Zoltán Horváth Rinus Plasmeijer
Anna Soós Viktória Zsók (Eds.)

Central European Functional Programming School

Second Summer School, CEFP 2007
Cluj-Napoca, Romania, June 23-30, 2007
Revised Selected Lectures

Springer

Volume Editors

Zoltán Horváth
Viktória Zsók
Eötvös Loránd University, Faculty of Informatics
1117 Budapest, Hungary
E-mail: {hz, zsv}@inf.elte.hu

Rinus Plasmeijer
University of Nijmegen, Faculty of Science
6500 GL Nijmegen, The Netherlands
E-mail: rinus@cs.ru.nl

Anna Soós
Babeş-Bolyai University
400084 Cluj-Napoca, Romania
E-mail: asoos@math.ubbcluj.ro

Library of Congress Control Number: 2008935561

CR Subject Classification (1998): D.1.1, D.3.2, F.3.3, D.2, D.1.5

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN	0302-9743
ISBN-10	3-540-88058-5 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-88058-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12525737 06/3180 5 4 3 2 1 0

Preface

This volume presents the revised lecture notes of selected talks given at the second Central European Functional Programming School, CEFP 2007, held June 23–30, 2007 at Babeş-Bolyai University, Cluj-Napoca, Romania.

The summer school was organized in the spirit of the advanced programming schools. CEFP focuses on involving an ever-growing number of students, researchers, and teachers from central, and eastern European countries. We were glad to welcome the invited lecturers and the participants: 15 professors and 30 students from 9 different universities. The intensive program offered a creative and inspiring environment and a great opportunity to present and exchange ideas in new topics of functional programming.

The lectures covered a wide range of topics like interactive work flows for the Web, proving properties of lazy functional programs, lambda calculus and abstract lambda calculus machines, programming in Ωmega, object-oriented functional programming, and refactoring in Erlang.

We are very grateful to the lecturers and researchers for the time and the effort they devoted to the talks and the revised lecture notes. The lecture notes were each carefully checked by reviewers selected from experts of functional programming. Afterwards the papers were revised once more by the lecturers. This revision process guaranteed that only high-quality papers are accepted in the volume of the lecture notes.

The PhD students were provided with a workshop, held in conjunction with the summer school. The workshop was an ideal opportunity to exchange ideas and get feedback from the lecturers about their research work. The reviewers decided to include the best papers in the revised volume of the summer school. Finally, the paper of Jan Martin Jansen was chosen as the student paper out of six presentations.

We would like to thank the work of all the members of the Program Committee and the Organizing Committee.

The web page for the summer school can be found at http://cs.ubbcluj.ro/cefp2007/.

June 2008

Zoltán Horváth
Rinus Plasmeijer
Anna Soós
Viktória Zsók

Organization

CEFP 2007 was organized by the Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania.

Executive Committee

Program Chair	Zoltán Horváth (Eötvös Loránd University, Hungary)
Organizing Chair	Anna Soós (Babeş-Bolyai University, Romania)
Organizing Committee	Zoltán Csörnyei (Eötvös Loránd University, Hungary) Zalán Bodó Lehel Csató Zsolt Minier Horia F. Pop (Babeş-Bolyai University, Romania)

Program Committee

Kevin Hammond	University of St. Andrews, UK
Zoltán Horváth	Eötvös Loránd University, Hungary
Rinus Plasmeijer	Radboud University Nijmegen, The Netherlands
Horia F. Pop	Babeş-Bolyai University, Romania
Anna Soós	Babeş-Bolyai University, Romania
Viktória Zsók	Eötvös Loránd University, Hungary

Sponsoring Institutions

The summer school was supported by the CEEPUS program (via the CEEPUS CII-HU-19 Network) and by the Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca.

Table of Contents

An Introduction to iTasks: Defining Interactive Work Flows for the Web

Rinus Plasmeijer, Peter Achten, and Pieter Koopman

Radboud University Nijmegen, Netherlands
{rinus,P.Achten,pieter}@cs.ru.nl

Abstract. In these lecture notes we present the iTask system: a set of combinators to specify *work flows* in a pure functional language at a very high level of abstraction. Work flow systems are automated systems in which *tasks* are coordinated that have to be executed by either humans or computers. The combinators that we propose support work flow patterns commonly found in commercial work flow systems. In addition, we introduce novel work flow patterns that capture real world requirements, but that can not be dealt with by current systems. Compared with most of these commercial systems, the iTask system offers several further advantages: tasks are statically typed, tasks can be higher order, the combinators are fully compositional, dynamic and recursive work flows can be specified, and last but not least, the specification is used to generate an executable web-based multi-user work flow application. With the iTask system, useful work flows can be defined which cannot be expressed in other systems: a work can be interrupted *and* subsequently directed to other workers for further processing. The iTask system has been constructed in the programming language Clean, making use of its generic programming facilities, and its iData toolkit with which interactive, thin-client, form-based web applications can be created. In all, iTasks are an excellent case of the expressive power of functional and generic programming.

1 Introduction

Work flow systems are automated systems that coordinate *tasks*. Parts of these tasks need to be performed by humans, other parts by computers. Automation of tasks in this way can increase the quality of the process, as the system keeps track of tasks, who is performing them, and in what order they should be performed. For this reason, there are many commercial work flow systems (such as Business Process Manager, COSA Workflow, FLOWer, i-Flow 6.0, Staffware, Websphere MQ Workflow, and YAWL) that are used in industry. If we investigate contemporary work flow systems from the perspective of a modern functional programming language such as Clean and Haskell, then there are a number of salient features that functional programmers are accustomed to that appear to be missing in work flow systems:

– Work flow situations are typically specified in a graphical language, instead of a textual language as typically used in programming languages. Functional programmers are keen on abstraction using higher order functions, generic programming techniques, rich type systems, and so on. Although experiments have been conducted to express these key features graphically (Vital [11], Eros [7]), functional programs are typically specified textually.
– Work flow systems mainly deal with control flow rather than data flow as in functional languages. As a result, they have focussed less on expressive type systems and analysis as has been done in functional language research.
– Within work flow systems, the data typically is globally known and accessible, and resides in databases. In functional languages, data is passed around between function arguments and results, and is therefore much more localized.

Given the above observations, we have posed the question if, and which, functional programming techniques can contribute to the expressiveness of work flow systems. In these lecture notes we show how web-applications with complex control flows can be constructed by presenting the iTask system: a set of combinators for the specification of interactive multi-user web-based *work flows*. It is built on top of the iData toolkit, and both can be used within the same program. The library covers all known *work flow patterns* that are found in contemporary commercial work flow tools [24]. The iTask toolkit extends these patterns with strong typing, higher-order functions and tasks, lazy evaluation, and a monadic style of programming. Its foundation upon the generic [1, 13] features of the iData toolkit yields compact, robust, reusable and understandable code. Work flows are defined on a very high level of abstraction. It truly is an executable specification, as much is done and generated automatically.

The iData toolkit [18, 19] is a high level library for creating interactive, thin client, web applications. For this reason it is well suited as an implementation platform for iTasks, because work flow systems are typically multi-user applications. As web browsers are ubiquitously available, it makes sense to implement a work flow system with web technology. The iData toolkit is a domain specific language embedded in the pure, lazy functional programming language Clean. In order to validate the expressiveness of the toolkit, a number of non-trivial web applications have been developed, such as a web shop, a project administration system [18], and a conference management system [17]. Based on these case studies, we observe that the iData toolkit is well suited to create complex GUI forms, which can be used to create and change values of complex data types. However, the iData toolkit is less suited for the specification of programs that require explicit *control flows*. To realize a control flow, the application programmer needs to keep track of the current application state by means of data storages. This can lead to programs that are difficult to comprehend and maintain, and it does not scale well.

A small, yet illustrative, exercise to handle work flow situations was given to us by Phil Wadler:

"Suppose that you want two integer forms to appear *one after another* on the screen and *then* show the sum of them, how do you programme this using iData?"

The key idea of an iData program is that it really is a collection of editors. From this point of view, the concept of a 'terminated' editor is not very natural. Instead, the collection of editors stays alive after each edit operation, allowing the user to enter other data as is also common in a spreadsheet. The exercise above illustrates the need to specify the control flow between editors as well. This is technically possible since all editors are created dynamically. However, there is no specific support in the iData library to do this conveniently and in our case studies we have encountered similar situations in which control flows could be defined with iData elements, but in an ad-hoc way. These issues are tackled within the iTask system.

In these lecture notes, we assume that the reader is familiar with the functional programming language Clean [1] that is used in this paper.

The major part of this tutorial is devoted to presenting the iTask toolkit by means of a range of examples that demonstrate its major concepts in Sect. 2. We briefly discuss its implementation in Sect. 3. We end with related work in Sect. 4 and conclusions in Sect. 5. Appendix A gives the complete *api* of the iTask toolkit.

2 Overview of the iTask System

In this section we present the main concepts of the iTasks toolkit by means of a number of examples.

2.1 A Simple Example

With the iTask system, the work flow engineer specifies a work flow situation using combinators. This specification is interpreted by the iTask system. It presents to the work flow user a web browser interface that implements the given task. As a starter, we give the complete code of an extremely simple work flow, viz. that of a single, elemental, task in which the user is requested to fill in an integer form (see also Fig. 1):

```
module example

import StdEnv, iTasks

Start :: *World → *World
Start world = doHtmlServer (singleUserTask 0 True simple) world

simple :: Task Int
simple = editTask "Done" createDefault
```

[1] See http://www.st.cs.ru.nl/papers/2007/CleanHaskellQuickGuide.pdf for the main differences between Clean and Haskell.

In line 3, the necessary modules are imported. `StdEnv` contains the standard functions, data structures, and type classes of Clean. `iTasks` imports the iTask system. The expression to be reduced as the main function is always given by the `Start` function. Because it has an effect on the external world, it is a function of type `*World` → `*World`. In Clean, effects on an environment of some type `T` are usually modeled with environment transformer functions of type (... `*T` → (... ,`*T`)). The *uniqueness attribute* `*` indicates that the environment is to be passed along in a single threaded way. This effect is similar to using the `IO` monad in Haskell, but uniquely attributed states are passed around explicitly. Violations against single threading are captured by the type system. In the iTask toolkit, tasks that produce values of some type `a` have type `Task a`:

```
:: Task a :== *TSt → (a,*TSt)
```

Here, `*TSt` is the unique and opaque environment that is passed along all tasks.

The iTasks library function `doHtmlServer` is a wrapper function that takes a function that generates an HTML page, and turns it into a real Clean application. The library function `singleUserTask` takes a work flow specification (here `simple`), provides it with a single user infrastructure, and computes the corresponding HTML page that reflects the current state of the work flow system. In Sect. 2.7 we encounter the `multiUserTask` function that dresses up multi-user work flow specifications. The infrastructure is a tracing option at the top of the window. It displays for each user her main tasks in a column. The selected main task is displayed next to this column.

The example work flow is given by `simple` (lines 8–9). It creates a single task with the library function `editTask` which has the following type:

```
editTask :: String a[2] → Task a |[3] iData a
```

Its first argument is the label of the push button that the user can press to tell the system that this task is finished. Its second argument is the initial value that the task will display. When the user is done editing, hence after pressing the push button, the edited value is emitted by `editTask`. The type of `editTask` is overloaded. The type class `iData` collects all generic functions that are required for the iTask library to derive the proper instances.

```
class iData            d | gForm {|*|}, iCreateAndPrint, gParse{|*|}, gerda {|*|}, TC d
class iCreateAndPrint d | iCreate, iPrint d
class iCreate          d | gUpd  {|*|}         d
class iPrint           d | gPrint{|*|}         d
```

They can be used for values of *any* type to automatically create an HTML form (`gForm`), to handle the effect of any edit action with the browser including the creation of default values (`gUpd`), to print or serialize any value (`gPrint`), to

[2] Note that in Clean the arity of functions is denoted explicitly by white-space between the arguments, hence the arity of `editTask` is two.

[3] Type class restrictions always occur at the end of a type signature, after a | symbol. The equivalent Haskell definition reads `editTask :: (iData a) => String -> a -> Task a`.

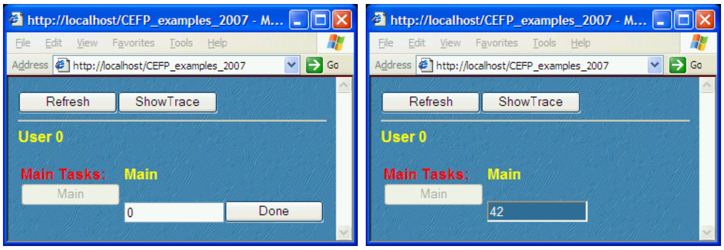

http://localhost/CEFP_examples_2007 - M...
File Edit View Favorites Tools Help
Address http://localhost/CEFP_examples_2007
Go
Refresh
ShowTrace
User 0
Main Tasks:
Main
Main
0
Done
http://localhost/CEFP_examples_2007 - M...
File Edit View Favorites Tools Help
Address http://localhost/CEFP_examples_2007
Go
Refresh
ShowTrace
User 0
Main Tasks:
Main
Main
42

Fig. 1. An elemental `Int` iTask when started (left) and finished (right)

parse or de-serialize any value (`gParse`), to store, retrieve or update any value in a relational database (`gerda`), or to serialize and de-serialize values and functions in a `Dynamic` (using the compiler generated `TC` class).

Note that the type of `simple` is more restrictive than that of `editTask`. This is because it uses the `createDefault` function which has signature:

```
createDefault :: d | gUpd{|*|} d
```

This function can generate a value for any type for which an instance of the generic `gUpd` function has been derived. Consequently, the most general type of `simple` is:

```
simple :: Task a | iData a
```

which is an overloaded type. Using this type makes the type of `Start` also overloaded, which is not allowed in Clean. There are basically two ways to deal with this: the first way is to replace `createDefault` with a concrete integer value, say 0:

```
simple = editTask "Done" 0
```

In that case, its type is `:: Task Int`. However, this is not very flexible: `simple` is now restricted to being an integer editing task. The second way, which was used in the original solution, is much more general: by only modifying the type signature of `simple`, but not its implementation, we can alter its editing task.

In the remainder of this tutorial, we skip the first three overhead lines of the examples, and show only the `Start` function.

Exercises

1. *Getting started*
Download Clean for free at

  `http://clean.cs.ru.nl/`.

Install the Clean system. Also download the iTask system, which is available at

  `http://www.cs.ru.nl/~rinus/iTaskIntro.html`.

Follow the installation instructions *"iTasks - Do Read This Read Me.doc"* file that can be found in the `iTasks Examples` folder.

When done, start the Clean IDE. Create a new Clean implementation module, named *"exercise1.icl"*, and save it in a new directory of your choice. Create a new project, and confirm the suggested name and location by the Clean IDE (i.e. *"exercise1.prj"* in the newly created directory). Set the Environment to *"iTasks and iData and Util"*; otherwise the Clean compiler will complain about a plethora of missing files. Create, within the newly created directory, a subdirectory with the *same name*, and copy the file *"back35.jpg"* into it. This file can be found in any of the `Examples\iTasks Examples\` example directories of the iTask system. Use for each of the exercises a separate directory, to allow the system to create databases in such a way that they do not cause conflicts of name and type.

Enter in *"exercise1.icl"* the complete code that has been displayed in Sect. 2.1. Compile and run the application. If everything has gone well, you should see a console window that asks you to open your favorite browser and direct it to the given address. Follow this instruction, and you should be presented with your first iTask application that should be similar to Fig. 1.

2.2 Playing with Types

In this example we exploit the general purpose code of the previous example. The only modification we make is in line 8:

```
simple :: Task (Int,Real)                                    8.
```

Compiling and running this example results in a simple task for filling in a form of a pair of an `Int` and `Real` input field (see Fig. 2).

Now suppose that we want to do the same for a simple person administration form: we introduce a suitable record type, `Person`, defined as:

```
:: Person = { firstName   :: String,  surname :: String
            , dateOfBirth :: HtmlDate, gender :: Gender }
:: Gender = Male | Female
```

`HtmlDate` is a predefined algebraic data type for which an editor is created that allows the user to manipulate dates with separate editors for the year, month, and day. The only thing we need to do is to change the signature of `simple` into:

```
simple :: Task Person                                        8.
```

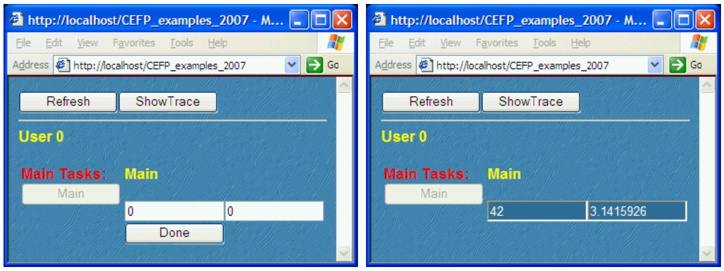
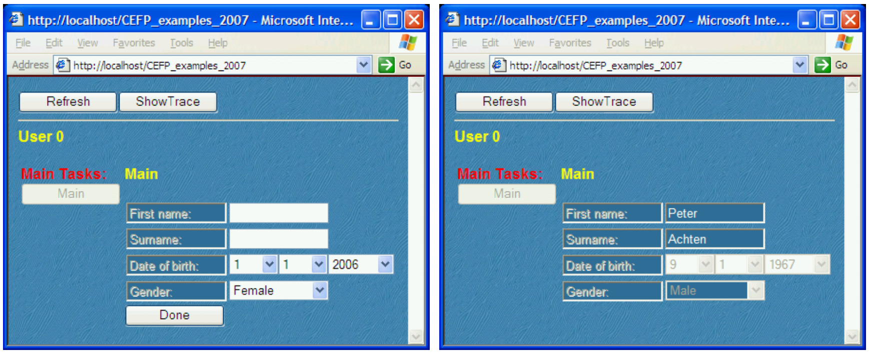

http://localhost/CEFP_examples_2007 - M...
File Edit View Favorites Tools Help
Address http://localhost/CEFP_examples_2007 Go
Refresh
ShowTrace
User 0
Main Tasks:
Main
Main
0
0
Done
http://localhost/CEFP_examples_2007 - M...
File Edit View Favorites Tools Help
Address http://localhost/CEFP_examples_2007 Go
Refresh
ShowTrace
User 0
Main Tasks:
Main
Main
42
3.1415926

Fig. 2. An (`Int`,`Real`) iTask when started (left) and finished (right)


http://localhost/CEFP_examples_2007 - Microsoft Inte...
File Edit View Favorites Tools Help
Address http://localhost/CEFP_examples_2007
Go
Refresh
ShowTrace
User 0
Main Tasks:
Main
Main
Done

First name:			
Surname:			
Date of birth:	1	1	2006
Gender:	Female		

First name:	Peter		
Surname:	Achten		
Date of birth:	9	1	1967
Gender:	Male		

Fig. 3. A Person iTask when started (left) and finished (right)

We intend to obtain an application such as the one displayed in Fig. 3.

