a] -> [a]
rplc x y (c z if (z==x)) = c_e (map (rplc x y) c_l) y
rplc x y l = l
```

For this example, we have only one pattern $p_1 = z$ and hence $t_0 = [a]$ and $t_1 = a$. Therefore, the top-level structure of the translation is

```
rplc x y l = let chk0 ... x0 = ...
               chk1 ... x0 = ...
               in case (chk0 ... l) of ...
```

To fill out the dots in this scheme we have to consider which additional arguments t_{args} and which result t_{res-j} are to be provided:

- (a) The functions chk_{t_j} must have information on which pattern p_i is to be searched next.
- (b) The result must contain information on whether the match was successful.
- (c) The result must contain bindings for the variables in the pattern p_i .
- (d) The result must contain a binding for the context c , the extended context c_e , and the list of unvisited substructures c_l . The binding for c can be obtained by the other ones: $c = (c_e \ c_l)$.

We can solve (a) and (b) by using the number of the pattern which is to be matched next as component of both t_{args} and t_{res-j} : The match was successful iff the number returned by chk_{t_0} on the top-level is $n + 1$.

To solve (c), we provide the complete list of variables occurring in the patterns p_1, \dots, p_n as argument and result for each function chk_{t_j} . Please note in this context that we have a function for *each subtype* of the argument and *not* a function for each pattern. Therefore it is possible that a function chk_{t_j} handles more than one pattern.

Condition (d) requires a slightly different approach: We cannot pass the binding for the context variable in the same way we passed the other bindings. The computation of the context function must be performed bottom-up because it depends on the environment of a sub-expression. Hence, each chk_{t_j} has one more result component, the *local extended context*. Such an object is a function taking a list of type $[t_0]$ and one argument for each pattern p_i and yields a result of type

t_j . The local context produced by the top-level evaluation of chk_{t_0} is the binding for the context variable. The list of all unvisited substructures is computed bottom-up as well.

Let t_{pxs} be a tuple of the types of all variables in the pattern p_i . We can now give the complete type of chk_{t_j}

$$chk_{t_j} :: (\text{Int}, t_{pxs}) \rightarrow t_j \rightarrow (\text{Int}, t_{pxs}, [t_0], [t_0] \rightarrow t_1 \rightarrow \dots \rightarrow t_n \rightarrow t_j)$$

To initialise the arguments for bindings which are not yet found we use the undefined value of arbitrary type: `undefined = error ""` as defined in the prelude. The complete top-level structure of the translation now looks like this (we abbreviated `undefined` by \perp):

```
case e0 of {c p1 if g1 ... pn if gn -> e ; _ -> e' }
~> let { <chkt0-decl> ; ... ; <chktm-decl> }
    in case (chkt0 (1, ⊥, ..., ⊥) e0) of {
        (n + 1, x1,1, ..., xn,kn, c_⊥, c_e) -> let { c = c_e c_⊥ } in e ;
        _ -> e' }
```

where $x_{i,1}, \dots, x_{i,k_i}$ are the variables in p_i .

Example: For `rplc` we have only one variable `z` in the pattern p_1 :

```
rplc x y l = let chk0 ... x0 = ...
              chk1 ... x0 = ...
              in case (chk0 (1, undefined) l) of
                  (2, z, c_⊥, c_e) -> let c = c_e c_⊥
                                      in c_e (map (rplc x y) c_⊥) y
              _ -> 1
```

The implementation of chk_{t_j} checks the number of the next pattern and handles each of the cases. Let n^i be new variables, $\bar{y}^i := y_{1,1}^i, \dots, y_{n,k_n}^i$ be vectors of unused variables, one variable for each variable in the patterns, and $\bar{z} := z_1, \dots, z_n$ be a vector of variables for the arguments of a context function:

$$\begin{aligned} \langle chk_{t_j} - decl \rangle \rightsquigarrow chk_{t_j} (n^0, \bar{y}^0) x = \text{case } n^0 \text{ of } & 1 \rightarrow \langle chk_{t_{j,1}} \rangle \\ & \dots \\ & n \rightarrow \langle chk_{t_{j,n}} \rangle \\ & n + 1 \rightarrow \langle rchk_{t_j} \rangle \end{aligned}$$

If all patterns are found (the last case) then what remains to be done is the search for all parts of type t_0 . The right hand side $\langle rchk_{t_j} \rangle$ is essentially the same as $\langle chk_{t_{j,i}} \rangle$ would be for a pattern $p_i = x$ of type t_0 except that the number of patterns found is not increased. We suspend the explanation of $\langle rchk_{t_j} \rangle$ until we have finished $\langle chk_{t_{j,i}} \rangle$.

For the realisation of the right hand side $\langle chk_{t_{j,i}} \rangle$ which checks for pattern p_i , we distinguish whether p_i can be found in a value of type t_j . Using the inference system from the last section, we can formalise this by $A \vdash t_j \rightarrow (\dots, t_{p_i}, \dots)$.

If it cannot be found, the implementation of $\langle chk_{t_j,i} \rangle$ is trivial, because there is no need for recursive search. All we have to do is to create a trivial context. Note that in contrast to the case that all patterns are found, we do not search for part of type t_0 inside this component: The extended context expects a list of the unvisited parts only.

However, if the pattern can occur is this sub-expression, $\langle chk_{t_j,i} \rangle$ proceeds in two steps:

1. It checks whether the pattern can occur directly (done by $\langle p_i - chk_{t_j} \rangle$).
2. It performs a recursive search in the sub-expressions (done by $\langle r_{t_j} - chk \rangle$).

Hence, we can define $\langle chk_{t_j,i} \rangle$ in the following way:

$$\langle chk_{t_j,i} \rangle \rightsquigarrow \begin{cases} \text{case } x^0 \text{ of } \{ \\ \quad \langle p_i - chk_{t_j} \rangle \\ \quad \langle r_{t_j} - chk \rangle \\ \quad - \rightarrow (i, \bar{y}^0, [], \backslash l \rightarrow \backslash \bar{z} \rightarrow x^0) \\ \quad \} \\ \quad \text{if } A \vdash t_j \rightarrow (\dots, t_{p_i}, \dots) \\ (i, \bar{y}^0, [], \backslash l \rightarrow \backslash \bar{z} \rightarrow x^0) \text{ otherwise} \end{cases}$$

If t_j is the type of p_i then the pattern can occur directly; Otherwise, there is nothing to do in $\langle p_i - chk_{t_j} \rangle$. Hence the code $\langle p_i - chk_{t_j} \rangle$ is defined as

$$\langle p_i - chk_{t_j} \rangle \rightsquigarrow \begin{cases} p_i \mid g_i \rightarrow (i+1, & \text{if } t_j = t_i \\ \quad y_{1,1}^0, \dots, y_{i-1,k_{i-1}}^0, \\ \quad x_{i,1}, \dots, x_{i,k_i}, \\ \quad y_{i+1,1}^0, \dots, y_{n,k_n}^0, \\ \quad [], \backslash l \rightarrow \backslash \bar{z} \rightarrow z_i); \\ \varepsilon & \text{otherwise} \end{cases}$$

The value consist of the following entries: the number of the next pattern ($i+1$), updated bindings for variables in the patterns, an (empty) list of unvisited sub-structures, and an extended context function returning the i -th argument (“ l ” “ \bar{z} ” “ z_i ”). The updated bindings consists of three groups: the unchanged values of the variables in the preceding patterns ($y_{1,1}^0, \dots, y_{i-1,k_{i-1}}^0$), the values of the variables in the current pattern ($x_{i,1}, \dots, x_{i,k_i}$), and the unchanged values of the variables in the succeeding patterns ($y_{i+1,1}^0, \dots, y_{n,k_n}^0$) (which are all equal to \perp). Note that each value in the vector is updated only once, because they are indexed by the number of the current pattern, which strictly increases. The construction of the context function is correct because by matching pattern p_i in the expression this sub-expression is “chopped” away.

The additional context pattern guard following the keyword **if** is implemented by the normal pattern match guard.

Example: For the **rplc** example, the pattern $p_1 = \mathbf{z}$ can occur directly in expressions of type $t_1 = \mathbf{a}$, but not recursively. Hence, the complete definition of **chk1** is:


```

chk1 (n0,xb0) x0 = case n0 of
  1 -> case x0 of
    z | z==x -> (2,z,[],\l->\z->z)
    -       -> (1,xb0,[],\l->\z->x0)
  2 -> (2,xb0,[],\l->\z->x0)

```

What remains is the implementation of the recursive search, which is performed if the pattern cannot match directly, or does not match. We have to know all constructors of type t_j to handle all possible values. Let $K_{t_j,1}, \dots, K_{t_j,n_j}$ be all constructors of type t_j and let $a_{j,i}$ be the arity of constructor $K_{t_j,i}$. The expressions $\langle r_{t_j}-chk \rangle$ which perform the recursive search have the following form (with w_i new variables):

$$\begin{aligned}
\langle r_{t_j}-chk \rangle &\rightsquigarrow K_{t_j,1} w_1 \dots w_{a_{j,1}} \rightarrow \langle chkrek_{t_j,1} \rangle ; \\
&\dots \\
&K_{t_j,n_j} w_1 \dots w_{a_{j,n_j}} \rightarrow \langle chkrek_{t_j,n_j} \rangle ;
\end{aligned}$$

All values of type t_j will match exactly one of these cases. To define the right hand side $\langle chkrek_{t_j,k} \rangle$ we abbreviate $K := K_{t_j,k}$ and $a := a_{j,k}$. This expression must traverse all sub-expression w_l and create new context functions from the results. If $a = 0$ then there is nothing to. Otherwise, all sub-expressions are traversed from left to right and the resulting context functions are combined by using the constructor K . The resulting list of unvisited parts is the concatenation of the lists of all substructures and the resulting local extended context splits its list argument into parts corresponding to the numbers of those parts.

Let t_{l_1}, \dots, t_{l_a} be the argument types of constructor K , and let $c_{\perp^1}^i, c_{\perp^e}^i, n^i, x$ be new variables. We define: $\langle chkrek_{t_j,k} \rangle$ in the following way:

$$a = 0: \langle chkrek_{t_j,k} \rangle \rightsquigarrow (n^0, \bar{y}^0, [], \backslash l \rightarrow \backslash \bar{z} \rightarrow x^0)$$

$$\begin{aligned}
a > 0: \langle chkrek_{t_j,k} \rangle &\rightsquigarrow \text{let } (n^1, \bar{y}^1, c_{\perp^1}^1, c_{\perp^e}^1) = chk_{t_{l_1}} (n^0, \bar{y}^0) w_1 \\
&\dots \\
&(n^a, \bar{y}^a, c_{\perp^1}^a, c_{\perp^e}^a) = chk_{t_{l_a}} (n^{a-1}, \bar{y}^{a-1}) w_a \\
&\text{in } (n^a, \bar{y}^a, c_{\perp^1}^{1++\dots+a}, c_{\perp^e}^a, \\
&\quad \backslash l \rightarrow \backslash \bar{z} \rightarrow \text{let } (l^1, r^1) = \text{splitAt } (\text{length } c_{\perp^1}^1) l \\
&\quad (l^2, r^2) = \text{splitAt } (\text{length } c_{\perp^1}^2) r^1 \\
&\quad \dots \\
&\quad (l^a, r^a) = \text{splitAt } (\text{length } c_{\perp^1}^1) r^{a-1} \\
&\text{in } K (c_{\perp^e}^1 l^1 \bar{z}) \dots (c_{\perp^e}^a l^a \bar{z})
\end{aligned}$$

Finally, we have to define $\langle rchk_{t_j} \rangle$, which performs the search for all unvisited parts of type t_0 . This is done as soon as all pattern have matched and here the functional parameter for the extended context is used. We define

$$\langle rchk_{t_j} \rangle \rightsquigarrow \begin{cases} (n+1, [x^0], \bar{y}^0, \backslash[x] \rightarrow \backslash\bar{z} \rightarrow x) & \text{if } t_j = t_0 \\ \langle r_{t_j} \text{-} chk \rangle & \text{if } t_j \neq t_0, A \vdash t_j \rightarrow (\dots, t_0, \dots) \\ (n+1, \bar{y}^0, \backslash l \rightarrow \backslash\bar{z} \rightarrow x^0) & \text{otherwise} \end{cases}$$

Example: *The complete program for rplc is*