Unfortunately, this does not compile successfully. A range of error messages is generated that complain that there are no instances of type `Person` for the generic functions that belong to the `iData` class. The reason that the (`Int`,`Real`) example does compile, and the `Person` example does not, is that for all basic types and basic type constructors such as (,), instances for these generic functions have already been asked to be derived. To allow this for `Person` and `Gender` values as well, we only need to be polite and ask for them:

```
derive gForm  Person, Gender
derive gUpd   Person, Gender
derive gPrint Person, Gender
derive gParse Person, Gender
derive gerda  Person, Gender
```

This example demonstrates that the code is very general purpose, and can be customized by introducing the desired type definitions, and politely asking the generic system to derive instance functions for the new types.

Exercises

2. *Playing with a type of your own*
Create a new directory and subdirectory with the same name. Copy the *"exercise1.icl"* file into the new directory, and rename it to *"exercise2.icl"*. Copy the *"back35.jpg"* file into the subdirectory. Within the Clean IDE, open *"exercise2.icl"* and create a new project. Set the Environment to *"iTasks and iData and Util"*.

Define a new (set of) type(s), such as the `Person` and `Gender` given in Sect. 2.2, and create a `simple` editing task for it.

2.3 Playing with Attributes

In the previous examples an extremely simple, single-user, work flow was created. Even for such simple systems, we need to decide were to store the state of the application, and whether it should respond to every user editing action or only after an explicit *submit* action of the user. These aspects are *attributes* of tasks, and they can be set with the overloaded infix operator `<<@`:

```
class    (<<@) infixl 3 b :: (Task a) b → Task a
instance <<@  Lifespan       // default: Session
           ,  Mode           // default: Edit
           ,  GarbageCollect // default: Collect
           ,  StorageFormat  // default: PlainString

:: Lifespan       = Session | Page    | Database | TxtFile | TxtFileRO | Temp
:: Mode           = Edit    | Submit | Display  | NoForm
:: GarbageCollect = Collect | NoCollect
:: StorageFormat  = PlainString | StaticDynamic
```

The `Lifespan` attribute controls the storage of the value of the iTasks: it can be stored persistently on the server side on disk in a relational database (`Database`) or in a file (`TxtFile` with `RO` read-only), it can be stored locally at the client side in the web page (`Session`, `Page` (default)), or one can decide not to store it at all (`Temp`). Storage and retrieval of data is done automatically by the system. The `Mode` attribute controls the *rendering* of the iTask: by default it can be `Edit`ed which means that every change made in the form is communicated to the server, one can choose for the more traditional handling of forms where local changes can be made that are all communicated when the `Submit` button is pressed, but it can also be `Display`ed as a constant, or it is not rendered at all (`NoForm`). The `GarbageCollect` attribute controls whether the task tree should be garbage collected. This issue is described in more detail in Sect. 3.6. Finally, the `StorageFormat` attribute determines the way data is stored: either as a string (`PlainString`) or as a dynamic (`StaticDynamic`).

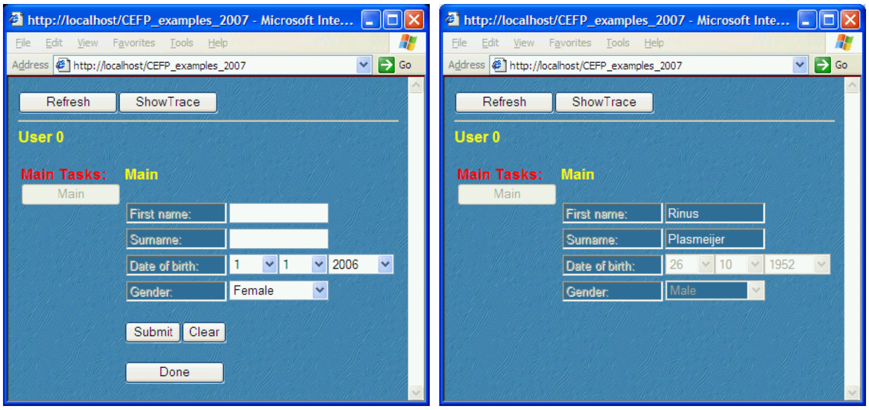
As an example, consider attributing the `simple` function of Sect. 2.1 in the following way (see Fig. 4):

```
simple :: Task Person                                                    8.
simple = editTask "Done" createDefault <<@ Submit <<@ TxtFile            9.
```

Exercises

3. *A persistent type of your own*
Create a new project for *"exercise3.icl"* as instructed in exercise 2.

Modify the code in such a way that it creates an application in which the most recently entered data is displayed, regardless whether the browser has been closed or not.


http://localhost/CEFP_examples_2007 - Microsoft Inte...
File Edit View Favorites Tools Help
Address http://localhost/CEFP_examples_2007 Go
Refresh ShowTrace
User 0
Main Tasks: Main
Main
Submit Clear
Done

First name:	
Surname:	
Date of birth:	1 / 1 / 2006
Gender:	Female

First name:	Rinus
Surname:	Plasmeijer
Date of birth:	26 / 10 / 1952
Gender:	Male

Fig. 4. A `Person` iTask attributed to be a 'classic' form editor

With these attributes, the application only responds to user actions after she has pressed the "Submit" button, and the value is stored in a text based database.

2.4 Sequencing with Monads: Wadler's Exercise

In the previous examples, the work flow consisted of a single task. One obvious combination of work flows is *sequential composition*. This has been realized within the iTask toolkit by providing it with appropriate instances of the *monadic* combinator functions:

```
(=>>) infix  1 :: (Task a) (a → Task b) → Task b | iCreateAndPrint b
(#>>) infixl 1 :: (Task a)     (Task b) → Task b
return_V       :: b                     → Task b | iCreateAndPrint b
```

where $=\!\gg$ is the *bind* combinator, and `return_V` the *return* combinator. Hence, ($m =\!\gg \lambda x \rightarrow n$) performs task m if it should be activated, and passes its result value to n, which is only activated when required. The only task of (`return_V` v) is to emit value v. As usual, the shorthand combinator $\#\!\gg$ that is defined immediately in terms of $=\!\gg$ ($m \;\#\!\gg n \equiv m =\!\gg \lambda\ \_ \rightarrow n$) is provided as well. It is convenient to have a few alternative *return*-like combinators:

```
return_VF :: b [BodyTag] → Task b | iCreateAndPrint b
return_D  :: b           → Task b | iCreateAndPrint, gForm{|*|} b
```

With (`return_VF` v *info*), customized information *info* given as HTML is shown to the application user. The algebraic type `BodyTag` maps one-to-one to the HTML-grammar. With (`return_D` v) the standard generic output of v is used instead. It should be noted that unlike `return_V` these combinators are not true *return* combinators, as they do have an effect. Hence, the monad law $m =\!\gg \lambda v \rightarrow \mathit{return}\ v = m$ is invalid when *return* is constructed with either `return_VF` or `return_D`.

When a task is in progress, it is useful to provide feedback to the user what she is supposed to be doing. For this purpose two combinators are introduced.

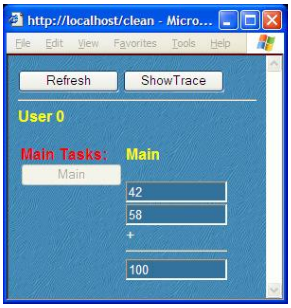
(p ?>> t) is a task that displays prompt p while task t is running, whereas (p !>> t) displays prompt p from the moment task t is activated. Hence, a message displayed with !>> stays displayed once it has appeared, and a message displayed with ?>> disappears as soon as its argument task has finished.

```
(?>>) infix 5 :: [BodyTag] (Task a) → Task a | iCreate a
(!>>) infix 5 :: [BodyTag] (Task a) → Task a | iCreate a
```

The prompt is defined as a piece of HTML.

With these definitions, the solution to Wadler's exercise becomes surprisingly simple.

```
sequenceITask :: Task a | iData, + a
sequenceITask
= editTask "Done" createDefault =>> λv1 →
  editTask "Done" createDefault =>> λv2 →
  [Txt "+",Hr []]
  !>> return_D (v1+v2)
```


http://localhost/clean - Micro...
File Edit View Favorites Tools Help
Refresh
ShowTrace
User 0
Main Tasks:
Main
Main
42
58
+
100

Exercises

4. *Hello!*
Create a work flow that first asks the name of a user, and then replies with "Hello" and the name of the user.

5. *To !>> or to ?>>*
Create a new project with the code of sequenceITask, and modify the !>> combinator into ?>>. What is the difference with the !>> combinator?

6. *Enter a prime number*
Create a work flow that uses the <| combinator (see Appendix A) to force the user to enter a prime number. A prime number p is a positive integral number that can be divided only by 1 and p.

7. *Tearing Person apart*
In Sect. 2.2, a Person editing task was created with which the user edits complete Person values. Create a new work flow in which the user has to enter values for the fields one by one, i.e. starting with first name, and subsequently asking the surname, date of birth, and gender. Finally, the work flow should return the corresponding Person value.

8. *Adding numbers*
Create a work flow that first asks the user a positive (but not too great) integer number n, and subsequently have him enter n values of type Real (use the seqTasks

combinator for this purpose – see Appendix A). When done, the work flow should display the sum of these values.

2.5 Sequence and Choice: A Single Step Coffee Machine

Coffee vending machines are popular examples to illustrate sequencing and choice. We present an example of a coffee machine that offers the user either coffee or tea. After choosing, the user pays the proper amount of money and obtains the selected product. This also terminates the coffee machine. This is a single user task. The `Start` function is standard:

```
Start world = doHtmlServer (singleUserTask 0 True coffeemachine) world
```

The coffee machine is specified by the function `coffeemachine`. Before we give its definition, we first introduce a number of functions. In Clean, `String`s are arrays of unboxed `Char`s. For convenient `String` concatenation, the overloaded operators (x+>str) and (str<+x) are used which concatenate the string representation of x and str. Two iTask combinators will be used in `coffeemachine`:

```
buttonTask ::   String (Task a)  → Task a | iCreateAndPrint a
chooseTask :: [(String, Task a)] → Task a | iCreateAndPrint a
```

(`buttonTask` l t) enhances a task t with a push button labeled with l that needs to be pressed first by the user before she can do t. Choosing between alternatives of labeled actions l_i and tasks t_i is given by (`chooseTask` $[(l_0,t_0)\ldots(l_n,t_n)]$). The resulting value is the value of the selected task t_i. The choice buttons are aligned horizontally.

We are now ready to give the definition of `coffeemachine`:

```
coffeemachine :: Task (String,Int)                                        1.
coffeemachine                                                             2.
= [Txt "Choose product:"]                                                 3.
  ?>> chooseTask [(p <+ ": " <+ c, return_V prod) \\ prod=:(p,c) ← products]  4.
  =>> λprod →                                                             5.
  [Txt ("Chosen product: " <+ fst prod)]                                  6.
  ?>> pay prod (buttonTask "Thanks" (return_V prod))                      7.
where                                                                     8.
  products     = [("Coffee",100),("Tea",50)]                              9.
  pay (p,c) t = buttonTask ("Pay " <+ c <+ " cents") t                   10.
```

First, the user is presented with a choice between coffee and tea (lines 3-4). Having chosen a product, the user is supposed to pay in a single step (line 7). In Sect. 2.6, we extend this to specifying a sub work flow for inserting coins in the coffee machine.

Besides `chooseTask`, the iTask toolkit offers a number of related task selection combinators:

```
chooseTaskV     :: [(String,Task a)] → Task  a  | iCreateAndPrint a
chooseTask_pdm  :: [(String,Task a)] → Task  a  | iCreateAndPrint a
mchoiceTasks    :: [(String,Task a)] → Task [a] | iCreateAndPrint a
```

`chooseTaskV` is the same as `chooseTask`, except that the choice buttons are aligned vertically. The same holds for `chooseTask_pdm`, except that it offers a pull down menu to select the desired task. Finally, a multiple choice of tasks is provided with `mchoiceTasks`.

Exercises

9. *Calculating on numbers*
In this exercise you extend the work flow in exercise 8 with the option to *add* (+), *subtract* (0), *multiply* (*), or *divide* (/) all numbers. Hence, if the input consists of numbers $x_1 \dots x_n$, and the operator $\odot$, then the result should be computed as $(\dots(x_1 \odot x_2) \odot \dots x_{n-1}) \odot x_n$.

2.6 Repetition, Recursion and State: A Coffee Machine

The coffee machine in the previous example offers a single beverage, and terminates. In order to get more profit out of this machine, we extend it to a beverage vending machine that runs forever with the `foreverTask` combinator:

```
Start world = doHtmlServer (singleUserTask 0 True (foreverTask coffeemachine)) world
```

The signature of `foreverTask` is not surprising:

```
foreverTask :: (Task a) → Task a | iData a
```

It repeats its argument task infinitely many times.

The previous example abstracted from the paying task: the function call (`pay` (p, c) t) offers a labeled action to pay the full amount of money c for the chosen product p, and then continues with task t. In a more refined model, the user is able to insert coins until the inserted amount of money exceeds the cost of the product. Moreover, she can also choose to abandon the paying task and not get the selected beverage at all. This is suitably modeled with a recursive task specification:

```
getCoins :: ((Bool,Int,Int) → Task (Bool,Int,Int))
getCoins = repeatTask_Std get (λ(cancel,cost,_) → cancel || cost ≤ 0)
where
  get (cancel,cost,paid)
        = newTask "pay" (
              [Txt ("To pay: " <+ cost)]
              ?>> chooseTask [(c +> " cents", return_V (False,c)) \\ c ← coins ]
                -||-
              buttonTask "Cancel" (return_V (True,0)) =>> λ(cancel,c) →
              return_V (cancel,cost-c,paid+c)
            )
  coins  = [5,10,20,50,100,200]
```

The iteration of inserting coins is modeled with the `repeatTask_Std` combinator:

```
repeatTask_Std :: (a → Task a) (a → Bool) a → Task a | iCreateAndPrint a
```

(`repeatTask_Std` $t\ p\ v_0$) executes a sequence of tasks $t\ v_0, t\ v_1, \ldots t\ v_n$ along a progressing sequence of values $v_0, v_1, \ldots v_n$. Here, v_i is the result value of task $(t\ v_{i-1})$. The final result value, v_n, is also the result value of (`repeatTask_Std` $t\ p\ v_0$). For each $i < n$, we have $\neg(p\ v_i)$, and $(p\ v_n)$. Hence, it works in a way similar to a *repeat t until p* control structure in imperative languages. The combinator `-||-` allows evaluation of two tasks in any order, and is finished as soon as either one task is finished. This is different from the behaviour of the task selection combinators that were discussed above in Sect. 2.5: they allow the user to select one task, which is then evaluated to the end. A similar combinator to `-||-` is `-&&-` which allows evaluation of two tasks in any order, but that finishes only if both tasks have finished.

The crucial combinator in this example is `newTask` (the implementation of `newTask` is discussed in Sect. 3.6). (`newTask` $l\ t$) promotes any user defined task t to a proper iTask such that t is only called when it is its turn to be activated. This is to prevent unwanted non-termination: although a *task description* is allowed to be defined recursively, at any stage of its execution, a workflow system is in some well defined state. Clearly, we regard `getCoins` not as a common recursive function, but as a definition of a recursive task that has to be activated when the previous task, which might be the previous invocation of `getCoins`, is finished.

We can now redefine the `pay` function of Sect. 2.5:

```
pay (p,c) t = getCoins (False,c,0) =>> λ(cancel,_,paid) →
               [Txt ("Product = "<+if cancel "cancelled" p
                                  <+". Returned money = "<+(paid-c))]
               ?>> t
```

It should be noted that `getCoins` and `pay` illustrate that tasks may depend on the actual values that are generated within the system. These kind of workflows are hard to model with other current day work flow specification tools.

Exercises

10. *A mini calculator*
Create a work flow that repeatedly offers the user the choice between:

– *First* enter a `Real` number r and *next* choose an operator $\odot$ (as in exercise 9) and that returns $c \odot r$, with c the current value; $c \odot r$ becomes the new current value.
– Return the current value c.