```

rplc x y l = let
  chk0 (n0,xb0) x0 = case n0 of
    1->case x0 of
      []      ->(n0,xb0, [], \l->\z->x0)
      (w1:w2)->let (n1,xb1,c11,c1)=chk1 (n0,xb0) w1
                  (n2,xb2,c12,c2)=chk0 (n1,xb1) w2
                  in (n2,xb2,c11++c12,
                     \l->\z->let (l1,r1)=splitAt (length c11) l
                               (l2,r2)=splitAt (length c12) r1
                               in ((c1 l1 z):(c2 l2 z)))
      -      ->(1,xb0, [], \bsl->\bsz->x0)
    2->(2,xb0,[x0], \[x]->\z->x)
  chk1 (n0,xb0) x0 = case n0 of
    1->case x0 of
      z | z==x -> (2,z,[], \l->\z->z)
      -      -> (1,xb0,[], \l->\z->x0)
    2->(2,xb0,[], \l->\z->x0)
  in case (chk0 (1,undefined) l) of
    (2,z,c_l,c_e) -> let c = c_e c_l
                    in c_e (map (rplc x y) c_l) y
    -      -> 1

```

5 Related Work

Patterns beyond the scope of the Haskell pattern have been studied in [7, 5, 22] in the context of *TrafoLa*, a functional languages for program transformations. Their *insertion patterns* allow an effect similar to our context patterns, but instead of modelling a context as a function they introduce special *hole constructors* @*n* to denote the position where a match was cut from the context. Additionally, they introduce several other special purpose patterns for lists, and allow non-linear patterns, which may interfere with lazy evaluation [15, p. 65]. Pattern matching usually results in a list of solutions.

An even more general approach was taken in [10] where *second-order schemes* are used to describe transformation rules. These allow the specification and selection of arbitrary subtrees but are not integrated in a functional language.

The language *Refal* [20] which is used as the basis for supercompilation has strings as data structure and allows patterns like $s_1 e_x s_1$ where s_1 is a string and e_x is a character.

In [17] patterns with *segment assignments* and their compilation are studied in the context of Lisp. The segments allow the access to parts of a matched list, e.g. the pattern $(?x \text{ ??}y \text{ ?}x)$ matches all lists which start and end with x . The inner part of the list can be accessed via y .

A different extension of pattern matching is to use *unfree* data types, where the matching or a pattern is done by evaluation of a user-defined function. Different approaches based on this idea are *Miranda's laws*, *Wadler's views* [21], and *Erwig's active patterns* [4].

Another root of our work can be seen in *higher-order unification* [9, 19, 3]. The general approach is to synthesise λ -terms in order to find bindings for free function variables in applications, such that equations β -reduce to equal terms. In general, this problem is undecidable [6]. However, for certain subclasses of generated λ -terms and equations, this problem becomes decidable [16]. Especially in our case, where only the pattern can contain unbound variables, i.e. unification becomes matching, the problem is decidable [9].

The representation of context by functions are related to [11], where lists of type $[a]$ are represented by a function of type $[a] \rightarrow [a]$. Given a list l , the representation is obtained by **append** l . In our setting such functions can occur as a special case, where the “hole” is the rest of the list.

Another attempt to solve problems very close to those addressed by context patterns are *generalised folds (catamorphisms)* [18]. The underlying observation is that a **fold** function can be derived for any data type from its definition. Subsequently, it can be used to perform a recursive traversal of a object of that type, replacing each constructor by a function application of corresponding type. However, this differs from our approach in several ways:

- A generalised **fold** traverses the complete data structure, whereas the match of a context pattern searches the structure only until the subpatterns are located, and not further.
- Each occurrence of a constructor is replaced by the same function during evaluation of a generalised **fold**. Context patterns do not have this restriction. Of course, it is possible to encode a state² in a generalised **fold** and to use the state to distinguish different occurrences of the same constructor. However, the resulting programs very soon become far too complicated.
- Context patterns allow to find patterns not restricted to one constructor and they can be used to locate several independent patterns. This is not possible with generalised **folds**.
- All types of constructors must be replaced by a generalised **fold**, even if only some of them are of interest. The whole point of context patterns is to focus on the interesting situations, without the necessity to think about the uninteresting ones.

² The same idea is used in the monadic **fold** for lists in *Haskell* 1.4.

6 Conclusions and Future Work

In [13] we have presented an extension of pattern matching called context patterns, which allow the matching of regions not adjacent to the root and their corresponding contexts as functional bindings. Typical examples of functions using this increased expressive power are functions which search and transform data structures. For special cases of transformations, however, this approach caused a superfluous repetition of the recursive search.

In this paper, we have introduced a new construct called *extended context*, which allows the definition of transformational functions without this additional overhead.

Furthermore, we have formally defined the static semantics of context patterns by an inference system and explained the translation into **Haskell** in detail.

The syntax of the extended context patterns is not yet satisfying. In the original workshop presentation, the extended context function had an additional functional parameter, which was applied to all unvisited structures. However, the discussion showed that this is too restrictive: The function `beta'` from Section 2.2 would not have been possible. Hence, we made the list of all unvisited structures explicit. However, this approach lacks a lot of the original elegance and we intend to find a better solution for this problem.

As future research we plan to gain more experience in the use of context patterns and to validate the usefulness of context patterns for transformations. Therefore, we want to re-implement (parts of) the Simplifier, which is a component of the Glasgow Haskell Compiler performing simple performance-enhancing source code transformations.

Another possible future work can be based on the observation that a context function c and an associated extended context function c_e differ only in the presence of an argument. The distinction between these two context functions could be avoided if the type system would support function types with *optional argument and default values*, similar to imperative languages like **C++** or **Ada**. For our case, we would have only one context function, whose first parameter is an optional argument with default value `c_1`.

We have integrated context patterns in the Glasgow Haskell Compiler, based on version 2.01. The implementation is described in more detail in an accompanying paper [14]. However, the extension presented in this paper is not yet included. The source code can be obtained from the URL:

<http://www-i2.informatik.rwth-aachen.de/markusm/CP/>.

References

- [1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics*. North-Holland, 1985.
- [2] P. Deransart and J. Małuszyński, editors. *Proceedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming (PLILP)*, number 456 in *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

- [3] D. J. Dougherty and P. Johann. A Combinatory Approach to Higher-Order *E*-Unification. *Theoretical Computer Science*, 139(1–2):207–242, March 1995.
- [4] M. Erwig. Active Patterns. In Kluge et.al. [12], pages 21–40.
- [5] C. Ferdinand. Pattern Matching in a Functional Transformational Language using Treeparsing. In Deransart and Małuszyński [2], pages 358–371.
- [6] W. D. Goldfarb. The Undecidability of the Second-Order Unification Problem. *Theoretical Computer Science*, 13(2):225–230, February 1981.
- [7] R. Heckmann. A Functional Language for the Specification of Complex Tree Transformations. In H. Ganzinger, editor, *Proceedings of the 2nd European Symposium on Programming (ESOP)*, number 300 in Lecture Notes in Computer Science, pages 175–190. Springer-Verlag, 1988.