2.7 Multi-user Workflows

The solution to Phil Wadler's exercise that was given in Sect. 2.4, was a *single user* application. Work flow systems usually involve arbitrarily many users. This is supported by the iTask system.

```
multiUserTask :: !Int !Bool !(Task a) !*HSt → (Html,*HSt) | iCreate a
:: UserID :== Int
```

We identify users (using type synonym `UserID`) with integer index values $i \geq 0$. The wrapper function `multiUserTask` *n trace t* creates a work flow system, defined by t for users $0 \ldots n-1$. For quick testing, it provides an additional user interface for selecting the proper user.

By default, tasks store their information on the client side of the HTML interface. If one wants to use the system with multiple users over the net, one has to store iTask information persistently on the server side. To conveniently control this, we use the attribute setting operator `<<@` that was introduced in Sect. 2.3.

Assigning a task t to user i with some motivation m is done by (m,i)`@:`t. If there is no motivation, then one uses i`@::`t.

```
(@:)  infix 3 :: (String,UserID) (Task a) → Task a | iCreate a
(@::) infix 3 ::        UserID  (Task a) → Task a | iCreate a
```

Suppose that the first integer editing task in Wadler's exercise should be performed by user 1, the second by user 2, and the result is shown to user 0 (the default user). The code becomes:

```
sequenceMU :: Task a | iData, +, zero a
sequenceMU
=   ("Enter a number",1) @: editTask "Done" zero =>> λv1 →
    ("Enter a number",2) @: editTask "Done" zero =>> λv2 →
    [Txt "+",Hr []] !>> return_D (v1 + v2)

Start world = doHtmlServer (multiUserTask 2 True sequenceMU <<@ Persistent) world
```

The iTask system ensures that each user sees only tasks assigned to them. This is essentially a *filter* of the full task tree, because any task may decide to assign tasks to any other user. It should be noted that users have access to data only via the editor tasks. Because every task is always assigned to exactly one user, there is no danger of having multiple users attempting to update the same data item.

Exercises

11. *orTasks versus andTasks*
Create a work flow that first asks the user to enter a positive integral value n, and that subsequently creates n tasks with `orTasks` and `andTasks`. The tasks are simple `buttonTasks`. Study the different behavior of `orTasks` and `andTasks`.

12. *Number guessing*
Create a 2-person work flow in which person 1 enters an integer value $1 \leq N \leq 100$, and who has person 2 guess this number. At every guess, the work flow should give feedback to person 2 whether the number guessed is too low, too high, or just right. In the latter case, the work flow returns `Just`N. Person 2 can also give up, in which case the work flow should return `Nothing`.
Optional: Person 1 is given the result of person 2, and has a chance to respond with a 'personal' message.

13. *Tic-tac-toe*
Create a 2-person work flow for playing the classic 'tic-tac-toe' game. The tic-tac-toe game consists of a 3×3 matrix. Player 1 places $\times$ marks in this matrix, and player 2 places $\circ$ marks. The first person to create a (horizontal, vertical, or diagonal) line of three identical marks wins. The work flow has to ensure that players enter marks only when it is their turn to do so.

2.8 Speculative Tasks and Multiple Users: Deadlines

Work flow systems need to handle time-related tasks: for instance, some task t has to be finished before a given time T or it is canceled. In this example we show how this is expressed with the iTasks toolkit. The time related combinators are the following:

```
waitForDateTask  :: HtmlDate → Task HtmlDate
waitForTimeTask  :: HtmlTime → Task HtmlTime
waitForTimerTask :: HtmlTime → Task HtmlTime
```

The algebraic types `HtmlDate` and `HtmlTime` are elements of the iData toolkit that have been specialized to show user convenient date and time editors. `waitForDate(Time)Task` terminates in case the given date (time of day) has passed; `waitForTimerTask` terminates after a given time interval.

In our example, we use the latter combinator to delegate work:

```
delegateTask who time t                                          1.
=   ("Timed Task",who)@:                                         2.
    @:(  (waitForTimerTask time ♯>> return_V Nothing)            3.
             -||-                                                4.
         ([Txt ("Please finish task within" <+ time)]            5.
          ?>> (t =>> λv → return_V (Just v)))                    6.
      )                                                          7.
```

(`delegateTask` i dt t) assigns a task t to user i that needs to be finished before dt time (line 5–6) is passed. If the user does not complete the task on time, delegation fails, and should also terminate (line 3).

The main work flow situation is modeled as follows:

```
deadline :: (Task a) → Task a | iData a                          1.
deadline t                                                       2.
= [Txt "Choose person you want to delegate work to:"]            3.
```

```
  ?>> editTask "Set" (PullDown size (0,map toString [1..n])) =>> λwho →   4.
  [Txt "How long do you want to wait?"]                                    5.
  ?>> editTask "SetTime" createDefault                        =>> λtime →  6.
  [Txt "Cancel delegated work if you get impatient:"]                      7.
  ?>> delegateTask who time t                                              8.
      -||-                                                                 9.
      buttonTask "Cancel" (return_V Nothing) =>> check                     10.

  check (Just v)                                                           11.
  = [Txt ("Result of task: " <+ v)] ?>> buttonTask "OK" (return_V v)      12.
  check Nothing                                                            13.
  = [Txt "Task expired/canceled; do it yourself!"] ?>> buttonTask "OK" t  14.
```

The main task consists of selecting a user to whom a task t should be delegated (lines 3–4), deciding how much time this user is given for this exercise (lines 5–6), and then delegating the task (line 8). We also model the situation that the current user gets impatient, and decides to abandon the delegated task (line 10). Either way, we know whether the task has succeeded and display the result and terminate (lines 11–12), or the current user has to do it herself (lines 13–14).

The work flow described by (`deadline` t) defines a single delegation. It can be transformed into an iteration with the `foreverTask` combinator that we have also used in Sect. 2.6. We are obviously creating a multi-user system, and hence use the `multiUserTask` wrapper function for some constant $n > 0$. As example task we reuse the `simple` task from Sect. 2.1 with a concrete, non-overloaded type. This finalizes the example:

```
Start world
= doHtmlServer (multiUserTask n True (foreverTask (deadline simple) <<@ Database))
               world
```

Exercises

14. *Delayed task*
Create a work flow in which first an integral value n is asked, and that subsequently waits n seconds before it is finished. Use the `waitForTimerTask` combinator for this purpose.

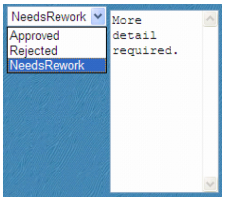
15. *Number guessing with deadline*
Use the delegation example of Sect. 2.8 in such a way that the number guessing game of exercise 12 can be created with it.

16. *Tic-tac-toe with deadline*
Use the delegation example of Sect. 2.8 in such a way that the tic-tac-toe game of exercise 13 can be created with it.

2.9 Parameterized Tasks: A Reviewing Process

In this example we show that iTasks and iData cooperate in close harmony. We present a reviewing process in which the product of a user is judged by a reviewer who can either approve, reject, or demand rework of the product. The latter is described with an algebraic data type:

```
:: Review = Approved
          | Rejected
          | NeedsRework TextArea
```



More detail required.

TextArea is an algebraic data type that is specialized by the iData toolkit as a multi-line text edit box that can be used by the reviewer to enter comments, as shown above.

A reviewer inspects the product *v* that needs to be judged, and makes a decision. This is defined concisely as:

```
review :: a → Task Review | iData a
review v = [toHtml v]
          ?>> chooseTask
              [("Rework",   editTask "Done" (NeedsRework createDefault) <<@ Submit)
              ,("Approved",return_V Approved)
              ,("Reject",   return_V Rejected)
              ]
```

Any task result that can be displayed, can also be subject to reviewing, hence the restriction to the generic iData class. The rendering is done with the iData toolkit function toHtml, which has signature:

```
toHtml :: a → BodyTag | gForm{|*|} a
```

Hence, (review *v*) displays *v* in the browser. The reviewer subsequently has to choose whether *v* should be reworked, and can comment on her decision, or *v* can be approved or rejected.

The main task is to produce a product *v* according to some task *t* that can be judged by a reviewer *u*. If the reviewer demands rework of *v*, the task should be restarted with that particular *v*, because the user would have to completely recreate a new product otherwise. Therefore, the product and the task to produce it are given as a pair (a, a → Task a), and the result of the main task is to return a product and its review (a,Review). This is done as follows:

```
taskToReview :: UserID (a,a → Task a) → Task (a,Review) | iData a      1.
taskToReview reviewer (v,task)                                          2.
= newTask "taskToReview"                                                3.
  ( task v                   =>> λnv →                                  4.
    reviewer @:: review nv =>> λr →                                     5.
    [Txt ("Reviewer " <+ reviewer <+ " says "),toHtml r]               6.
    ?>> buttonTask "OK"                                                 7.
    case r of                                                      8.
       (NeedsRework _) → taskToReview reviewer (nv,task)           9.
       else            → return_V (nv,r)                           10.
  )
```

The task is performed to return a product (line 4), which is reviewed by the given reviewer (line 5). Her decision is reported (line 6), and only in case of a demanded rework, this has to be repeated (line 9).

For the example, we select a two-user system (`multiUserTask` 2) in which user 0 creates the product, and user 1 reviews it:

```
Start world
= doHtmlServer (multiUserTask 2 True (foreverTask reviewtask <<@ TxtFile)) world

reviewtask :: Task (Person,Review)
reviewtask = taskToReview 1 (createDefault, t)

t :: a → Task a | iData a
t v = [Txt "Fill in Form:"] ?>> editTask "TaskDone" v <<@ Submit
```

Note the high degree of parameterization and therefore re-useability of the code: `taskToReview` handles *any* task, and by providing *only* a type signature to `reviewtask` above, we get a form task for values of that type for free. Above, we have chosen the `Person` type. This is similar to the `simple` example that we started with in Sect. 2.1.

2.10 Higher Order Tasks: Shifting Work

A distinctive feature of the iTask system is that tasks can be higher order: data can be communicated but also (partially evaluated) tasks can. One can create task closures, i.e. a task t that already has been partially evaluated by someone can be shipped to some other user as (`TCl` t) who can continue to work on t.

```
:: TCl a        = TCl (Task a)
```

The proper generic functions have been specialized for type `TCl` such that it acts as a container of tasks. Any task can be put in a value of this type, but we want to be able to put a partially evaluated task in it. Therefore we need a way to interrupt a task that is being evaluated.

```
(-!>) infix 4 :: (Task stop) (Task a) → Task (Maybe stop,TCl a)
                 | iCreateAndPrint stop & iCreateAndPrint a
```

(*stop* `-!>` *t*) is a variant of an or-task which takes two tasks: whenever *stop* is done, *t* is interrupted and this possibly partially evaluated task is delivered as result. However, *t* can also finish normally, and the fully completed task is delivered. The result of *stop*, therefore, is only returned when it finishes before *t*. Note that, because `stop` is a type variable, any task can be used as the *stop* task.

As an example of using `-!>`, we present a highly dynamic case in which a worker pool of people can work on a given task. At any time, a worker can decide to stop working on that task, which should then be *continued* to work on by somebody else. Of course, the next person should not restart the task, but work with the partially evaluated task. The code of this example is given by delegate:

```
delegate :: (Task a) HtmlTime → Task a | iData a                                   1.
delegate t time                                                                    2.
=   [Txt "Choose persons you want to delegate work to:"]                           3.
    ?>> determineSet []           =>> λpeople →                                    4.
    delegateToSomeone t people =>> λresult →                                       5.
    return_D result                                                                6.
where                                                                              7.
    delegateToSomeone :: (Task a) [UserID] → Task a | iData a                      8.
    delegateToSomeone t people = newTask "delegateToSet" doDelegate                9.
    where                                                                          10.
        doDelegate                                                                 11.
         =  orTasks [ ( "Waiting for " <+ who                                      12.
                      , who @:: buttonTask "I Will Do It" (return_V who)           13.
                      )                                                            14.
                    \\ who ← people                                                15.
                    ] =>> λwho →                                                   16.
            who @:: stopTask -!> t =>> λ(stopped,TCl t) →                          17.
            if (isJust stopped) (delegateToSomeone t people) t                     18.

        stopTask         = buttonTask "Stop" (return_V True)                       19.
```

The function delegate first creates a worker pool of people to choose from (line 3–4). All people are asked whether they want the task (line 5 and lines 8–18). The first user who accepts the task obtains it and she can work on it. However, the work can be interrupted by completion of stopTask which ends when the user has pushed the Stop button. If this is the case, all persons are asked again to volunteer for the job. The one who accepts, obtains the task in the state as it has been left by the previous worker and she can continue to work on it. The whole recursively defined process finally ends when the delegated task is fully completed by someone.

The conditions for stopping a task can be arbitrarily complex. For instance, by using stop2 not only the user herself can stop the task, but someone else can do it for her as well (e.g. the user who delegated the task in the first place), or it can be timed out.

```
stop2 user time = stopTask -||- (0 @:: stopTask) -||- timer time
timer time      = waitForTimerTask time #>> return_V True
```

Finally, creating the worker pool is a recursive work flow in which the user can select from candidates 1 upto n.

```
determineSet :: [UserID] → Task [UserID]                                           1.
determineSet people = newTask "determineSet" pool                                  2.
where                                                                              3.
    pool   = [Txt ("Current set:" <+ people)]                                      4.
             ?>> chooseTask                                                        5.
```

```
                  [("Add Person", cancelTask person)
                  ,("Finished",   return_V Nothing)
                  ] =>> λresult →
              case result of
                (Just new) → determineSet (sort (removeDup [new:people]))
                Nothing    → return_V people
    person =  editTask "Set" (PullDown size (0,map toString [1..npersons]))
              =>> λwhomPD → return_V (Just (toInt (toString whomPD)))

cancelTask task = task -||- buttonTask "Cancel" (return_V createDefault)
```

Exercises

17. *Number guessing in a group*
In this exercise you extend the number guessing game of exercises 12 and 15 to a fixed set of persons $1 \ldots N$ in which user 0 determines who of $1 \ldots N$ is the next person to try to guess the number.

2.11 Summary

In this section we have given a range of examples to illustrate the expressive power of the iTask toolkit. We have not covered all of the available combinators. They can be found in Appendix A.

3 The iTasks Core System

The examples that have been given in Sect. 2 illustrate that iTask applications are multi-user applications that use mainly forms to communicate with end users, have various options to store data (client side and server side), and are highly dynamic. In general, implementing such kind of web applications is quite a challenge, especially when compared with desktop applications. One reason for this complication is that desktop applications can directly interact with the environment at any point in time because they are directly connected with that environment. Due to the client-server architecture, web applications cannot do this. A web application emits an HTML page and terminates. It has to store information somewhere to handle the next request from the user in an appropriate way. It has to recover the relevant states, find out what it was doing and what it has to do next. The resulting code is hard to understand.

A conceivable alternative is to adopt the Seaside approach [6]. If the application can automatically remember where it was, programs become easier to write and read. Since a Clean application is compiled to native code, suspending execution, as Seaside does, involves creating core dumps of the run-time system. However, a work flow system needs to support several users that work together. The action of one user can influence the work of others. A core dump only reflects the work of one user. For this reason, we propose a simpler set-up of the system: we start the same application from scratch, as we already did, and use iData elements to remember the state for all users. For a programmer, the application still appears to behave as if it continues evaluation after an I/O request from a browser.

In this section we introduce the main implementation principles of the iTasks system. For didactic reasons we restrain ourselves to a strongly simplified iTask *core system*. This core system is single user and has limited possibilities to manipulate tasks. The core system is already sufficient to create the solution to Wadler's exercise that was shown in Sect. 2.4. The full iTask toolkit that has been shown in Sect. 2 is built according to these principles.

3.1 iData as Primitive iTask in the Core System

In this section we describe how to lift iData elements to become iTasks. The iData library function `mkIData` creates an iData element. `mkIData` is an explicit `*HSt` environment transformer function. Its signature is:

```
mkIData :: (InIDataId d) → HStIO d | iData d

:: HStIO d :== *HSt → (Form d,*HSt)
```

`*HSt` contains the internal administration that is required for creating HTML pages and handling forms. Please consult [19] for details. `mkIData` is applied to an (`InIDataId d`) argument that describes the type and value of the iData element that is to be created:

```
:: InIDataId d :== (Init, FormId d)
:: Init         = Const | Init | Set

mkFormId         :: String d → FormId d
```

The function `mkFormId` creates a default (`FormId d`) value, given a unique identifier string[4] and the value of the iData element. The `Init` value describes the use of that value: it is either a `Const`ant or it can be edited by the user. In case of `Init`, it concerns the initial value of the editor. Finally, it can be `Set` to a new value by the program. A (`FormId d`) value is a record that identifies and describes the *use* of the iData element:

```
:: FormId d = { id :: String, ival :: d, lifespan :: Lifespan, mode :: Mode }
```

The `Lifespan` and `Mode` types were introduced in Sect. 2.3. They have the same meaning in the context of iData. To facilitate the creation of non-default (`FormId d`) values, the following straightforward type classes have been defined:

```
class    (<@) infixl 4 att :: (FormId d) att → FormId d
class    (>@) infixr 4 att :: att (FormId d) → FormId d
instance <@ String, Lifespan, Mode
instance >@ String, Lifespan, Mode
```

[4] The use of strings for form identification is an artifact of the iData toolkit. It can be a source of (hard to locate) errors. The iTask system eliminates these issues by an automated systematic identification system.

When evaluated, (`mkIData (init, iDataId)`) basically performs the following actions: it first checks whether an earlier incarnation of the iData element (identified by `iDataId.id`, i.e. the name of the iData element) exists. If this is not the case, or `init` equals `Set`, then `iDataId.ival` is used as the current value of the iData element. If it already existed, then it contains a possibly user-edited value, which is used subsequently. Hence, the final iData element is always up-to-date. This is kept track of in the (`Form d`) record:

```
:: Form d = { changed :: Bool, value :: d, form :: [BodyTag] }
```

The `changed` field records the fact whether the application user has previously edited the value of the iData element; the `value` is the up-to-date value; `form` is the HTML rendering of this iData element that can be used within an arbitrary HTML page.

If we want to lift iData elements to the iTask domain, we need to include a concept of termination because this is absent in the iData framework: an iData application behaves as a set of iData elements that can be edited over and over again by the application user without predetermining some evaluation order. We 'enhance' iData elements with a concept of termination. We define a special function to make such a `taskEditor`. It is an 'ordinary' editor extended with a `Bool`ean iData state in which we record whether the editor task is finished. It is not up to an iData editor to decide whether a task is finished, but this is indicated by the user by pressing an additional button. Hence, a standard iData editor is extended with a button and a boolean storage. These elements are created by the functions `simpleButton` and `mkStoreForm`:

```
simpleButton :: String String (d → d) → HStIO (d → d)
mkStoreForm  :: (InIDataId d) (d → d) → HStIO d | iData d
```

(`simpleButton` *name l f*) creates an iData element whose appearance is that of a push button labeled *l*. It is identified with *name*. When pressed (which is an edit operation by the user), its value is the function *f*, otherwise it is the identity function. (`mkStoreForm` *iD f*) creates an iData element that applies *f* to its current state.

With these two standard functions from the iData toolkit we can enhance any iData editor with a button and boolean storage:

```
taskEditor :: String String a *HSt → (Bool,a,[BodyTag],*HSt) | iData a          1.
taskEditor btnName label v hst                                                  2.
♯ (btn,  hst) = simpleButton btnLabel btnName (const True)  hst                 3.
♯ (done, hst) = mkStoreForm (Init,mkFormId storeLabel False) btn.value hst      4.
♯ (f, btnF)   = if done.value ((>@) Display,Br) (id,btn.form)                   5.
♯ (idata,hst) = mkIData (Init,f (mkFormId editLabel v)) hst                     6.
= (done.value,idata.value,idata.form ++ [btnF],hst)                             7.
where editLabel  = label +> "_Editor"                                           8.
      btnLabel   = label +> "_Button"                                           9.
      storeLabel = label +> "_Store"                                            10.
```

In the function `taskEditor` we create, as usual, an iData element for the value `v` (line 6). The `label` argument is used to create three additional identifiers for the value (`editLabel`), the button element (`btnLabel`), and the boolean storage element (`storeLabel`).

The trigger button (line 3) is a simple button that, when pressed, has the function value (`const True`), and which is the identity function `id` otherwise. The boolean storage is created as an iData storage (line 4). It is interconnected with the trigger button by its value: it applies the function value of the button to its boolean value (initially `False`). Therefore, the value of the boolean storage becomes `True` only if the user presses the trigger button. If the user has indicated that the editor has terminated, then the trigger button should not appear, and the iData element should be in `Display` mode, and otherwise the trigger button should be shown and the iData element should still be editable (line 5). In this way, the user is forced to continue with whatever user interface is created after pressing the trigger button.

The definition of `taskEditor` suggests that we need to extend the `*HSt` with some administration to keep track of the generated HTML, and identification labels for the editors that are lifted. This is what `*TSt` is for. It extends the `*HSt` environment with a boolean value `activated` to indicate the status of a task (when a task is called it tells whether it has to be activated or not, when a task has been evaluated it tells whether it is finished or not), a `tasknr` for the automatic generation of fresh task identifier values, and `html` which accumulates all HTML output. For each of these fields, we introduce corresponding update functions (`set_activated`, `set_tasknr`, and `set_html`):

```
:: *TSt   = { hst :: *HSt, activated :: Bool, tasknr :: TaskID, html :: [BodyTag] }
:: TaskID :== [Int]
set_activated :: Bool      *TSt → *TSt
set_tasknr    :: TaskID    *TSt → *TSt
set_html      :: [BodyTag] *TSt → *TSt
```

With this administration in place, we can use `taskEditor` to lift iData elements to elemental iTasks, viz. ones that allow the user to edit data and indicate termination of this elemental task. Recall that `Task a` was defined as (Sect. 2.1) `*TSt → (a,*TSt)`:

```
editTask :: String a → Task a | iData a
editTask label a = doTask editTask`
where
  editTask` tst=:{tasknr,hst,html}
  # (done,na,nhtml,hst) = taskEditor label (toString tasknr) a hst
  = (na,{tst & activated = done, hst = hst, html = html ++ nhtml})
```

`editTask` takes an initial value of any type and delivers an iTask of that type. When the task is activated, an extended iData element is created by calling `taskEditor`. A unique identifier is generated by this system (function `doTask`, which is explained below), which eliminates the shortcoming that was mentioned above. Any iData element automatically remembers the state of any edit action, no matter how complicated the editor is. The HTML code produced by `taskEditor` is added to the accumulator of the iTask state. In the end all HTML code of all iTasks can be displayed by showing the HTML of the top-task. There can be many active iTasks, so in practice this is probably not what we want. However, for the core system this will do.

The function doTask is an internal wrapper function that is used within the iTasks toolkit for every iTask.

```
doTask :: (Task a) → Task a | iCreate a
doTask mytask          = doTask` o incTaskNr
where doTask` tst=:{activated, tasknr}
       | not activated = (createDefault, tst)
       ♯ (val, tst)    = mytask tst
       = (val,{tst & tasknr = tasknr})
```

doTask first ensures that the task number is incremented. In this way, each task obtains a unique number. Tasks are numbered systematically, in the same way as chapters, sections and subsections are numbered in a book or in this paper: tasks on the same level are numbered subsequently with incTaskNr below, whereas a subtask j of task i is numbered i.j with subTaskNr below. Fresh subtask numbers are generated with newSubTaskNr. We represent the numbering with an integer list, in reversed order.

```
incTaskNr    tst = {tst & tasknr = case tst.tasknr of
                                        []     = [0]
                                        [i:is] = [i+1:is]
                   }
subTaskNr  i tst = {tst & tasknr = [ i:tst.tasknr]}
newSubTaskNr tst = {tst & tasknr = [-1:tst.tasknr]}
```

The systematic numbering is important because it is also used for garbage collection of subtasks (see Sect. 3.6).

Next doTask checks whether the task indeed is the next task to be activated by inspecting the value of tst.activated:

– If not activated, the createDefault value is returned. This explains the overloading context restriction of doTask. As a consequence, an iTask *always has a value*, just as an iData element.
– If activated, the task can be executed. This means that the user can select this task via the web interface, and proceed by generating an input event for this task. Task definitions are fully compositional, so the started tasks may actually consist of many subtasks of arbitrary complexity. When a task is started, it is either activated (or re-activated for further evaluation) or, the task has already been finished in the past, its result is stored as an iData object and is retrieved. In any of these cases, the result of a task (either finished or not yet finished) is returned to the caller of doTask and the task number is reset to its original value.

  It is assumed that any task sets activated to True if the task is finished (indicating that the next task can be activated), and to False otherwise. In the latter case the user still has to do more work on it in the newly created web page.

3.2 Basic Combinators of the Core System

As we have discussed in Sect. 2.4, sequential composition within the iTask toolkit is based on monads. Thanks to uniqueness typing we can freely choose how to thread the unique iTask state `*TSt`: either in explicit environment passing style or in implicit monadic style. In the implementation of the iTask system we have chosen for the explicit style: it gives more flexibility because we have direct access to both the unique iTask state `*TSt` and the unique iData state `*HSt` as is shown in the definition of `editTask`. However, to the application programmer `*TSt` should be opaque, and for her we provide a monadic interface. In the core system, their implementation is simply that of a state transformer function. Therefore, we do not include their code.

The implementation of the alternative `return_D` function is straightforward:

```
return_D :: a → Task a | gForm{|*|}, iCreateAndPrint a
return_D a = doTask (λtst → (a,{tst & html = tst.html ++ toHtml a})
```

The implementation of the prompting combinators `?>>` and `!>>` is also not very difficult:

```
(?>>) infix 5 :: [BodyTag] (Task a) → Task a | iCreate a
(?>>) prompt task = prompt_task
where
    prompt_task tst=:{html = ohtml,activated}
    | not activated = (createDefault,tst)
    # (a,tst=:{activated,html = nhtml}) = task {tst & html = []}
    | activated      = (a,{tst & html = ohtml})
    | otherwise      = (a,{tst & html = ohtml ++ prompt ++ nhtml})

(!>>) infix 5 :: [BodyTag] (Task a) → Task a | iCreate a
(!>>) prompt task = prompt_task
where
    prompt_task tst=:{html = ohtml,activated}
    | not activated              = (createDefault,tst)
    # (a,tst=:{html = nhtml}) = task {tst & html = []}
    = (a,{tst & html = ohtml ++ prompt ++ nhtml})
```

3.3 Reflection (Part I)

The behavior of the described core system is a combination of re-evaluating the application and having the enhanced iData elements retrieve their previous states that are possibly updated with the latest changes done by the application user. The Clean application is still restarted from scratch when a new page is requested from the browser. However, the application now automatically finds its way back to the tasks it was working on during the previous incarnation. Any iTask editor created with `editTask` automatically remembers its contents and state (finished or not) while the other iTask combinators are pure functions which can be recalculated and in this way the system can determine which other tasks have to be inspected next. Tasks that are not yet activated might deliver some default value, but it is not important because it is not used anywhere yet, and the task produces no HTML code. In this way we achieve the same result as in Seaside, albeit that we reconstruct the state of the run-time system by a combination of re-evaluation from scratch and restoring of the previous edit states.

3.4 Work Flow Pattern Combinators of the Core System

The core system presented above is extendable. The sequential composition is covered by the combinators =>> and #>>. In this section we introduce parallel composition, repetition and recursion.

The infix operator (t_1 -&&- t_2) activates subtasks t_1 and t_2 and ends when both subtasks are completed; the infix operator (t_1 -||- t_2) also activates two subtasks t_1 and t_2 but ends as soon as one of them terminates, but it is biased to the first task at the same time. In both cases, the user can work on each subtask in any desired order. A subtask, like any other task, can consist of any composition of iTasks.

```
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a,b) | iCreate a & iCreate b
(-&&-) taska taskb = doTask and
where  and tst=:{tasknr}
        # (a,tst=:{activated=adone}) = mkParSubTask 0 tasknr taska tst
        # (b,tst=:{activated=bdone}) = mkParSubTask 1 tasknr taskb tst
        = ((a,b),set_activated (adone && bdone) tst

(-||-) infixr 3 :: (Task a) (Task a) → Task a | iCreate a
(-||-) taska taskb = doTask or
where  or tst=:{tasknr}
        # (a,tst=:{activated=adone}) = mkParSubTask 0 tasknr taska tst
        # (b,tst=:{activated=bdone}) = mkParSubTask 1 tasknr taskb tst
        = ( if adone a (if bdone b createDefault)
          , set_activated (adone || bdone) tst
          )

mkParSubTask :: Int TaskID (Task a) → Task a
mkParSubTask i tasknr task = task o newSubTaskNr o set_activated True o subTaskNr i
```

The function mkParSubTask is a special wrapper function for subtasks. It is used to activate a subtask and to ensure that it gets a correct task number.

Another iTask combinator is foreverTask which repeats a task infinitely many times.

```
foreverTask :: (Task a) → Task a | iCreate a
foreverTask task = doTask (foreverTask task o snd o task o newSubTaskNr)
```

As an example, consider the following definition:

```
t = foreverTask (sequenceITask -||- editTask "Cancel" createDefault)
```

In t the user can work on sequenceITask (Sect. 2.4), but while doing this, she can always decide to cancel it. After completion of any of these alternatives the whole task is repeated.

More general than repetition is to allow arbitrary recursive work flows. As we have stated in Sect. 2.6, a *crucial* combinator for recursion is `newTask`.

```
newTask :: (Task a) → Task a | iCreate a
newTask task = doTask (task o newSubTaskNr)
```

(`newTask` t) promotes any user defined task t to a proper iTask such that it can be recursively called without causing possible non-termination. It ensures that t is only called when it is its turn to be activated and that an appropriate subtask number is assigned to it. Consider the following example of a recursive work flow:

```
getPositive :: Int → Task Int
getPositive i = newTask (getPositive‘ i)                                1.
where                                                                   2.
    getPositive‘ i = [Txt "Type in a positive number:"]                 3.
                     ?>> editTask "Done" i =>> λni →                    4.
                     if (ni > 0) (return ni) (getPositive ni)           5.
```

Function `getPositive` requests a positive number from the user. If this is the case the number typed in is returned, otherwise the task calls itself recursively for a new attempt. This example works fine. However, it would not terminate if `getPositive‘` calls itself directly in line 5 instead of indirectly via a call to `newTask`. Remember that every editor returns a value, whether it is finished or not. If it is not yet finished, it returns `createDefault`. The default value for type `Int` happens to be zero, and therefore by default `getPositive‘` goes into recursion. The function `newTask` will prevent infinite recursion because the indicated task will not be activated when the previous task is not yet finished. Hence, one has to keep in mind to regard `getPositive` as a task that can be recursively activated, and not as a plain recursive function.

The combinator `repeatTask` repeats a given `task`, until the predicate `p` holds.

```
repeatTask task p = t createDefault
where
   t v = newTask (task v) =>> λnv → if (p nv) (return_D nv) (t nv)
```

Using this combinator the task `getPositive` can be expressed as:

```
getPositive = repeatTask (λi → [Txt "Type in a positive number:"]
                               ?>> editTask "Done" i) (λx → x > 0)
```

Note the importance of the place of the `newTask`. If it would be moved to the recursive call, by replacing (`t v`) by `newTask t v`, the task would always be executed immediately for a first time (i.e. without waiting for activation). This is generally not the desired behavior.

3.5 Reflection (Part II)

With the combinators presented above, iTasks can be composed as desired. As discussed in Sect. 3.4, one can imagine all kinds of additional combinators. For all well-known work flow patterns we have defined iTask combinators that mimic their behavior. They have been discussed in Sect. 2. The actual implementation of the combinators in the iTask library is more complicated than the combinators introduced in the core system. There are additional requirements, such as:

Presentation issues: One can construct complicated tasks that have to be presented to the user systematically and clearly. The system needs to prompt the user for information on the right moment, remove feedback information when it is no longer needed, and so on. Users should be able to work on several tasks in any order they want. Such tasks have to be presented clearly as well, e.g. by creating a separate web page for each task and a button to navigate between these tasks.

Multiple users: A work flow system is a multi-user system. Tasks can be assigned to different users, persistent storage and retrieval of information in a database needs to be handled, think about version control, ensure consistent behavior by ruling out possible race conditions, ensure that the correct information is communicated to each user, inform a user that she has to wait on information to be produced by someone else, and so on.

Efficiency: Real world work flow systems run for years. How can we ensure that the system will scale up and that it can reconstruct itself efficiently?

Features: One can imagine many more options one would like to have. For instance, it might be important that tasks are performed on time. A manager might want to know which tasks and/or persons are preventing the completion of other tasks.

The consequences for the implementation of the core system are described next.

3.6 The Actual iTask Implementation

In this section we discuss the most interesting aspects of the actual implementation by building on the core system.

Handling Multiple Users. On each event every iTask application is (re)started for all its users. All tasks are recalculated from scratch, but only for one user the tasks are shown. By default, tasks are assigned to user 0. As presented in Sect. 2.7, users can be assigned to tasks with the operators `@:` and `@::`. We give the HTML accumulator within the `TSt` environment (Sect. 3.1) a tree structure instead of a list structure, and we keep track of the user to whom a task is assigned, as well as the identification of the application user:

```
:: *TSt     = { ...
                , myId      :: UserID   // id of task user
                , userId    :: UserID   // id of application user
                , html      :: HtmlTree // accumulator for html code
                }
:: HtmlTree = BT [BodyTag]
            | (@@:) infix 0 (UserID,String) HtmlTree
```

```
              | (-@:) infix 0  UserID        HtmlTree
              | (+-+) infixl 1 HtmlTree      HtmlTree
              | (+|+) infixl 1 HtmlTree      HtmlTree
defaultUser = 0
```

(BT *out*) represents HTML output; ((*u* ,*name*)@@:*t*) assigns the html tree *t* to user *u* where *name* is the label of the button with which the user can select this task; (*u*-@:*t*) also assigns the html tree *t* to user *u*, but now *t* should not be displayed. These two alternatives are used to distinguish between output for a given user, and other output. The remaining constructors (t_1+-+t_2) (and (t_1+|+t_2)) place output t_1 left (above) of output t_2.

In a single-user application, the only user is defaultUser; in a multi-user application, the current user can be selected with a menu at the top of the browser window. This feature is added for testing, for the final application one needs of course to add a decent login procedure. Initially, myId is defaultUser, userId is the selected user, and the accumulator html is empty (BT [ ]). After evaluation of a task, the accumulator contains all HTML output of each and every activated iTask. It is not hard to define a filtering function that extracts all tasks for the current user from the output tree.

Version management is important as well for a multi-user web enabled system. Back buttons of browsers and cloning of browser windows might destroy the correct behavior of an application. For every user a version number is stored and only requests matching the latest version are granted. An error message is given otherwise after which the browser window is updated showing the most recent version. Since we only have one application running on the server side, storage and retrieval of any information is guaranteed to be indivisible such that problems in this area cannot occur.

Another aspect to think about is that the completion of one task by one user, e.g. a Cancel action, may remove tasks others are working on (see e.g. the deadlines example in Section 2.8). This effects the implementation of all choice combinators: one has to remember which task was chosen to avoid race conditions.