- [8] P. Hudak, S. L. Peyton Jones, and P. Wadler et. al. Report on the Programming Language Haskell — A Non-strict, Purely Functional Language. *ACM SIGPLAN Notices*, 27(4), 1992.
- [9] G. Huet. A Unification Algorithm for Typed λ -Calculus. *Theoretical Computer Science*, 1:27–57, 1975.
- [10] G. Huet and B. Lang. Proving and applying Program Transformations Expressed with Second Order Patterns. *Acta Informatica*, 11:31–55, 1978.
- [11] R. J. M. Hughes. A Novel Representation of Lists and Its Application to the Function “reverse”. *Information Processing Letters*, 22(3):141–144, March 1986.
- [12] W. Kluge et.al., editor. *Selected Papers of the 8th International Workshop on Implementation of Functional Languages (IFL)*, number 1268 in Lecture Notes in Computer Science. Springer-Verlag, 1997.
- [13] M. Mohnen. Context Patterns in Haskell. In Kluge et.al. [12], pages 41–58.
- [14] M. Mohnen and S. Tobies. Implementing Context Patterns in the Glasgow Haskell Compiler. Technical Report AIB-97-04, RWTH Aachen, 1997.
- [15] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- [16] C. Prehofer. Decidable Higher-order Unification Problems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction (CADE)*, number 814 in Lecture Notes in Computer Science, pages 635–649. Springer-Verlag, 1994.
- [17] C. Queinnec. Compilation of Non-Linear, Second Order Patterns on S-Expressions. In Deransart and Małuszyński [2], pages 340–357.
- [18] T. Sheard and L. Fegaras. A Fold for All Seasons. In *Proceedings of the 6th Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 233–242. ACM, September 1993.
- [19] W. Snyder and J. Gallier. Higher Order Unification Revisited: Complete Sets of Transformations. *Journal of Symbolic Computation*, 8(1 & 2):101–140, 1989. Special issue on unification. Part two.
- [20] V. F. Turchin. Program Transformation by Supercompilation. In H. Ganzinger and N. Jones, editors, *Programs as Data Objects*, number 217 in Lecture Notes in Computer Science, pages 257–281. Springer-Verlag, 1986.
- [21] P. Wadler. Views: A Way for Pattern-matching to Cohabit with Abstraction. In *Proceedings of the 14th Symposium on Principles of Programming Languages (POPL)*, pages 307–313. ACM, January 1987.
- [22] R. Wilhelm. Tree Transformations, Functional Languages, and Attribute Grammars. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, number 461 in LNCS, pages 116–129. Springer-Verlag, 1990.

A Compacting Garbage Collector for Unidirectional Heaps

Kent Boortz¹ and Dan Sahlin²

¹ `kent@erlang.ericsson.se`,

Erlang Systems, Ericsson Software Technology, Sweden

² `dan@cslab.ericsson.se`,

Computer Science Laboratory, Ericsson Telecom, Sweden

Abstract. A unidirectional heap is a heap where all pointers go in one direction, e.g. from newer to older objects. For a strict functional language, such as Erlang, the heap may be arranged so that it is unidirectional. We here present a compacting garbage collection algorithm which utilizes the fact that a heap is unidirectional.

Only one memory space is used in our algorithm. In fact, no extra memory is used at all, not even any reserved bits within the cells. The algorithm is quite easy to extend to a variant of generational garbage collection.

1 Introduction

The Erlang functional programming language [5] is typically implemented using a copying garbage collection algorithm. In [4] a completely new method was suggested utilizing the fact that objects on the heap may be allocated so that each object only points to earlier allocated ones. In order to be able to deallocate data in the middle of the heap, without disturbing the order of objects, all objects were linked, each with a ‘history cell’ showing the time of creation.

Here we propose an alternative method to utilize unidirectionality of the heap which has a quite standard representation of the heap, not using linked lists or history cells.

One main motivation to develop the algorithm was to be able to design a garbage collector for an Erlang engine that could run on systems with more limited memory resources. The algorithm presented has the following properties:

- It does not copy data to a new heap, thereby minimizing memory fragmentation.
- No extra memory is used, not even mark-bits or extra tags.
- The order of the objects on the heap is preserved.
- Execution time is linear proportional to the size of the data areas.
- A version of generational garbage collection is simple to implement.
- Overlapping objects are handled.

The algorithm has two phases:

1. Marking, pointer updating and compaction, all combined.
2. Sliding the heap and updating external pointers.

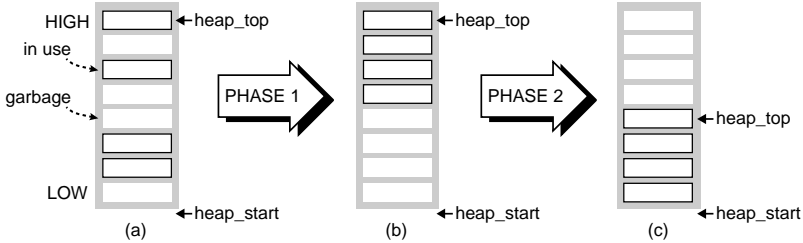


Fig. 1. Phases of the garbage collector

The second phase may sometimes be omitted as a correct heap is returned although in “the wrong place”. This optimization is discussed in Sect. 4.5.

The paper is structured as follows. First some basic assumptions are discussed, then the algorithm itself is presented, followed by some optimizations and variants. Finally, the approach is discussed and compared to other approaches.

2 Basic Assumptions

The most unusual assumption of the algorithm is the unidirectionality of all heap pointers, i.e. all pointers point from newer to older objects, which will be shown in more detail below.

2.1 Memory Model

In a typical Erlang implementation there are four memory areas to be considered:

1. Registers: holding function arguments and local variables
2. Stack: holds function activation records
3. Heap: holds dynamically created objects
4. “Static” data: e.g. code and symbol tables

The algorithm presented will only free memory in the heap. Three basic assumptions are made:

1. There may not be references from the heap to the stack or to the registers.
2. All references to the heap come from the registers or the stack.
3. All references in the heap are *unidirectional*, i.e. go from a more recently allocated object to an earlier allocated one. This is because new objects are created on the top of the heap in sequential order.

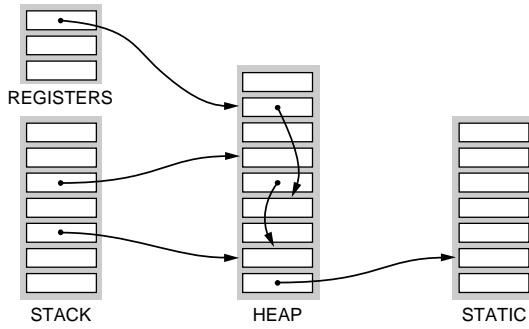


Fig. 2. The data areas

The first two assumptions are quite standard. For most programming languages, e.g. lazy functional, logical or imperative languages, the third assumption would be unrealistic. For a strict functional language without updateable structures, like Erlang, it is in fact an assumption that is almost automatically fulfilled as newly created objects can only point to older ones. However, a copying garbage collection algorithm would immediately destroy this property. Our algorithm, in contrast, preserves the unidirectionality and will also strongly utilize it.

2.2 Unidirectional Memory in Erlang

At first, the unidirectionality assumptions may seem unreasonable in the sense that no useful programming language could fulfill them. However, for Erlang the unidirectionality assumptions can be fulfilled, and Erlang is indeed a practical and useful programming language, as demonstrated by a number of very large programs (more than 100.000 lines of code) used in products being sold by Ericsson[3].

Perhaps one key factor making Erlang a practical language is the fact that processes may be created within the language. The processes communicate by message passing. A special data type, *process identifier*, is used when referring to a process. Each process has its own heap which is garbage collected separately as there may be no references into the heap from other processes.

However, a heap may well contain a process identifier referring to a different process, thereby effectively making it possible to create cyclic process references. As only the heaps of the processes, not the processes themselves are being garbage collected, these cyclic references cause no problem for our garbage collection algorithm.

2.3 Data Representation

The description is simplified by assuming that the heap only contains three types of objects; tuples, list cells and atoms. All cells are tagged. Typically a cell fits

into a 32-bit word, with a few bits used for the tag and the rest of the word containing a pointer or an atom identifier. Note that no bits or tags are reserved for garbage collection. In the following pseudo-C code `tag` is typically two bits and union `v` is the remaining 30 bits.