Optimizing the Reconstruction of the Task Tree. An iTask application reconstructs itself over and over each time a client browser is manipulated by someone. The more progress made in the application, the more tasks are created. Hence, the evaluation tree increases in size and it takes longer to reconstruct it. In a naive implementation, this would lead to a linear increase in time per user action on the work flow, which is clearly unacceptable.

We optimize the reconstruction process similar to the normal rewriting that takes place in the implementation of functional languages such as Clean and Haskell. When a closure is evaluated, the function call is replaced by its result. Similar, when a task is finished, it can be replaced by its result. We have to store such a result persistently, for which we can of course again use an iData element. However, it is not necessary to optimize each result in order to avoid the creation of too many iData storages. We can freely choose between recalculation (saving space) or storing (saving time). In the iTask toolkit we have decided to optimize “big” tasks only. Combinators such as `repeatTask` produce only intermediate results and can be replaced by the next call to itself. For these kinds of combinators the task tree will not grow at all. However, user defined tasks that are created with `newTask` are likely being used to abstract from such “big” tasks.

Here is what the actual `newTask` combinator does, as opposed to the core version of Sect. 3.4.

```
newTask :: (Task a) → Task a | iData a                                          1.
newTask t = doTask (λtst=:{tasknr,hst}                                          2.
 ♯ (taskval,hst) = mkStoreForm (Init,storeId) id hst                            3.
 ♯ (done,v)      = taskval.value                                                4.
 | done          = (v,{tst & hst = hst})                                        5.
 ♯ (v,tst=:{activated = done,hst})                                              6.
                 = t {tst & tasknr = [-1:tasknr],hst = hst}                     7.
 | not done      = (v,{tst & tasknr = tasknr})                                  8.
 ♯ (_,hst)       = mkStoreForm (Init,storeId) (const (True,v)) hst              9.
 = (v,{tst & tasknr = tasknr, hst = hst})                                       10.
)                                                                               11.
where storeId    = mkFormId (tasknr +> "_New") (False,createDefault) <@ Session 12.
```

A storage is associated with task t (line 3) that initially has a default value (line 12). If the task was finished in the past, it is not re-evaluated. Instead, its value is retrieved from the storage (line 4 and 5), otherwise it needs to be evaluated (lines 6–7). If the user actions have not terminated task t, then it has not produced a final value yet, thus the storage need not be updated (line 8). If the user has terminated the task, then the storage is updated with the final value (line 9), and a boolean mark to prevent re-evaluation of this “redex”.

Garbage Collection of iData Objects. The optimization described above prevents the task evaluation tree from growing, but all persistent iData objects created in previous runs are not garbage collected automatically. Although certain results are not needed for the computation of the task tree anymore, one nevertheless might want to keep them for other reasons. Consider the gathering of statistical information such as “who has performed a certain task in the past?” and “which tasks have taken a long time to complete?”. Another reason is that one wants to remember a result of a task, but not of any of its subtasks. We have therefore included variants of certain combinators in the iTask library, such as `repeatTaskGC` and `newTaskGC` which automatically take care of the garbage collection of their subtasks, no matter where they are stored. The numbering discipline plays a crucial role in identifying which subtasks belong to a given task, such that any choice of garbage collection strategy can be implemented.

Higher-Order Tasks. A distinctive feature of the iTask toolkit is the ability to communicate higher-order tasks that have been partially evaluated (Sect. 2.10). In the real world it is obvious that work that has been done partially can be handed over to other persons who finish the work. This is not one of the standard work flow patterns that can be found in contemporary work flow tools (see [24]). We show that the iTask toolkit does support this work flow pattern, and that it does so in a concise way. The complete realization of the (p-!>t) is as follows:

```
(-!>) infix 4 :: (Task s) (Task a) → Task (Maybe s,TClosure a)                1.
              | iCreateAndPrint s & iCreateAndPrint a                         2.
(-!>) p t = doTask (λtst=:{tasknr,html}                                        3.
 ♯ (v,tst=:{activated = done,html = task})                                     4.
            = t {set (BT []) True tst & tasknr = taskId}                       5.
 ♯ (s,tst=:{activated = halt,html = stop})                                     6.
            = p {set (BT []) True tst & tasknr = stopId}                       7.
 | halt      = return (Just s, TClosure (close t))                             8.
                      (set  html                  True  tst)                   9.
 | done      = return (Nothing,TClosure (return v))                           10.
                      (set (html +|+ task)        True  tst)                  11.
 | otherwise = return (Nothing,TClosure (return v))                           12.
                      (set (html +|+ task +|+ stop) False tst)                13.
)                                                                             14.
where close t        = t o (set_tasknr taskId)                                15.
      set html done = (set_html html) o (set_activated done)                  16.
      stopId        = [-1,0:tasknr]                                           17.
      taskId        = [-1,1:tasknr]                                           18.
```

Both the suspendable task t and the terminator task p are evaluated (lines 4–5 and 6–7). Their current renderings are `task` and `stop` respectively, and they both contain the most recent user edit operations. The most exciting spot is line 8: if p is finished (condition `halt` is true), then the task t *as far as it has been evaluated* has to be returned. However one has to realize that a task t is only a recipe that is executed by applying it to its state. When a task is executed, it *always* returns a result and a state, even if the task is not yet finished. This also holds for task t when it is activated in line 5. There actually are no partially evaluated task closures in this system, there are only tasks and when they are applied they return their result. The crucial issue is how to return a partially evaluated task if none exist? The answer is given in line 15! Remember that an iTask application can reconstruct itself completely from scratch. This property also holds for any iTask expression in the application. The only thing we need is the task recipe and the state of a task, and in particular, the task number stored in this state. Given a task number and a task we can reconstruct the work done so far! So by passing the task function and the task number to somebody else, the work can be reconstructed and the person can continue the work. Line 15 assures that an interrupted task is reapplied on the original task number when it is restarted.

4 Related Work

In the realm of functional programming, many solutions that have been inspiring for our work have been proposed to program web applications. We mention just a few of them in a number of languages: the Haskell CGI library [16]; the Curry approach [12]; writing XML applications [9] in *SMLserver* [8]. One sophisticated system is WASH/CGI by [23], based on Haskell. Here, HTML is produced as an effect of the CGI monad whereas we consider HTML as a first-class citizen, using data types. Instead of storing state, WASH/CGI logs all user responses and

I/O operations. These are replayed when needed to bring the application to its desired, most recent state. In iTasks, we replay the program instead of the session, and restore the state of the program on-the-fly using the storage capabilities of the underlying iData. Forms are programmed explicitly in HTML, and their elements may, or may not, contain values. In the iTask toolkit, forms and tasks are generated from arbitrary data types, and always have value. Interconnecting forms in WASH/CGI is done by adding callback actions to submit fields, whereas the iData toolkit uses a functional dependency relation.

Two more recent approaches that are also based on functional languages are Links [5] and Hop [22]. Both languages aim to deal with web programming within a single framework, just as the iData and iTask approach do. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. A Links program stores its session state at the client side. Notable differences between Links and iData and iTasks are that the latter has a more refined control over the location of state storage, and even the presence of state, which needs to be mimicked in Links with recursive functions. Compiling to JavaScript gives Links programs more expressive and computational power at the client side: in particular Links offers thread-creation and message-passing communication, and finally, the client side code can call server side logic and vice versa. The particular focus of Hop is on rendering graphically attractive applications, like desktop GUI applications can. Hop implements a strict separation between programming the user interface and the logic of an application. The main computation runs on the server, and the GUI runs on the client(s). Annotations decide where a computation is performed. Computations can communicate with each other, which gives it similar expressiveness as Links. The main difference between these systems and iTasks (and iData) is that the latter are restricted to thin-client web applications, and provide a high degree of automation using the generic foundation.

iData components that reside in iTasks are abstractions of forms. A pioneer project to experiment with form-based services is Mawl [2]. It has been improved upon by means of Powerforms [3], used in the <bigwig> project [4]. These projects provide *templates* which, roughly speaking, are HTML pages with *holes* in which scalar data as well as lists can be plugged in (Mawl), but also other *templates* (<bigwig>). They advocate compile-time systems, because this allows one to use type systems and other static analysis. Powerforms reside on the client-side of a web application. The type system is used to filter out illegal user input. Their and our approach make good use of the type system. Because iData are encoded by ADTs, we get higher-order forms for free. Moreover, we provide higher-order tasks that can be suspended and migrated.

Web applications can be structured with *continuations*. This has been done by Hughes, in his arrow framework [14]. Queinnec states that "A browser is a device that can invoke continuations multiply/simultaneously" [21]. Graunke *et al* [10] have explored continuations as one of three functional compilation techniques to transform sequential interactive programs to CGI programs. The Seaside [6] system offers an API for programming web pages using a Smalltalk interpreter.

When waiting for new information from the browser, a Seaside application is suspended and continues evaluation as soon as input is available. To make this possible, the whole state of the interpreter's run-time system is stored after a page has been produced and this state is recovered when the next user event is posted such that the application can resume execution. In contrast to iTask, Seaside has to be by construction a single user system.

Our approach is simpler yet more powerful: every page has a complete (set of) model value(s) that can be stored and recovered generically. An application is resurrected by restarting the very same program, which recovers its previous state on-the-fly.

Workflow systems are distributed software systems, and as such can also be implemented using a programming language with support for distributed computing such as D-Clean [25], GdH [20], Erlang, and Java. iTasks, on the other hand, makes effective use of the distributed nature of the web: web browsers act as distributed rendering resources, and the server controls what gets displayed where and when. Furthermore, the interactive components are created in a type-directed way, which makes the code concise. There is no need to program the data flow between the participating users, again reducing the code size.

Our combinator library has been inspired by the comprehensive analysis of work flow patterns of over more than 30 contemporary commercial work flow systems [24]. These patterns are typically based on a Petri-net style, which implies that patterns for *distributing* work (also called *splitting*) and *merging* (*joining*) work are distinct and can be combined more or less arbitrarily. In the setting of a strongly typed combinatorial approach such as the iTasks, it is more natural to define combinator functions that pair splitting and merging patterns. For instance, the two combinators -&&- and -||- that were introduced in Sect. 2.6 pair the *and split – and join* and *or split – synchronizing merge* patterns. Conceptually, the Petri-net based approach is more fine-grained, and should allow the work flow designer greater flexibility. However, we believe that we have captured the essential combinators of these systems. We plan to study the relationship between the typical functional approach and the classic Petri-net based approach in the near future.

Contemporary commercial work flow tools use a graphical formalism to specify work flow cases. We believe that a textual specification, based on a state-of-the-art functional language, provides more expressive power. The system is strongly typed, and guarantees all user input to be type safe as well. In commercial systems, the connection between the specification of the work flow and the (type of the) concrete information being processed, is not always well typed. Our system is fully dynamic, depending on the values of the concrete information. For instance, recursive work flows can easily be defined. In a graphical system the flows are much more static. Our system is higher order: tasks can communicate tasks. Work can be interrupted and conditionally moved to other users for further completion. Last but not least: we generate a complete working multi-user web application out of the specification. Database storage and retrieval of the information, version management control, type driven generation of web forms, handling of web forms, it is all done automatically such that the programmer only needs to focus on the flow specification itself.

5 Conclusions

The iTask system is a domain specific language for the specification of work flows, embedded in Clean. The specification is used to generate a multi-user interactive web-based work flow management system.

The notation we offer is concise as well as intuitive. For functional programmers the monadic style of programming should look familiar. Users of commercial work flow systems, who design work flows, typically use a graphical formalism for this purpose. For this group of potential users a text based approach is likely to be harder to understand. It should be investigated in what way a mapping from a graphical approach to the textual approach can be constructed.

The iTask toolkit covers all standard work flow patterns in a combinatorial style (see Appendix A). Moreover, it adds further expressive power in terms of a strongly typed system, dynamic run-time behavior, and higher-order tasks that can be suspended, passed on to other users, and continued. At the same time it generates a multi-user interactive web-based application that automatically handles sessions, state and state storage, HTML rendering, and more.

This latter feature is due to building the iTask toolkit on top of the iData toolkit. This project provides further evidence that the iData concept is a versatile, elementary unit to create interactive web applications. One particular helpful design decision was to separate handling values and constructing the rendering of the application in the iData toolkit. This allows the iTask toolkit to separately handle the flow of information and the filtering of the correct HTML code for the end user. The iData enabled us to do "task rewriting" in a similar way as expressions are rewritten in languages such as Clean and Haskell. Finally, iTasks profit from these advantages, and strengthen them by extended the expressive power by defining work flow system on a sophisticated high level of abstraction.

Future work will be the investigation of more "unusual" useful work flow patterns. Also we are working on a new option for the evaluation of tasks on the client side using Ajax technology in combination with an efficient interpreter for functional languages [15].

Acknowledgements

The authors would like to thank Phil Wadler for his inspiring exercise, Erik Zuurbier for the many discussions on the state-of-art of contemporary work flow systems and as a source of many examples, and Wil van der Aalst for commenting on the difference between the combinator approach and contemporary work flow specification languages.

References

1. Alimarine, A.: Generic Functional Programming - Conceptual Design, Implementation and Applications. PhD thesis, University of Nijmegen, The Netherlands (2005) ISBN 3-540-67658-9
2. Atkins, D., Ball, T., Benedikt, M., Bruns, G., Cox, K., Mataga, P., Rehor, K.: Experience with a Domain Specific Language for Form-based Services. In: Usenix Conference on Domain Specific Languages (October 1997)
3. Brabrand, C., Møller, A., Ricky, M., Schwartzbach, M.: Powerforms: Declarative client-side form field validation. World Wide Web Journal 3(4), 205–314 (2000)
4. Brabrand, C., Møller, A., Schwartzbach, M.: The <bigwig> Project. ACM Transactions on Internet Technology (TOIT) (2002)
5. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709. Springer, Heidelberg (2007)
6. Ducasse, S., Lienhard, A., Renggli, L.: Seaside - A Multiple Control Flow Web Application Framework. In: Ducasse, S. (ed.), Proceedings ESUG 2004 International Conference – Research Track, volume Technical Report IAM-04-008, pp. 231–254. Institut für Informatik und Angewandte Mathematik, University of Bern, Switzerland, November 7 (2004)
7. Elliot, C.: Tangible Functional Programming. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007), Freiburg, Germany, October 1–3, pp. 59–70. ACM, New York (2007)
8. Elsman, M., Hallenberg, N.: Web programming with SMLserver. In: Dahl, V., Wadler, P. (eds.) PADL 2003. Springer, Heidelberg (2003)
9. Elsman, M., Larsen, K.F.: Typing XHTML Web applications in ML. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 224–238. Springer, Heidelberg (2004)
10. Graunke, P., Krishnamurthi, S., Findler, R.B., Felleisen, M.: Automatically Restructuring Programs for the Web. In: Feather, M., Goedicke, M. (eds.) Proceedings 16th IEEE International Conference on Automated Software Engineering (ASE 2001). IEEE CS Press, Los Alamitos (2001)
11. Hanna, K.: A Document-Centered Environment for Haskell. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015. Springer, Heidelberg (2006)
12. Hanus, M.: High-Level Server Side Web Scripting in Curry. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 76–92. Springer, Heidelberg (2001)
13. Hinze, R.: A new approach to generic functional programming. In: The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Massachusetts, Boston, pp. 119–132 (January 2000)
14. Hughes, J.: Generalising Monads to Arrows. Science of Computer Programming 37, 67–111 (2000)
15. Jansen, J., Koopman, P., Plasmeijer, R.: Efficient Interpretation by Transforming Data Types and Patterns to Functions. In: Nilsson, H. (ed.) Proceedings Seventh Symposium on Trends in Functional Programming, TFP 2006, Nottingham, UK, April 19-21, 2006, pp. 157–172. The University of Nottingham (2006)
16. Meijer, E.: Server Side Web Scripting in Haskell. Journal of Functional Programming 10(1), 1–18 (2000)
17. Plasmeijer, R., Achten, P.: A Conference Management System based on the iData Toolkit. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 108–125. Springer, Heidelberg (2007)

18. Plasmeijer, R., Achten, P.: iData For The World Wide Web - Programming Interconnected Web Forms. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945. Springer, Heidelberg (2006)
19. Plasmeijer, R., Achten, P.: The Implementation of iData - A Case Study in Generic Programming. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 106–123. Springer, Heidelberg (2006)
20. Pointon, R., Trinder, P., Loidl, H.: The Design and Implementation of Glasgow distributed Haskell. In: Mohnen, M., Koopman, P. (eds.) IFL 2000. LNCS, vol. 2011. Springer, Heidelberg (2001)
21. Queinnec, C.: The influence of browsers on evaluators or, continuations to program web servers. In: Proceedings Fifth International Conference on Functional Programming (ICFP 2000) (September 2000)
22. Serrano, M., Gallesio, E., Loitsch, F.: Hop, a language for programming the web 2.0. In: Proceedings ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006), Portland, Oregon, USA, pp. 975–985, October 22-26 (2006)
23. Thiemann, P.: WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Form. In: Krishnamurthi, S., Ramakrishnan, C. (eds.) PADL 2002. LNCS, vol. 2257, pp. 192–208. Springer, Heidelberg (2002)
24. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. QUT Technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane (2002)
25. Zsók, V., Hernyák, Z., Horváth, Z.: Distributed Pattern Design in D-Clean. In: Central-European Functional Programming School, CEFP 2005, oldal, vol. 33 (2005), `http://plc.inf.elte.hu/cefp/download/dclean_dbox_lecturenotes.pdf`