```
typedef struct cell {
    unsigned int tag;           /* ATOM, LIST, TUPLE, ARITY */
    union { struct cell *ptr;   /* LIST or TUPLE pointer */
            int id;            /* ATOM id */
            int size;          /* ARITY size */
        } v;
} Cell;
```

An `ATOM` cell corresponds to an Erlang atom, e.g. ‘hello’, where the `id` field is unique for the atom. A `LIST` cell points to the first of two consecutive cells. A `TUPLE` points to a sequence of cells where the first cell is an `ARITY` cell containing the length of the sequence.

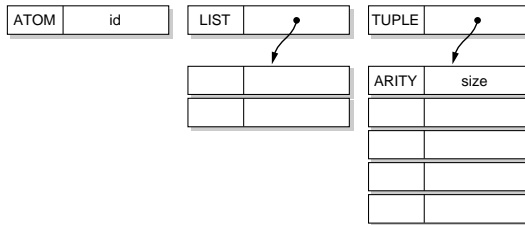


Fig. 3. Representation of data types in Erlang

In the garbage collection algorithm it will be assumed that the pointer to an object points to the part of the object closest to the top the heap. However, as will be shown in Sect. 4.6, this assumption may be lifted and that in fact most other kinds of objects could also be handled by the algorithm.

3 Garbage Collection Algorithm

3.1 Relocation Chains

The *relocation chains* are central for the garbage collection algorithm. A relocation chain for a heap cell `c` is a linked list starting at `c` and containing all the cells that originally pointed to `c`. The original contents of the heap cell is found at the end of the chain.

But having a relocation chain, how is the end of the chain found? It may contain a pointer just as the rest of the cells!

Here the unidirectionality of the memory area is used. Since all heap cells originally only contain pointers pointing downwards, all cells but the last one in a relocation chain point upwards. For a member of the relocation chain located on the *stack* (which we define to be higher than the heap) this is trivially true. As the stack is swept from high to low it will remain true even if several pointers from the stack point to the same cell on the heap.

3.2 Invocation of the Garbage Collection

Only objects not reachable from a set of *root nodes* may be deallocated during garbage collection. These root nodes are the current registers¹ and the stack frames. The structure of the algorithm is shown below.

```
garbage_collection()
{
    int distance;
    push_registers();
    link_cells_in_stack();
    distance = collect_heap();
    slide_heap(distance);
    update_stack_pointers(distance);
    pop_registers();
}
```

In order to be able to point to the registers during garbage collection, these are pushed onto the stack. The procedure `push_registers()` which does this is not shown in detail here.

To simplify the description of the rest of the algorithm, we will assume that the stack is located above the heap. The algorithm may easily be modified to cater for a different placement of the heap by making the function `points_up()` somewhat more complicated.

¹ A mechanism must exist for deciding which registers are “current”. This is often determined by looking at the code pointed to by the program counter. We will not elaborate on this, as this is a standard practice in garbage collection.

3.3 Linking Cells in the Stack

All cells, including the pushed registers, in the stack are scanned for pointers to the heap by `link_cells_in_stack()`, see Fig. 4.

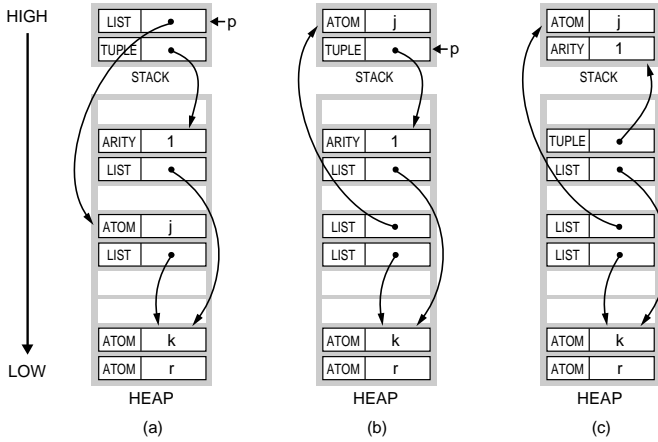


Fig. 4. Linking the cells in the stack. The two cells in the stack point to the Erlang terms $[j \mid [k \mid r]]$ and $\{[k \mid r]\}$. In (b) the reference to the `LIST` has entered its relocation chain, and in (c) the reference to the `TUPLE` has also entered its relocation chain.

```
link_cells_in_stack()
{
    for each cell p in the stack from high to low
        if p is a pointer to the heap
            link_cell(*p,p);
}

static void link_cell(Cell c, Cell *to)
{
    Cell *x;

    x = c.v.ptr;
    *to = *x;
    c.v.ptr = to;
    *x = c;
}
```

The procedure `link_cell(c,to)` will link cell `c` into the relocation chain for the heap cell that `c` points to. `link_cell()` is more general than needed by `link_cells_in_stack()` as `c` is always equal to `*to`. In the main procedure, `collect_heap`, that will however not be the case.

3.4 Collecting the Heap

In this section, the main garbage collection procedure, `collect_heap()` (Fig.5), is described. It will remain true that all but the last member of the relocation chain in the *heap* will also point upwards.