A iTask Toolkit

This is the complete *api* of the iTask toolkit.

```
definition module iTasks

// iTasks library for defining interactive multi-user workflow tasks (iTask) for the web
// defined on top of the iData library

// ©iTask & iData Concept and Implementation by Rinus Plasmeijer, 2006,2007 - MJP
// Version 1.0 - april 2007 - MJP
// This library is still under construction - MJP

import iDataSettings, iDataButtons

derive gForm   Void
derive gUpd    Void, TCl
derive gPrint  Void, TCl
derive gParse  Void
derive gerda   Void
```

```
:: *TSt                          // task state
:: Task a         :== St *TSt a  // an interactive task
:: Void           = Void         // for tasks returning non interesting results,
                                 // won't show up in editors either

/* Initiating the iTask library: to be used with an iData server wrapper!
startTask        :: start iTasks beginning with user with given id, True if trace allowed
                      id < 0  : for login purposes.
startNewTask     :: same, lifted to iTask domain, use it after a login ritual
singleUserTask   :: start wrapper function for single user
multiUserTask    :: start wrapper function for user with indicated id with option to switch
                    between [0..users − 1]
multiUserTask2   :: same, but forces an automatic update request every (n minutes, m seconds)
*/
startTask       ::                  !Int !Bool !(Task a) !*HSt → (a,[BodyTag],!*HSt) | iCreate a
startNewTask    ::                  !Int !Bool !(Task a)       → Task a        | iCreateAndPrint a

singleUserTask  ::                  !Int !Bool !(Task a) !*HSt → (Html,*HSt) | iCreate a
multiUserTask   ::                  !Int !Bool !(Task a) !*HSt → (Html,*HSt) | iCreate a
multiUserTask2  :: !(!Int,!Int) !Int !Bool !(Task a) !*HSt → (Html,*HSt) | iCreate a

/* Setting options for any collection of iTask workflows
(<<@)            :: set iData attribute globally for indicated (composition of) iTasks
*/
class (<<@) infix 3 b :: (Task a) b → Task a
:: GarbageCollect = Collect | NoCollect

instance <<@        Lifespan                 // default: Session
                ,   StorageFormat            // default: PlainString
                ,   Mode                     // default: Edit
                ,   GarbageCollect           // deafult: Collect

defaultUser       :== 0                      // default id of user

// Here follow the iTask combinators:

/* promote any iData editor to the iTask domain
editTask          :: create a task editor to edit a value of given type,
                     and add a button with given name to finish the task
*/
editTask          :: String a                  → Task a   | iData a

/* standard monadic combinators on iTask
(=>>)             :: for sequencing: bind
(#>>)             :: for sequencing: bind, but no argument passed
return_V          :: lift a value to the iTask domain and return it
*/
(=>>) infix  1    :: (Task a) (a → Task b)     → Task b   | iCreateAndPrint b
(#>>) infixl 1    :: (Task a) (Task b)         → Task b
return_V          :: a                         → Task a   | iCreateAndPrint a
```

```
/* prompting variants
(?>>)            :: prompt as long as task is active but not finished
(!>>)            :: prompt when task is activated
(<|)             :: repeat task as long as predicate does not hold, give error otherwise
return_VF        :: return the value and show the HTML code specified
return_D         :: return the value and show it in iData display format
*/
(?>>) infix 5  :: [BodyTag] (Task a)          → Task a   | iCreate a
(!>>) infix 5  :: [BodyTag] (Task a)          → Task a   | iCreate a
(<|)  infix 6  :: (Task a) (a → .Bool, a → [BodyTag])
                                              → Task a   | iCreate a
return_VF      :: a [BodyTag]                 → Task a   | iCreateAndPrint a
return_D       :: a                           → Task a   | gForm {|*|}, iCreateAndPrint a

/* Assign tasks to user with indicated id
(@:)             :: will prompt who is waiting for task with give name
(@::)            :: same, default task name given
*/
(@:)  infix 3  :: !(!String,!Int) (Task a)    → Task a   | iCreateAndPrint a
(@::) infix 3  ::             !Int (Task a)   → Task a   | iCreate a

/* Handling recursion and loops
newTask          :: use the to promote a (recursively) defined user function to as task
foreverTask      :: infinitely repeating Task
repeatTask       :: repeat Task until predict is valid
*/
newTask        :: !String (Task a)            → Task a   | iData a
foreverTask    ::         (Task a)            → Task a   | iData a
repeatTask_Std :: (a → Task a) (a → Bool) → a → Task a   | iCreateAndPrint a

/*  Sequencing Tasks:
seqTasks         :: do all iTasks one after another, task completed when all done
*/
seqTasks         :: [(String,Task a)]            → Task [a] | iCreateAndPrint a

/* Choose Tasks
buttonTask       :: Choose the iTask when button pressed
chooseTask       :: Select one iTask with button, buttons horizontally displayed
chooseTaskV      :: Select one iTask with button, buttons vertically displayed
chooseTask_pdm   :: Select one iTask with pull down menu
mchoiceTask      :: Select several iTasks with marked check boxes
*/
buttonTask       ::  String (Task a)      → Task a           | iCreateAndPrint a
chooseTask       :: [(String,Task a)]     → Task a           | iCreateAndPrint a
chooseTaskV      :: [(String,Task a)]     → Task a           | iCreateAndPrint a
chooseTask_pdm   :: [(String,Task a)]     → Task a           | iCreateAndPrint a
mchoiceTasks     :: [(String,Task a)]     → Task [a]         | iCreateAndPrint a

/* Do m Tasks parallel / interleaved and FINISH as soon as SOME Task completes:
orTask           :: both iTasks in any order, completion when first done
(-||-)           :: same, now as infix combinator
orTask2          :: both iTasks in any order, completion when first done
```

```
orTasks          :: all iTasks  in any order, completion when first done
*/
orTask           :: (Task a,  Task a)     → Task a              | iCreateAndPrint a
(-||-) infixr 3  :: (Task a) (Task a)     → Task a              | iCreateAndPrint a
orTask2          :: (Task a,  Task b)     → Task (EITHER a b)   | iCreateAndPrint a
                                                                & iCreateAndPrint b
orTasks          :: [(String, Task a)]    → Task a              | iData a

/* Do Tasks parallel / interleaved and FINISH when ALL Tasks done:
andTask          :: both iTasks  in any order, completion when both done
(-&&-)           :: same, now as infix combinator
andTasks         :: all iTasks  in any order, completion when all done
andTasks_mu      :: assign task to indicated users, task completed when all done
*/
andTask          :: (Task a,  Task b)     → Task (a,b)          | iCreateAndPrint a
                                                                & iCreateAndPrint b
(-&&-) infixr 4  :: (Task a) (Task b)     → Task (a,b)          | iCreateAndPrint a
                                                                & iCreateAndPrint b
andTasks         :: [(String,Task a)]     → Task [a]            | iCreateAndPrint a
andTasks_mu      :: String [(Int,Task a)] → Task [a]            | iData a

/* Time and Date management:
waitForTimeTask :: Task is done when time has come
waitForTimerTask:: Task is done when specified amount of time has passed
waitForDateTask :: Task is done when date has come
*/
waitForTimeTask :: HtmlTime               → Task HtmlTime
waitForTimerTask:: HtmlTime               → Task HtmlTime
waitForDateTask :: HtmlDate               → Task HtmlDate

/* Experimental department
   Will not work when the tasks are garbage collected to soon !!

-!>             :: a task, either finished or interrupted (by completion of the first task)
                   is returned in the closure if interrupted, the work done so far is
                   returned(!) which can be continued somewhere else
channel         :: splits a task in respectively a sender task closure and receiver task
                   closure; when the sender is evaluated, the original task is evaluated as
                   usual; when the receiver task is evaluated, it will wait upon completion
                   of the sender and then gets its result;
                   Important:
                      Notice that a receiver will never finish if you don't activate the
                   corresponding receiver somewhere.
closureTask     :: The task is executed as usual, but a receiver closure is returned
                   immediately. When the closure is evaluated somewhere, one has to wait
                   until the task is finished. Handy for passing a result to several
                   interested parties.
closureLzTask   :: Same, but now the original task will not be done unless someone is asking
                   for the result somewhere.

*/
:: TCl a        = TCl (Task a)
```

```
(-!>) infix 4   :: (Task stop) (Task a) → Task (Maybe stop,TCl a) | iCreateAndPrint stop
                                                                  & iCreateAndPrint a
channel         :: String (Task a)      → Task (TCl a,TCl a)      | iCreateAndPrint a
closureTask     :: String (Task a)      → Task (TCl a)            | iCreateAndPrint a
closureLzTask   :: String (Task a)      → Task (TCl a)            | iCreateAndPrint a

/* Operations on Task state
taskId          :: id assigned to task
userId          :: id of application user
addHtml         :: add HTML code
*/
taskId          :: TSt → (Int,TSt)
userId          :: TSt → (Int,TSt)
addHtml         :: [BodyTag] TSt → TSt

/* Lifting to iTask domain
(*>>)           :: lift functions of type (TSt→ (a,TSt)) to iTask domain
(@>>)           :: lift functions of (TSt→ TSt) to iTask domain
appIData        :: lift iData editors to iTask domain
appHSt          :: lift HSt domain to TSt domain, will be executed only once
appHSt2         :: lift HSt domain to TSt domain, will be executed on each invocation

*/
(*>>) infix 4   :: (TSt → (a,TSt)) (a → Task b) → Task b
(@>>) infix 4   :: (TSt → TSt) (Task a)         → Task a
appIData        :: (IDataFun a)                 → Task a    | iData a
appHSt          :: (HSt → (a,HSt))              → Task a    | iData a
appHSt2         :: (HSt → (a,HSt))              → Task a    | iData a

/* Controlling side effects
Once           :; task will be done only once, the value of the task will be remembered
*/
Once           :: (Task a)                    → Task a    | iData a
```

Proving Properties of Lazy Functional Programs with SPARKLE

Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer

Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands
{maartenm,marko,rinus}@cs.ru.nl

Abstract. This tutorial paper aims to provide the necessary expertise for working with the proof assistant SPARKLE, which is dedicated to the lazy functional programming language CLEAN. The purpose of a proof assistant is to use formal reasoning to verify the correctness of a computer program. Formal reasoning is very powerful, but is unfortunately also difficult to carry out.

Due to their mathematical nature, functional programming languages are well suited for formal reasoning. Moreover, SPARKLE offers specialized support for reasoning about CLEAN, and is integrated into its official development environment. These factors make SPARKLE a proof assistant that is relatively easy to use.

This paper provides both theoretical background for formal reasoning, and detailed information about using SPARKLE in practice. Special attention will be given to specific aspects that arise due to lazy evaluation and due to the existence of strictness annotations. Several assignments are included in the text, which provide hands-on experience with SPARKLE.

1 Introduction

In 2001, the distribution of the lazy functional programming language CLEAN [5,28,29] was extended with the dedicated proof assistant SPARKLE [11]. The purpose of a proof assistant is to verify the correctness of a computer program without executing it. This is accomplished by means of the mathematical process of formal reasoning, which makes use of the source code of the program and the semantics of the programming language.

SPARKLE is intended as an additional tool for the CLEAN-programmer and aims to make formal reasoning accessible. It is conveniently integrated into the official Development Environment of CLEAN, allows reasoning on the level of the programming language itself and offers dedicated support for dealing with CLEAN-programs. Unfortunately, formal reasoning is a complex mathematical process that requires specialized expertise. Therefore, it is often still difficult to carry out, even in dedicated proof assistants such as SPARKLE.

In practice, SPARKLE has already been applied for various purposes. It has been used for proving properties of I/O-programs by Butterfield [7] and Dowse [14]. In [30,19], Tejfel, Horváth and Kozsik have proposed an extension for it for dealing with temporal properties. Support for class-generic properties has been added to it by van Kesteren [21]. Furthermore, it has also been used in education at the Radboud University of Nijmegen.

The purpose of this paper is to provide the information that is necessary for functional programmers to start making use of Sparkle. A combination of both theoretical and practical expertise will be provided. No special knowledge is required to understand the contents of this paper: a basic understanding of lazy functional languages and elementary logic suffices. Upon completion of this paper, the reader will be able to use Sparkle to prove basic properties of small Clean-programs with minimal effort. Furthermore, a solid foundation will be laid for proving properties that are more complex.

This paper is structured as follows. First, the concept of formal reasoning will be explained independently of Sparkle in Section 2. Then, the important design principles of Sparkle will be summarized in Section 3, and their effect on the way that formal reasoning is implemented will be explained. Then, in Sections 4 and 5 a tutorial of the use of Sparkle in practice is presented. The first part (Section 4) presents a step-by-step introduction of all the basic features of Sparkle; the second part (Section 5) describes several advanced features that are specific for Sparkle. We discuss related work in Section 6 and draw conclusions in Section 7. Finally, the complete tactic library of Sparkle is summarized separately in Appendix A.

The tutorial is written in an explanatory style and contains assignments with which the provided theory can be put into practice. The assignments require the standard Clean 2.2 distribution to be installed, and the Sparkle version from http://www.cs.ru.nl/~marko/research/sparkle/SparkleCEFP2007.zip must be merged in it. The worked out answers to the assignments are available online at http://www.cs.ru.nl/~marko/research/sparkle/cefp2007/.

2 Formal Reasoning

In the following sections, a general introduction to formal reasoning will be presented independently of Sparkle. In Section 2.1, formal reasoning will first be described as an abstract process that transforms input to desired output. In Section 2.2, the underlying formal framework will be identified; this framework is a prerequisite for carrying out formal reasoning. The most important component of the framework is the proof language, which will be explored in more detail in Section 2.3. Finally, the soundness of formal reasoning will be discussed in Section 2.4.

2.1 The Abstract Process of Formal Reasoning

Formal reasoning is a mathematical process that fully takes place on the formal level. The goal of formal reasoning is to verify the correctness of some kind of formal object by means of reasoning about it. The process as a whole can roughly be characterized as follows:

1. Formalize an object o;
2. Formalize a property p that says something about o;
3. Build a formal proof that shows that p holds for o.

If formal reasoning succeeds and a formal proof is built, then it is shown with absolute certainty that the formalized object o behaves as specified by means of property p. This holds for all environments in which o may occur, because the formal proof is obliged to take all possible circumstances into account. As such, a positive result of formal reasoning is more powerful than for instance a positive result of testing, which is restricted by the test-set that was used.

If formal reasoning does not succeed in building a proof, however, then not much information has been gained. It may either be the case that o is incorrect, or it may be the case that the desired behavior of o was incorrectly specified by p, or it may simply be the case that the proof builder did not build the proof in the right way. A negative result of formal reasoning is hard to interpret correctly and is therefore less useful than a negative result of for instance testing.

2.2 Formal Framework

Formal reasoning makes use of the formal representations of the object to reason about (input), the property to prove (input) and the proof to be built (output). Moreover, to ascertain the soundness of reasoning (see Section 2.4), a formal semantics that assigns a meaning to properties must be available as well. The combination of these prerequisites of reasoning will be called a formal framework:

Definition 2.2.1. *(Formal framework)*
A formal framework is a tuple $(O, P, \models_o, \vdash_o)$ such that:
- O is the set that contains all possible objects to reason about;
  $\dashrightarrow$ ($o \in O$ denotes that o is a valid object to reason about)
- P is the set that contains all possible properties that may be specified;
  $\dashrightarrow$ ($p \in P$ denotes that p is a valid property to prove)
- $\models_o$ is the relation that defines the semantics of properties;
  $\dashrightarrow$ ($\models_o p$ denotes that $p \in P$ holds in the context of $o \in O$)
- $\vdash_o$ is the derivation system that defines proofs of properties.
  $\dashrightarrow$ ($\vdash_o p$ denotes that a proof of $p \in P$ exists in the context of $o \in O$)

(The formal framework of SPARKLE is described completely in [13]. In the remainder of this paper, it will be treated implicitly only.)

Note that the elements of a framework are connected: it must be possible to refer to components of objects within properties; the semantics of a property can only be determined in the context of a given object; and the derivation of a proof depends on a given object as well.

Using the notations introduced by the formal framework, formal reasoning can now be characterized as follows:

Definition 2.2.2. *(Formal reasoning)*
Formal reasoning is the process that given a formal framework $(O, P, \models_o, \vdash_o)$, a specific object $o \in O$ and a specific property $p \in P$, attempts to determine

whether $\vdash_o p$ holds or not. From the soundness of the formal framework it then follows that $\vDash_o p$ holds as well.

In other words, the goal of formal reasoning is to determine $\vDash_o$ by means of $\vdash_o$. This approach only makes sense for frameworks in which $\vdash_o$ is less complicated than $\vDash_o$, which is often the case, because derivation systems are usually simpler than semantic relations.

2.3 Proof Language

The most important component of the formal framework is the proof language, which is usually represented by means of a derivation system. The derivation rules of this system are reasoning steps that form the building blocks of proofs. Building proofs is basically the repeated application of these reasoning steps, and can be characterized as follows:

– **Goal**: prove a property p.
– **Apply**: reasoning step R. This transforms p to $p_1, \ldots, p_n$. If $n = 0$, then the proof is complete (R proves p). Otherwise, $p_1, \ldots, p_n$ become the new goals which *all* have to be proved recursively by the same reasoning process.
– **Goal**: prove all properties $p_1, \ldots, p_n$.

In other words, reasoning steps are functions that transform propositions into (possibly more) propositions, and the proof language is the set of functions that one is allowed to apply during reasoning. Furthermore, reasoning itself is 'goal-busting': at each point in time a number of propositions (goals) have to be proved, and these propositions can be simplified (busted) by means of the repeated application of predefined reasoning steps.

The result of reasoning is a derivation tree in which the nodes are propositions (and the root node is the initial proposition to prove) and each set of edges leading from a single node corresponds with a reasoning step. Edges in this tree do not necessarily have to lead to another node, because reasoning steps may produce the empty list of propositions. The leaves of the tree are the propositions that still have to be proved.

The derivation tree is of course the formal representation of a proof. It can easily be serialized, provided that the reasoning steps are named. A serialized proof can be transferred to anyone with knowledge of the formal framework that it uses. Furthermore, the receiver can even automatically check the validity of the proof by re-running it. Note that validating proofs is easy, because it only requires the formal framework, but building proofs is difficult, because it requires the continuous selection of the 'right' reasoning step.

2.4 Soundness of Formal Reasoning

Building formal proofs is an exercise in the repeated simplification of propositions according to predefined reasoning steps. This, however, is a purely syntactic exercise that does not take the actual meaning of propositions into account in any way. In order for the results of reasoning to be meaningful, the underlying formal framework must be sound as well:

Definition 2.4.1. *(Soundness of formal frameworks (1))*
A formal framework $(O, P, \vDash_o, \vdash_o)$ is sound if for all $o \in O$ and $p \in P$ it holds that $\vdash_o p$ implies $\vDash_o p$.

Because $\vdash_o$ is composed of individual derivation rules, the soundness of a formal framework as a whole can be determined by verifying these rules as follows:

Definition 2.4.2. *(Soundness of a derivation rule)*
A derivation rule $R \in \vdash_o$ is sound if for all $p \in P$ it holds that $\vDash_o (p_1 \wedge \ldots \wedge p_n)$ implies $\vDash_o p$, assuming that $R(p) = p_1, \ldots, p_n$.

Definition 2.4.3. *(Soundness of formal frameworks (2))*
A formal framework $(O, P, \vDash_o, \vdash_o)$ is sound if all its derivation rules $R \in \vdash_o$ are sound.

Formal reasoning only makes sense if the underlying formal framework is sound. Soundness should therefore preferably be proved explicitly. If the complexity of the derivation system makes this too difficult, then some degree of confidence can still be gained from practice ('no untrue propositions have ever been proved, so it must be correct'), but this weakens the results of formal reasoning considerably. The soundness of the formal framework of Sparkle has been proved in [13].

Finally, note that for the usefulness of formal reasoning it is important that the reverse property of completeness (for all properties p, $\vDash_o p$ implies $\vdash_o p$) holds too. Full completeness is extremely difficult to achieve for complex frameworks. Using proof theory, however, it can usually be approximated quite closely.

3 Design Principles of Sparkle

The main purpose of Sparkle is to allow functional programmers to reason about the Clean-programs that they are developing, which improves the quality of the program as a whole. The reasoning support that Sparkle offers is in the first place tailored towards this main purpose, although in general Sparkle is also useable for anyone who would like to reason about functional programs. In particular, a frontend for Haskell'98 is currently being added to Clean, which in the future would allow reasoning about mixed Clean /Haskell-programs.

In the following sections, the effect that the main purpose of Sparkle has on its design will be explored closely. In Section 3.1, first the intended users of Sparkle will be analyzed in detail. Then, in Section 3.2 a list of resulting consequences for the design will be presented. Finally, the important consequence of dedicated reasoning will be explored in detail in Sections 3.3 and 3.4.

3.1 Intended Users: Functional Programmers

The intended users of Sparkle are functional programmers, or more specifically anyone who has downloaded the Clean-distribution and is developing programs with it. Of course, there is much diversity in this group, and there is no such thing as 'the functional programmer'. Still, for the sake of design, we will make the following tentative assumptions about the intended users of Sparkle:

– They do not necessarily have much experience with formal reasoning, and may not even know about it at all;
– They often have some theoretical background, and usually have at least a basic understanding of elementary logic;
– They usually have good knowledge of functional programming in general and of Clean (and its semantics) in specific;
– They are not necessarily aware of the benefits of formal reasoning for the purpose of improving the quality of software;
– They are mainly interested in the programs that they develop.

Other proof assistants may be geared towards different users; for instance, the major independent proof assistants (such as for instance Pvs [24] and Coq [31]) are mainly intended for logicians who already know about formal reasoning and are interested in it as well.

3.2 Design Choices

Sparkle implements a theoretically sound formal framework, and therefore fully supports general formal reasoning on the fundamental level. In its design, however, Sparkle focuses mainly on functional programmers as its intended users. The most important choices in the design of Sparkle are:

– The *object language* should be Clean, because this allows programmers to reason on the level of the programming language, which is their area of expertise. Although this has not been realized fully, a good approximation by means of Core-Clean has been adopted by Sparkle (see Section 3.4).
– For the *property language*, it suffices to use a standard first-order logic which has been extended with an equality on arbitrary program expressions. In such a logic most common properties can be expressed easily. Moreover, functional programmers are likely to be capable of handling standard first-order logic. The property language will be introduced in the tutorial in Section 4.3.
– The *semantics* of the property language should conform to the semantics of Clean. This ensures that properties that are proved with Sparkle hold for the real-world Clean-program as well. This is achieved by giving '$e_1 = e_2$' the meaning 'it is possible to interchange e_1 with e_2 in any program without changing its observational behavior'. The full semantics will be introduced on an informal level in the tutorial in Section 4.4.
– Formal reasoning should be *integrated* with programming, such that switching between the two activities becomes easy. This makes formal reasoning

more attractive, because it is linked to an activity that is carried out continuously. The integration of SPARKLE is realized by allowing it to be started directly from the IDE (Integrated Development Environment) of CLEAN, in which case the current project is loaded automatically in SPARKLE.
– The reasoning steps of SPARKLE should be *specialized* for dealing with lazy functional programs in general, and for dealing with CLEAN in specific. In particular, lazy evaluation and explicit strictness have a profound influence on semantics, and therefore on reasoning as well. The specialized features of SPARKLE will be described in Section 5.
– The *first impression* of SPARKLE should be positive, and should entice programmers to continue with formal reasoning. This is realized by SPARKLE's attractive user interface (see tutorial), and by allowing small proofs to be carried out automatically with the hint mechanism (see Section 4.5).
– SPARKLE should have up-to-date and extensive documentation. This paper is the first attempt to achieve this goal.

The design choices with the most profound influence on SPARKLE are the level of the object language and the specialization of the reasoning steps. The consequences of the level of the object language will be examined further in Sections 3.3 and 3.4; the specialized features of SPARKLE will be described in detail later in Section 5.

3.3 Dedicated vs General-Purpose Formal Reasoning

If one wants to add support for formal reasoning to a specific programming language, two different approaches can be taken:

1. Build one's own *dedicated* proof assistant that directly supports reasoning on the level of the programming language itself; or
2. Build a shell around an existing *general-purpose* proof assistant, combined with a translation mechanism to and from its object language.

Currently, several good general-purpose proof assistants are available in practice, such as for instance PVS [24], COQ [31] and ISABELLE [26]. These proof assistants all have a large user base and make use of well-developed formal frameworks that are extremely expressive and powerful. In the shell approach, such a well-established formal framework is re-used automatically, which is a major advantage.

Unfortunately, general-purpose proof assistants have a major disadvantage as well: none have an object language that fully supports the semantics of CLEAN, which is based on lazy graph-evaluation with explicit strictness. Therefore, the evaluation mechanism of the proof assistant cannot be re-used, and an interpreter for CLEAN has to be built completely within the object language of the general-purpose proof assistant. This has the following important drawback:

actual reasoning no longer takes place on the level of the CLEAN*-program, but instead on a meta-representation of it in the object language of the general-purpose proof assistant*

From the programmer's point of view, however, it is crucial that reasoning at least *appears* to be taking place on the level of the CLEAN-program. In the case that a general-purpose proof assistant is used, it is therefore the task of the shell to hide the underlying meta-level completely from the end user. Consequently, applying a reasoning step in a shell actually requires three activities: (1) translate the program and the reasoning step to the meta-level; (2) execute the reasoning step on the meta-level; (3) translate the feedback back to the programming level.

To summarize, the shell approach has the advantage that a well-established formal framework is re-used, but the disadvantage that an interpreter and a two-way translation and communication mechanism have to be realized. We feel that the general-purpose approach poses more practical problems than it offers advantages; therefore, we have chosen to make use of the dedicated approach.

In hindsight, SPARKLE has been the result of only about 18 'man-months' of work, which shows that writing one's own dedicated proof assistant is certainly doable. We estimate that writing a shell would have taken considerably more effort. On the other hand, the formal framework of SPARKLE does lack some expressiveness, but this has turned out to be only a slight disadvantage for reasoning about functional programs.

3.4 SPARKLE's Approximation of Dedicated Reasoning

SPARKLE is a dedicated proof assistant and aims to support formal reasoning on the level of the programming language itself. For this purpose it allows reasoning on the level of CLEAN, but with the following restrictions:

– All uniqueness annotations are removed automatically from the program;
– I/O-operations have no semantic model and are meaningless;
– Overflow and rounding is disregarded;
– Programs are syntactically simplified to an essential subset before reasoning.

Due to the first restriction, it is not possible in SPARKLE to specify properties that make use of uniqueness. Programs with uniqueness, however, can still be loaded: the uniqueness check is first performed as usual, and then the uniqueness annotations are simply removed. Due to the second restriction, it is not possible to use SPARKLE for proving properties of I/O. Due to the third restriction, many laws about numbers (such as for instance $\forall_n . n < n+1$) hold in SPARKLE, but are falsified by programs in which overflow/rounding occurs. Adding user-friendly support for uniqueness, I/O, overflow and rounding is still future work.

The fourth restriction differs from the first three. Firstly, it does not restrict the scope of reasoning, because it allows all programs to be simplified without loss of meaning. Secondly, it always has an influence on reasoning, because *every* program is simplified implicitly. Thirdly, it is almost impossible to avoid, because defining reasoning support (both on the theoretical and on the practical level) for all of the many syntactic constructs of CLEAN is practically undoable.

The simplification of programs is performed automatically by SPARKLE for all programs that are loaded. The target of the simplification is CORE-CLEAN, which is the intermediate language of the Clean-compiler. From the user's point of view, it seems that Sparkle operates on the level of Clean, but reasoning actually takes place on the level of Core-Clean. Still, the level of Core-Clean approximates dedicated reasoning very well, because:

- Core-Clean has the *same expressive power* as Clean.

  Without loss of meaning, any Clean-program can be transformed to an equivalent Core-program, on which reasoning with Sparkle is possible. Furthermore, the transformation itself has already been implemented in the actual Clean-compiler. Because both Sparkle and the compiler are written in Clean, the existing transformation can be re-used. This not only saves a lot of time, but also ensures soundness of the transformation.
- Core-Clean is a *subset* of Clean.

  Programs in Core-Clean can easily be understood by Clean-programmers, because they make use of the syntax and semantics of Clean. Understanding the program to reason about is vital for the success of formal reasoning.
- Programs in Core-Clean are *very similar* to their Clean-originals.

  The changes between the Core-program and the Clean-original are mainly syntactical in nature, and can in many cases even be hidden by Sparkle. Moreover, the structure of the program is unchanged. As a result, much of the programmer's expertise of the source program is still valid on the Core-Clean level. Again, this increases the understanding of the program to reason about.

Of the four restrictions, the lack of support for dealing with I/O is the most significant, as I/O is an important component of many programs and one would like to reason about it. On the other hand, the usefulness of properties that make use of uniqueness still has to be established, and rounding and overflow are not an issue for the majority of programs. Furthermore, Core-Clean is a suitable intermediate reasoning level.

The differences between Core-Clean and Clean, as well as the feature of Sparkle to present Core-programs as if they were Clean-programs, will be explained further in the Tutorial in Section 4.1.

4 Tutorial Part I: Getting Started with Sparkle

In the following sections, a step-by-step introduction of the basic functionality of Sparkle will be presented. The introduction covers the user interface, the specification of programs and properties, the semantics, and the three different supported styles of reasoning. At various places assignments are included, with the purpose of giving the reader the opportunity to gain hands-on experience with the Sparkle proof assistant.

The tutorial will be continued in Section 5, in which the specialized features of Sparkle will be described. A summary of all available reasoning steps is given in Appendix A.

In order to carry out the assignments in the tutorial, the standard Clean 2.2 distribution (available at http://clean.cs.ru.nl) must be installed, and the files from http://www.cs.ru.nl/~marko/research/sparkle/SparkleCEFP2007.zip must be merged in it. This additional package contains both a full version of Sparkle, and the used example programs `undefined` and `primes` (which will be placed in the `Examples\CEFP` folder of the Clean distribution). Note that Sparkle is available for Windows only. The answers to the assignments are available at http://www.cs.ru.nl/~marko/research/sparkle/cefp2007/.

4.1 Loading a Program

The first step of formal reasoning with Sparkle is loading a Clean-program into its memory. This program provides the context information that is required for stating and proving properties. The fastest way of starting Sparkle and loading a program is by means of the standard IDE of Clean, in which access to Sparkle has been integrated:

Assignment 1. *(Loading a program into* Sparkle *automatically)*
(a) Open the Clean-project `primes.prj` in the `Examples\CEFP` folder.
(b) Examine the code of the main module (`primes.icl`) and attempt to predict the behavior of the program. Then, compile and run the program.
(c) Find the `Theorem Prover Project` option and use it to launch Sparkle.

Alternatively, programs (and individual modules) can also be loaded from within Sparkle, either by opening entire projects (Ctrl-O), or by opening the standard environment only (Ctrl-E), or by adding single modules (Ctrl-+).

Internally, Sparkle maintains its own representation of the program. In this representation, a program is simply considered to be a list of (interdependent) modules, and each module is considered to be a list of definitions. Sparkle does not distinguish between the definition (`.dcl`) and implementation (`.icl`) parts of a module and allows access to all components of a program at any time.

$$\begin{aligned}
\mathit{Program} &:== \mathit{Module}^* \\
\mathit{Module} &:== \mathit{Definition}^* \\
\mathit{Definition} &:== \mathit{Algebraic\ Type} \mid \mathit{Record\ Type} \mid \mathit{Function} \mid \mathit{Class} \mid \mathit{Instance}
\end{aligned}$$

Sparkle has a powerful graphical user interface that allows the structure of the loaded program to be inspected in detail:

Assignment 2. *(Browsing through the program structure)*
(a) Find the window that displays the list of modules that are currently loaded. In this list, find the `primes` module and open it.
(b) The opened window actually filters *all* available definitions with the formula 'functions from the `primes` module'. Change the filter to find all functions in `StdList` and `StdFunc` that begin with the letter 's'.

The user interface also allows each individual definition of the loaded program to be displayed in a separate window. Furthermore, these definition windows are interconnected by means of the symbols that are used within it:

Assignment 3. *(Browsing through the program components)*
(a) Open the definition of the function `isPrime` in the `primes` module.
(b) Follow the internal link to the `canBeDividedByAny` function.
(c) Follow the internal link to the predefined `rem` function.

SPARKLE is a *dedicated* proof assistant that aims to support reasoning on the level of the programming language. Unfortunately, reasoning on the level of CLEAN is not practical, because of the many different syntactical constructs that are allowed. Therefore SPARKLE uses CORE-CLEAN, which is basically the subset of CLEAN in which all syntactic sugar has been removed, as intermediate reasoning language. The only remaining definitions in CORE-CLEAN are algebraic types and global functions, and expressions may only be constructed by means of applications, case distinction and lets.

Even though CORE-CLEAN is a small language only, all CLEAN-programs can be represented in it. When a CLEAN-program is loaded into SPARKLE, it is always automatically converted to CORE-CLEAN. As a result, the program in the memory of SPARKLE differs from the original CLEAN version. Some important differences between the CLEAN-program and its CORE-CLEAN-equivalent are:

- All local functions have been lifted to the global level;
- All pattern matches have been transformed to case distinctions;
- All sharing has been expressed by means of recursive lets;
- All overloading has been expressed by means of dictionaries;
- All synonym types and macro's have been expanded fully;
- All list comprehensions and dot-dot-expressions have been transformed to function applications.

Fortunately, the differences between the internally loaded CORE-CLEAN program and the original CLEAN version only have a slight effect on reasoning, and are therefore hardly noticeable most of the time. Furthermore, the user interface of SPARKLE is able to optionally display parts of CORE-CLEAN programs in the syntax of their original CLEAN versions:

Assignment 4. *(Effect of the optional display options)*
(a) Open the function definitions `isPrime` and `canBeDividedByAny` from the `primes` module and `span` from the `StdList` module.
(b) Toggle the display options `Pattern Matching` and `Case/Let vs #/!`. The 'real' CORE-CLEAN program is displayed when the options are toggled off.
(c) There is one difference between the internal version of `isPrime` and the CLEAN version that cannot be hidden. What is this difference?

4.2 Undefinedness in Clean and Core-Clean

As in any other programming language, computations in Core-Clean and in Clean can terminate erroneously. This can happen in a number of situations, for example when dividing by zero, or when a partial function is applied to an argument for which it was not intended. Additionally, Clean even offers two standard functions that always terminate erroneously, namely `abort` and `undef`.

One of the features of lazy languages is that it is possible for a computation to produce a (partial) end result, even when it contains subcomputations that terminate erroneously. This is only possible, however, when the subcomputation is not needed for producing the end result at all.

Assignment 5. *(Partial undefinedness in practice)*
(a) Open the `undefined` project with the IDE. Run and compile it.
(b) Replace the body of `my_undefined` with another computation that also terminates erroneously.
(c) Cycle through the available `Start` bodies and examine the run-time results.

A formal model of Clean needs to be able to handle expressions that contain undefined subexpressions. For this purpose, Core-Clean defines the additional expression alternative '$\perp$'. This constant expression is treated as a base value of any type, because a computation of any type can terminate erroneously. All different kinds of errors are treated equally; therefore, only one $\perp$ suffices and it does not need additional arguments.