```
static int collect_heap()
{
    Cell *to, *from;

    to = heap_top;
    fields = 0;
    for (from = heap_top; from > heap_start; from--)
    {
        /* update pointers in relocation chain */

        Cell c = *from;

        while (is_pointer(c.tag) && points_up(to,c.v.ptr))
        {
            fields = max(fields,pointer_length(c.tag));
            c = unlink_cell(c, to);
        }

        if (fields > 0)                /* cell is reachable */
        {
            fields--;
            fields = max(fields,object_length(c));
            if(is_pointer(c.tag))
                link_cell(c,to);      /* enter relocation chain */
            else
                *to = c;
            to--;
        }
    }
    return (to - heap_start);
}
```

Fig. 5. The main garbage collection procedure: `collect_heap()`

Two global variables are used:

```
Cell *heap_start;
    /* pointing to the cell just before the heap bottom */
Cell *heap_top;
    /* pointing to the most recently allocated heap cell */
```

When entering the heap we know exactly which objects are reachable from the stack: those in a relocation chain, i.e. pointing upwards. The pointing direction will be used instead of a mark bit.

When collecting the garbage, the heap is swept from high to low using a **from** pointer. Each cell determined not to be garbage gets moved to the next free cell at the top of the heap (the **to** cell).

There are two ways a cell may not be garbage: it is either reachable directly from another cell, or the cell is a field in a larger object whose head is reachable. Recall that all cells reachable from the stack have pointers going up.

During the traversal of the heap, if a cell points up, it is the first cell in a relocation chain, and all cells in that chain should now be changed to point to the new location of the cell. This operation is performed by calling `unlink_cell(c,to)` until the end of the chain is reached.

While traversing the relocation chain, the variable `fields` is updated so that it contains the length of the referred object. The function `pointer_length()` returns the length of an object. However, for the Erlang tuples, the length is stored in the object itself, and this will also be catered for below. For simplicity and generality, the maximum of all `pointer_length()` calculations will be stored in `fields`.

If `fields` becomes greater than zero, the cell is reachable and is moved towards the top of the heap. The function `object_length()` checks if the cell has the tag `ARITY`, in which case the field gets updated to the length of the tuple.

Finally, for a reachable object, if it is a pointer, it is entered into the relocation chain for the cell that it refers to.

Thus `collect_heap()` compacts the heap and updates all pointers. Unfortunately the heap gets compacted in the “wrong” direction, towards the top of the heap, so it has to be slid down. The distance that the heap needs to get slid is returned by `collect_heap()` and is used by `slide_heap()`.

```
static Cell unlink_cell(Cell c, Cell* to)
{
    Cell *next, next_c;

    next = c.v.ptr;
    next_c = *next;
    c.v.ptr = to;
    *next = c;
    return next_c;
}
```

```

int is_pointer(unsigned tag)
{
    return (tag == LIST || tag == TUPLE);
}

static int points_up(Cell *addr, Cell *ptr)
{
    return (addr < ptr);
}

static int pointer_length(unsigned tag)
{
    if(tag == LIST)
        return 2;
    else
        return 1;
}

static int object_length(Cell c)
{
    if(c.tag==ARITY)
        return c.v.size;
    else
        return 0;
}

```

The **from**-cell in Fig. 6 (a) points upwards indicating it being reachable and the beginning of a relocation chain. An **unlink_cell** operation is therefore performed on this **TUPLE** cell which has arity 1.

In (b) the **from**-cell points downwards, but is nevertheless reachable as a **TUPLE** structure of arity 1 contains two cells. So the **from**-cell containing a **LIST** pointer has to be linked into the relocation chain for the list being referred.

The heap is continued being scanned downwards with the **from** pointer until a cell containing an upward going pointer is found in figure (c). An **unlink_cell** operation is performed for the **LIST** cell found, see figure (d). As a list structure always contains two cells, the next cell, pointed to by **from** is also reachable.

That cell also contains a **LIST** pointer, and is entered into the relocation chain for that list and the **from**-pointer is advanced to the next up-going pointer, see figure (e).

The **from**-cell now points to the beginning of a relocation chain, containing two elements, corresponding to the fact that originally there were two pointers to this list cell, see Fig. 4 (a).

In (f) that relocation chain has been traversed, updating the pointers to the new location of the list structure. Finally, in (g) the **ATOM** is moved as it is reachable being the second of the two list cells.

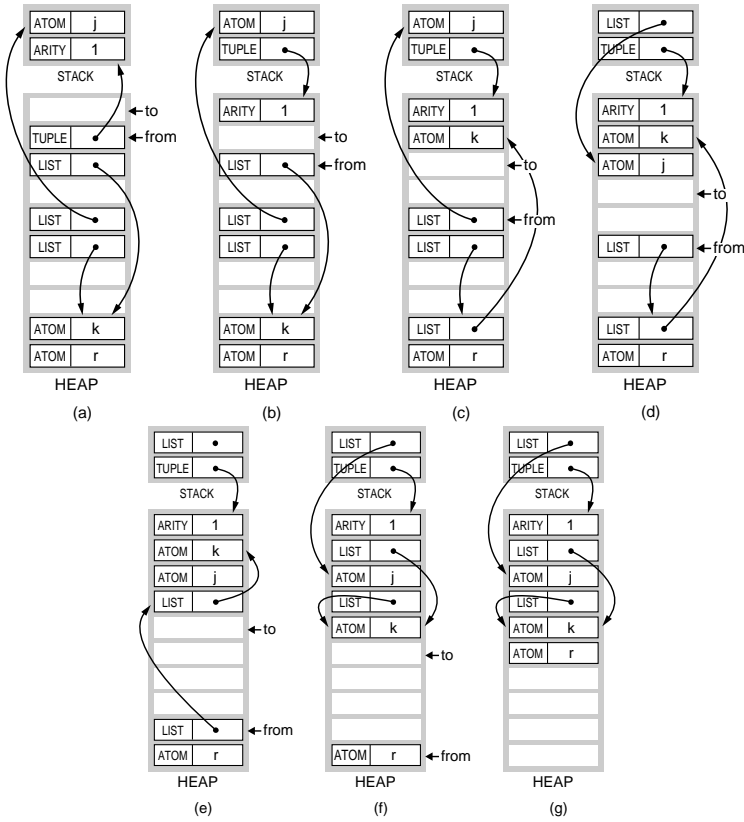


Fig. 6. Compacting the heap, continuing the example from Fig. 4

3.5 Heap Sliding

When the heap gets slid down all pointer objects become updated.

```
static void slide_heap(int distance)
{
    Cell *from, *to, c;
    to = heap_start+1;
    for(from = to+distance; from <= heap_top; from++,to++)
    {
        c = *from;
        if(is_pointer(c.tag))
            c.v.ptr -= distance;
        *to = c;
    }
    heap_top = to-1;
}
```

Also the stack needs to get updated as it still contains pointers to the old locations on the heap. This is done by calling `update_stack_pointers(distance)` which updates all pointers to the heap. It now only remains to restore all registers by calling `pop_registers()`.

4 Performance

One way to estimate the performance of a garbage collection algorithm is to look at how many read and write operations are required to the main memory for each cell.

If a cell is not reachable, it nevertheless has to be read once, as shown in Fig. 7. If overlapping objects are not allowed, this may be reduced to one read operation per *object* instead of one read operation per *cell*, see Sect.4.6 (“Non overlapping objects”).

cell type		copying gc	compacting gc
non-reachable		0	1R
reachable	non-pointer	2R + 1W	2R + 2W
reachable pointer	unique	2R + 3W	4R + 4W
	non-unique	3R + 2W	

Fig. 7. Comparison between Cheney copying GC and our compacting GC with respect to the number of read (R) and write (W) operations for each cell to main memory.

A reachable cell that is not a pointer (i.e. an `ATOM` or `ARITY` cell) is copied to the new location requiring one read and one write operation. Together with the heap sliding in pass two, which performs one read and one write operation per reachable cell, that sums up to two read operations and two write operations per cell.

In pass one, a reachable cell containing a pointer will be swapped with what it points to and copied in one step requiring two read and two write operations. We then have to add one read and one write operation as part of the unlinking. In pass two, the heap sliding adds one read and one write operation, so in total we get four read and four write operations for a reachable pointer cell.

The table in Fig.7 compares our algorithm with a typical Cheney copying garbage collector[10]. The work done for each reachable cell is higher in our algorithm mainly because of the heap sliding that requires one additional read and write operation. In Sect.4.5 it is shown how the heap sliding phase may be eliminated, thereby making the two methods more comparable. Our algorithm reads data that is garbage and this is a major drawback compared to the Cheney algorithm if the heap contains lots of garbage. Sections 4.4 and 4.6 discuss how to improve on this situation.

4.1 Generational Garbage Collection

Some programs gain garbage collection speed by only performing garbage collection to the top cells of the heap[7]. The rational for this is that usually most objects are short-lived, and garbage collection for the newly collected objects may be particularly useful.

Generational garbage collection is extremely simple to implement, just two things have to be changed:

1. **heap_start** is moved up in the heap. In fact, **heap_start** may be set at any point in the heap, thereby making it very flexible to decide how much of the heap will be garbage collected.
2. **is_pointer()** returns false for all pointers below **heap_start**

This optimization resembles generational garbage collection [7], but old objects are never moved into “old space” as the order of the objects must be preserved.

4.2 Towards Real-Time Garbage Collection

One of the main motivations in [4] to utilize the unidirectionality of the heap was the possibility to get a truly real-time garbage collector. In their system the garbage collector may interleave with ordinary execution, and suspend and resume at any time.

In the currently distributed Erlang JAM system each process has its own heap [2] and uses a traditional copying algorithm. This makes the average process heap a lot smaller than in a unified heap system and garbage collection time has less influence on the response time. But in order to maintain good real-time response, each process heap must not become too large in JAM.

Our algorithm is not a true real-time garbage collector as [4], but will like [2] collect each process heap separately. However, by invoking generational garbage collection the execution time collecting each heap can be made even shorter. The garbage collection time can be fairly accurately predicted as it is limited by a linear function of the heap size to be collected and the stack size. However, when the oldest cells become garbage it may occasionally be necessary to do a complete garbage collection.

4.3 Trimming the Heap

If the most recently created structures are garbage, the top of the heap will contain only non-reachable data.

Again we may utilize the fact that the heap is unidirectional, as we know that nothing in the heap above the topmost cell with an up-going pointer is reachable.

By keeping track of which cell is the topmost reachable heap cell when **link_cells_in_stack()** is executed, the heap only needs to be scanned starting from that cell and downwards.

In a similar manner, `collect_heap()` and `link_cells_in_stack()` may be extended to keep track of the lowest object reachable. When that object is reached during heap scanning, and it does not contain any pointers, that will be the last object reachable on the heap.

An alternative way of finding the last object on the heap is using a global reference counter, as outlined below.

4.4 A Global Reference Counter

The algorithm may be enhanced by maintaining a global reference counter which counts up for each call to `link_cell()` and down for each `unlink_cell()`. When this global reference counter reaches 0, we know that all reachable objects have been handled, and there is no need to continue sweeping the heap.

If that global reference counter is called `ref_count`, besides the changes in `link_cell` and `unlink_cell` only `collect_heap()` needs to get modified as shown in Fig. 8.

Normally the counter seldom would reach 0 much before the end of the heap, as most programs create long-lived data in the beginning of an execution. But in combination with a generational garbage collector, those long-lived data would not add to the global reference counter, making this enhancement more likely to be useful.

```

if (fields > 0)                /* cell is reachable */
{
    fields--;
    fields = max(fields,object_length(c));
    if(is_pointer(c.tag))
        link_cell(c,to);      /* enter relocation chain */
    else
        *to = c;
    to--;
}
else if (ref_count == 0)
    break;

```

Fig. 8. Adding a global reference counter. `collect_heap()` in Fig.5 is changed so that the heap sweep is stopped as soon as `ref_count` reaches 0 and `fields==0`

4.5 One Pass GC: Skipping the Heap Sliding

The heap sliding may be skipped if it is acceptable that the memory freed is at the bottom of the heap rather than the top of the heap.

For an operating system where the memory management unit may be somewhat controlled by the user, it might be possible to return arbitrary sections of the memory. In that case, the heap sliding might not always be necessary.

An alternative approach does not rely on the co-operation of the operating system. Instead the heap is allocated as a linked list of numbered blocks. Blocks may be allocated at the top of the heap and deallocated at the bottom of the heap, thereby eliminating the need to slide the heap.

The test for telling if a pointer “points up” however becomes somewhat more complicated. An address comparison is only sufficient if two pointers belong to the same block, otherwise the numbers of the blocks must be compared.

For this to be practical there must exist a fast method that given a pointer find the number of the block that it points to. One standard way to accomplish this is to allocate aligned blocks of the same size, e.g. 1 kbyte blocks on even 1 kbyte boundaries. Given a pointer, the beginning of the block is found by clearing the low ten bits ($1\text{ kbyte}=2^{10}\text{ byte}$) of the pointer.

4.6 Other Data Objects

In the code above only three kinds of objects were handled: atoms, lists and tuples. Most other objects may be catered for by only changing `pointer_length()` and `object_length()`. For instance, a short integer is stored almost like an atom. Similarly, a pointer to a more static area should also be treated like the `id` field of the atom cell, i.e. it should be unchanged by garbage collection.

The algorithm is very easy to extend to handle almost all sorts of objects as all pointers to the object, as well as the object itself of course, are available just before the object is moved.

Binary Data on the Heap In the algorithm it is assumed that all cells are tagged, but this restriction can easily be removed. E.g. a binary object usually contains a tagged word with length information in the first cell. When the heap is scanned, the binary object is simply moved. However, the heap sliding operation goes from low to high addresses, and at the end of the object no distinguishing tag might be present.

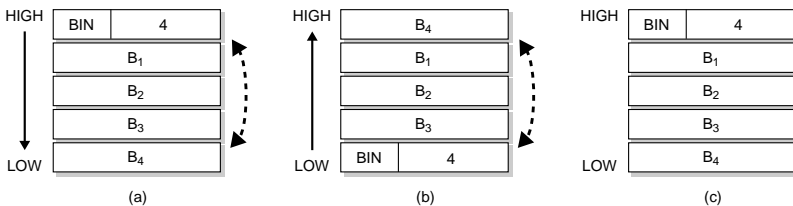


Fig. 9. Moving binary header while scanning in both directions

This problem is solved by swapping cells: In `collect_heap` after having moved the binary object, the first and last cells are swapped so the length information is stored at the end of the object. Then, during heap sliding, the length information is read (so that the objects is not scanned for pointers to updated), the object is moved, and the cells are swapped back again.

Overlapping Objects The algorithm actually accepts that pointers may refer to parts of objects or even overlapping objects. Figure 10 shows a situation that may not occur in ordinary Erlang execution, but that is nevertheless handled correctly by the algorithm.

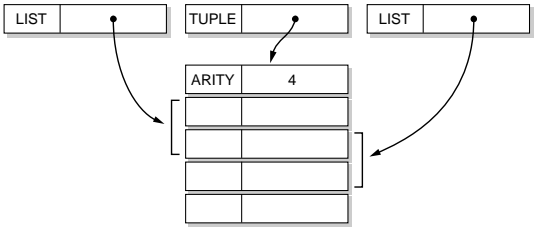


Fig. 10. Overlapping objects

Non Overlapping Objects In Erlang, objects are not overlapping and all references to an object are to its first cell. Thus, when it is possible to calculate the size of the object from the first cell (e.g. a tuple), a non-reachable object when encountered may be skipped entirely. This optimization is particularly important when the heap contains many large objects that are no longer reachable.

Objects with the Header Towards the Bottom of the Heap In the above presentation it has been assumed that a pointer to an object points to the part of the object closest to the top the heap. For an implementation based on copying garbage collection the arrangement is probably the opposite: the head of the object is towards the bottom of the heap.

The algorithm can be changed to cater also for objects of that kind. When the marked head of the object is encountered (`points_up(from)` is true), the `from` pointer is reset to a position higher up on the heap where the object actually started and the `fields` value is set to the length of the object. Thus the whole object will then become copied.

There is a slight performance penalty when objects are stored in this way during garbage collection: To find the size of an object the relocation chain starting at the head of the object must first be searched. Only then can we know where the object will be relocated, and the relocation chain has to be traversed again to do the relocation.

It does not seem possible to extend the algorithm to handle overlapping objects, or to skip over unreachable objects when the header is located towards the bottom of the heap.

5 Related Work and Discussion

Already in 1974, David Fisher published a paper on garbage collection for non-cyclic, i.e. unidirectional, list structures[6]. Mark bits were used for reachable data, and the algorithm needed three passes, two of which swept all memory cells.

One main source of inspiration for us was [4] where the advantages of a unidirectional heap were first exploited, implementing the heap as a linked list of numbered objects. That method has excellent real-time properties as the garbage collection may be stopped and resumed at any time during a computation. However, there is an up to 50% overhead in memory consumption due to the storage of the “history counters”. Maintaining the free list of objects during execution and updating the “history counter” for each cell is also probably quite time consuming.

Pointer reversal techniques have successfully been used for marking of live data [1] and for compacting the heap [8]. By utilizing the unidirectionality of the heap we found that a single sweep through the heap was sufficient to both mark, compact and relocate all data. This phase sometimes has to be followed by a heap sliding to relocate the data towards the beginning of the heap. It seems unusual to exploit the unidirectionality of the heap in garbage collection. In a survey [10], no such techniques were mentioned.

Traditionally garbage collection based on copying has been used in Erlang implementations [9]. Copying garbage collectors only traverse live data, whereas the proposed algorithm may have to scan over all data once.

Still there are some advantages with our proposed algorithm. The historical order of the data is preserved and that often gives good locality of the data. It is easy to implement generational garbage collection that may be good enough for the real-time requirements. Last, and perhaps most important, no extra memory is used so fragmentation of the main memory is reduced.

6 Acknowledgments

Our colleagues at Ericsson have given us many helpful comments. The constructive comments by the referees have made us focus on a number of weak points in our presentation.

References

- [1] Karen Appleby, Mats Carlsson, Seif Haridi, and Dan Sahlin. Garbage collection for Prolog based on WAM. *CACM*, pages 719–741, 1988.

- [2] J. L. Armstrong, B. O. Däcker, S. R. Virding, and M. C. Williams. Implementing a functional language for highly parallel real time applications. In *SETSS 92*, 1992.
- [3] Joe Armstrong. Erlang – a survey of the language and its industrial applications. In *INAP'96 – The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, October 1996. Hino, Tokyo, Japan.
- [4] Joe Armstrong and Robert Virding. One pass real-time generational mark-sweep garbage collection. In *International Workshop on Memory Management 1995*, 1995.
- [5] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1996.
- [6] David A. Fisher. Bounded workspace garbage collection in an address-order preserving list processing environment. *Info. Proc. Letters*, 3(1), July 1974.
- [7] H. Lieberman and C. Hewitt. A real time garbage collector based on the life time of objects. *CACM*, 26(6):419–429, 1983.
- [8] F.L. Morris. A time and space efficient garbage compaction algorithm. *CACM* 21, 21(8), 1978.
- [9] Robert Virding. A garbage collector for the concurrent real-time language Erlang. In Henry G. Baker, editor, *International Workshop on Memory Management 1995*, number 986 in Lecture Notes in Computer Science. Springer, September 1995. ISBN 3-540-60368-9.
- [10] Paul R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management 1992*, volume 637. Springer-Verlag, September 1992. A much expanded version of the paper is available from <http://www.cs.utexas.edu/users/oops/papers.html>.