Note that $\perp$ is a special value with special characteristics. It cannot be used as a pattern, or in a case distinction. In fact, it is not possible at all in Clean to produce a defined result based on a successful check of undefinedness.

Assignment 6. *(Undefinedness cannot be detected)*
(a) What famous (unsolvable) problem would be solved if it was possible to detect undefinedness within a Clean program?

4.3 Stating a Property

A property in Sparkle is a logical statement, either true or false, that deals with the executional behavior of a Clean-program. Properties can be used to state that the program functions correctly with respect to its specification. Expressing the desired behavior of a program by means of properties is very useful.

Sparkle allows properties to be expressed in an extended first-order logic. The usual logic operators $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies) and $\leftrightarrow$ (iff) are supported, as well as the quantors $\forall$ (for all) and $\exists$ (exists), and the constants `TRUE` and `FALSE`. Variables and quantors can range over propositions and over expressions of an arbitrary type, but not over predicates or relations of any kind. To state properties of programs, the logic also supports *equality* on expressions.

$$
\begin{aligned}
\mathit{Prop} ::= {} & \mathit{Var}^{\mathit{Prop}} \\
& \mid \texttt{TRUE} \mid \texttt{FALSE} \\
& \mid \neg \mathit{Prop} \mid \mathit{Prop} \wedge \mathit{Prop} \mid \mathit{Prop} \vee \mathit{Prop} \mid \mathit{Prop} \rightarrow \mathit{Prop} \mid \mathit{Prop} \leftrightarrow \mathit{Prop} \\
& \mid \forall_{\mathit{Var}^{\mathit{Prop}}}.\mathit{Prop} \mid \forall_{\mathit{Var}^{\mathit{Expr}}}.\mathit{Prop} \mid \exists_{\mathit{Var}^{\mathit{Prop}}}.\mathit{Prop} \mid \exists_{\mathit{Var}^{\mathit{Expr}}}.\mathit{Prop} \\
& \mid \mathit{Expr} = \mathit{Expr}
\end{aligned}
$$

Many concepts of the proposition level are also available on the expression level, which can be a little confusing. Note for instance the subtle differences between:

- `True` and `False`, which are expressions of type `Bool`, and `TRUE` and `FALSE`, which are propositions;
- `not`, `&&` and $||$, which are CLEAN-functions that operate on values of type `Bool`, and $\neg$, $\wedge$ and $\vee$, which are operators that connect propositions;
- `==`, which is an overloaded CLEAN-function that produces a `Bool` and must be defined manually for each type, and $=$, which produces a proposition and is available automatically for each type.
  (the CLEAN-*function* `==` *is computable and cannot compare undefined values, while the formal* $=$ *is not computable and can compare undefined values; this additional expressiveness is really important, because many properties have definedness preconditions that could otherwise not be expressed)*

On the other hand, the availability of the expression level also allows boolean expressions to sometimes be used as predicates and relations (see Section 5.7).

Assuming the context of the `primes` project, examples of properties are:

1. $\forall_P \forall_Q.(P \wedge Q) \leftrightarrow (Q \wedge P)$
2. $17 > 12 = \texttt{True}$
3. $\forall_f \forall_{xs} \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys$
4. $\forall_{xs}.\texttt{reverse}\ (\texttt{reverse}\ xs) = xs$
5. $\forall_n \forall_{xs}.(n < \texttt{length}\ xs = \texttt{True}) \rightarrow \texttt{length}\ (\texttt{take}\ n\ xs) = n$
6. $\forall_i \forall_j.(i > j = \texttt{True} \wedge j > 0 = \texttt{True}) \rightarrow \texttt{primes !!}\ i > \texttt{primes !!}\ j = \texttt{True}$

Of these properties, the first does not refer to any component of the program; in fact, it is a *tautology* which is independent of any program. The second property refers to the function `>`, which is defined for integers in the module `StdInt`. The third, fourth and fifth properties refer to the functions `map`, `++`, `reverse`, `take` and `length`, which are all defined in the module `StdList`. The sixth property, finally, is the only property that is really specific for the `primes` project. It not only depends on the standard functions `>` and `!!`, but also on the `primes` function of the `primes` module.

Assignment 7. *(Validity of the example properties)*

(a) Of the six example properties, only five are true, and one is in fact false (it needs an additional precondition). Which one is false?
(*Hint:* lists may be infinite in CLEAN)

(b) What happens to the sixth property if either i or j is undefined?

The only way to enter properties in SPARKLE is by means of textual input. The parser allows the natural syntax to be used, with the following conventions:

- `~P` denotes $\neg P$;
- `P /\ Q` denotes $P \wedge Q$;
- `P \/ Q` denotes $P \vee Q$;
- `P -> Q` denotes $P \rightarrow Q$;
- `P <-> Q` denotes $P \leftrightarrow Q$;
- `_|_` denotes $\perp$;
- `[x]` denotes $\forall_x$; and
- `{x}` denotes $\exists_x$.

Type-checking of propositions is performed automatically by Sparkle. During this check, the types of the variables are inferred as well. Alternatively, it is also possible to explicitly specify the type of a variable in a quantor. These explicit types may contain type variables, which are implicitly assumed to be bound by universal quantors. Typed quantors are denoted by:

- `[x::a]` denotes $\forall_{x::a}$; and
- `{x::a}` denotes $\exists_{x::a}$.

Assignment 8. *(Specify the example properties (1))*
(a) Use `New Theorem` to manually enter all six example properties. (*Hint:* in case of failure, attempt to add brackets)

Assignment 9. *(Specify properties with overloading)*
The manual specification of types is essential when making use of overloading:
(a) Without explicit types, attempt to specify $\forall_x \forall_y . x + y = y + x$.
(b) Use explicit types (x :: `Int`, y :: `Int`) to help Sparkle solve the overloading in $\forall_x \forall_y . x + y = y + x$.

For the sake of convenience, Sparkle offers two features to make the manual specification of properties easier:

- Each free symbol in the proposition is assumed to be a variable, and a universal quantor is created automatically for it. This feature allows universal quantors to be omitted when specifying properties. It also means, however, that mistyping the name of an identifier, or using an identifier that is not defined by the current program, does *not* lead to a bind error, but instead results in an incorrect universal quantor.
- When possible, boolean expressions are automatically lifted to propositions by implicitly adding '= `True`'. This feature shortens specifications, but may also lead to confusion between the expression and the proposition level. Note that the '= `True`' behind a lifted boolean expression is not even displayed by Sparkle if the `Boolean Predicates` display option is turned on.

Assignment 10. *(Specify the example properties (2))*
(a) Specify the example properties again, using the features described above. Do *not* quit Sparkle afterwards.

Sparkle organizes theorems and proofs into sections, much in the same way as Clean organizes definitions into modules. Sections are stored in a semi-readable internal format in Sparkle's `\Sections` subdirectory. Theorems and (parts of) proofs can be assigned to individual sections, which must then be saved explicitly. The special section `main` is always available, but it cannot be saved and should only be used for temporary properties. A warning for users: Sparkle does not save sections automatically, and does not prompt you to do so either!

Assignment 11. *(Save properties into sections)*
(a) Create a new section with the name `temp`.
(b) Open both the `main` section and the `temp` section.
(c) Move the example properties from the `main` section into the `temp` section.
(d) Save the `temp` section and quit Sparkle.

Of course, sections can be loaded into Sparkle as well. Because the contents of a section may depend on various other components, the following actions are carried out when a section is loaded:

– First, it is verified if the symbols are available that are required for stating the properties of the section. If this is not the case, then the section is not loaded at all. Otherwise, theorems are created for the properties of the section. The proofs themselves, however, are not loaded yet.
– Then, the sections are loaded recursively that contain the theorems that are used within the proofs of the top-level section.
– Finally, the proofs of the section are loaded and carried out again, step by step. If a step fails, which may be the case if a definition within the program has been altered (but its name and type were unchanged), then the proof can be loaded partially until the error point.

After this process, it can be guaranteed that the internal state of Sparkle is consistent, and that all proofs that were loaded successfully are valid.

Assignment 12. *(Load sections into memory)*
(a) Start Sparkle manually (directly and not from within the IDE).
(b) Attempt to load the predefined section `lists`.
(c) Use Ctrl-O to open the `primes` project from within Sparkle.
(d) Load the predefined section `lists`.
(e) Load the section `temp` of the previous assignment.

4.4 The Meaning of Properties

The meaning of properties is described by a formal algorithm that determines whether a given property, in the context of a given program, is true or false. This algorithm is expressed at the formal level only, and cannot be executed in practice, neither by a human nor by a computer. If it could be executed, formal reasoning would not have been necessary in the first place.

A meaning must be provided for all alternatives of SPARKLE's first-order logic, which was introduced in Section 4.3. This logic contains both standard elements (`TRUE`, `FALSE`, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\forall$ on propositions, $\exists$ on propositions) and customized ones ($=$, $\forall$ on expressions, $\exists$ on expressions). The meaning of the standard elements is the same as in standard logic, which we assume to be well-known. The meaning of the customized elements is as follows:

– The equality $e_1 = e_2$ holds if for all programs P the *observational* behavior stays the same if e_1 is interchanged with e_2 (or vice versa, e_2 with e_1). The observational behavior of a program is the visible output that is produced when it is executed. SPARKLE cannot deal with programs that perform I/O; therefore, only output that is displayed on the console is considered.

  To be able to determine the equality between observational behaviors, it has to be taken into account that programs may not terminate, and that the output that they produce may be infinite. On the formal level, observational behavior is therefore modeled by time indexed *streams*, and *bisimulation* is used to determine equality. On the intuitive level, this is equivalent to assuming that infinite time is available to programs, and that the resulting infinite streams are equal only if all their finite substreams are equal.

  Finally, note that it is not possible to determine if e_1 and e_2 are semantically equal based only on the observational behaviors of the programs `Start = `e_1 and `Start = `e_2. This is because e_1 and e_2 may be functions that only produce meaningful output when they are supplied with arguments.
– The universal quantification $\forall_x.P$ holds if for all wellformed expressions E the instantiated proposition $P[x \mapsto E]$ holds. An expression E is *wellformed* if the resulting $P[x \mapsto E]$ is both closed and welltyped.

  Note that the undefined expression $\perp$ is always a valid value for E, because it is closed and of any type. Furthermore, if the domain of x allows for it, infinite expressions are also valid values for E.
– The meaning of the existential quantification $\exists_x.P$ is defined in the same way as the universal quantification.

Assignment 13. *(Examples of (in)equality)*
(a) Are '`ones`' and '`let x = [1:x] in x`' equal? If so, argue; if not, give the program that distinguishes between them.
(b) Same question for '`ones`' and '`ones ++ ones`'.
(c) Same question for '`ones`' and '`[2] ++ ones`'.
(d) Same question for '`ones`' and '`ones ++ [2]`'.
(e) Same question for '$\perp$' and '`[1:`$\perp$`]`'.
(f) Same question for '$\perp$' and 'λ`x.`$\perp$'.
(*Hint*: make use of explicit strictness)
(g) Same question for '$\perp$' and '`let x = x in x`'.
(*Hint*: only basic values and constructors are meaningful output)

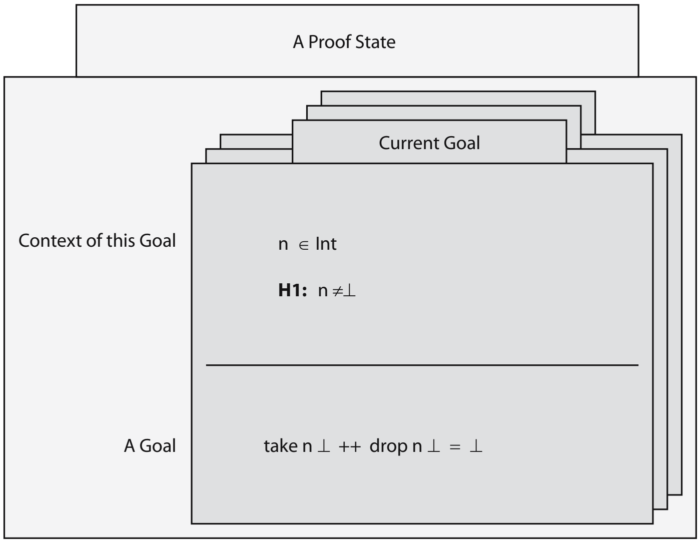


Fig. 1. A proof state

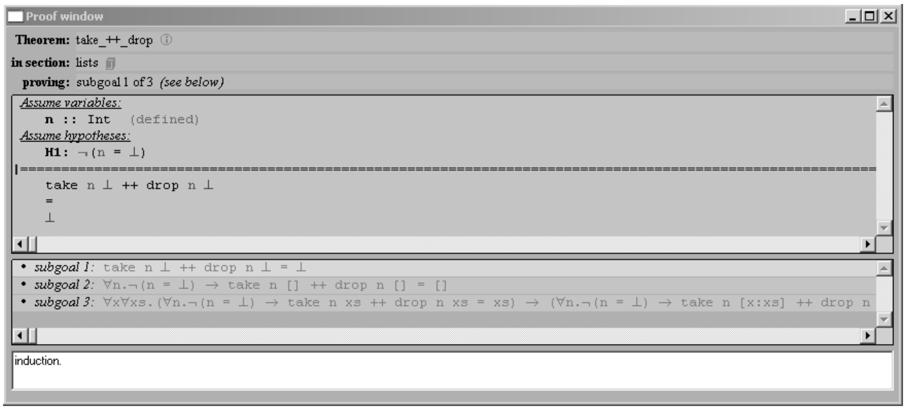
4.5 Reasoning Style in SPARKLE

As most modern day proof assistants, SPARKLE is based on the LCF-approach. This means that reasoning takes place by the repeated simplification of a list of goals by means of the application of tactics. This process of reasoning was first introduced by the LCF[18] proof assistant, and has since been named after it.

The theoretical background of this style of reasoning was already introduced in Sections 2.3 and 2.4. From the user's point of view, each theorem requires the repeated manipulation of a list of goals (=properties to be proved) by means of the application of tactics (=reasoning steps). The goals can be proved in any order; the goal currently being manipulated is called the *active* goal and the others are called *subgoals*. The tactics must be selected from a fixed library, and are guaranteed to be sound. The formal proof tree is maintained internally by SPARKLE and can be browsed manually for an overview of the proof, but it is otherwise not needed for reasoning at all.

Assignment 14. *(Backwards proving)*
(a) Why is SPARKLE's reasoning style sometimes also called *backwards proving*?

A goal corresponds to a property that still to be proved, but on the syntactic level it is broken into components which can be manipulated separately by the reasoning process. The components of a goal are introduced variables, introduced hypotheses and the 'to prove'. If $x_1, \ldots, x_n$ are the introduced variables, $H_1 : P_1, \ldots, H_m : P_m$ are the introduced hypotheses, and Q is the to prove, then the goal corresponds to the property $\forall_{x_1 \ldots x_n}.P_1 \rightarrow \ldots P_m \rightarrow Q$.



```
Proof window
Theorem: take_++_drop
in section: lists
 proving: subgoal 1 of 3 (see below)
Assume variables:
    n :: Int  (defined)
Assume hypotheses:
    H1: ¬(n = ⊥)
|==========================================================================================
    take n ⊥ ++ drop n ⊥
    =
    ⊥

• subgoal 1: take n ⊥ ++ drop n ⊥ = ⊥
• subgoal 2: ∀n.¬(n = ⊥) → take n [] ++ drop n [] = []
• subgoal 3: ∀x∀xs.(∀n.¬(n = ⊥) → take n xs ++ drop n xs = xs) → (∀n.¬(n = ⊥) → take n [x:xs] ++ drop n

induction.
```

Fig. 2. Screen shot of the SPARKLE proof window at the same proof state as in Fig. 1

Assignment 15. *(Decompose the property)*
The proof states in Fig. 1 and Fig. 2 are taken from an actual proof.
(a) Which property corresponds to the current goal in Fig. 1?
(b) Which property was the starting point of the proof?

4.6 Proving a Simple Property

In this section, we will use SPARKLE to prove a simple property which concerns the behavior of the `map` function from the standard environment of CLEAN.

Assignment 16. *(Specification of a property of* `map`*)*
(a) Open SPARKLE from scratch, then load the standard environment (Ctrl-E).
(b) Create a new section with the name `map_section`.
(c) In `map_section`, create a new theorem named `map_property`, stating:
$$\forall_f \forall_{xs} \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys$$
(d) Open the proof window (Ctrl-P) that corresponds to the created theorem.

Building a proof is the repeated process of selecting tactics and applying them on the current goal. For this process, SPARKLE makes a total of 39 tactics available, which are all described briefly in Appendix A. The user interface of SPARKLE allows tactics to be applied by means of three different methods:

– The *hint mechanism*, which is activated by opening the Tactic Suggestion Window during proving. This window holds a dynamically updated list of suggestions for tactics that can be applied to the current goal. SPARKLE generates these suggestions automatically based on built-in heuristics. Each suggestion is assigned a score between 1 and 100 that indicates the likelihood of that tactic being helpful in the proof. Based on this score, the suggestions are ordered. A suggested tactic can be applied by either clicking on it, or by means of its associated hot-key (F1 for the first hint, F2 for the second, etc.).

It is also possible to configure Sparkle to apply the top hint automatically if it has a score higher than a manually set threshold.

The hint mechanism is mainly for *beginning* Sparkle users. It is fast and easy to use, and requires little expertise of the available tactics (simply trust Sparkle !). The hint mechanism is a valuable tool that can be used as a means of learning Sparkle, and with which many small proofs can be built fully. However, it is not very powerful and by no means failsafe. Sometimes the right tactic is not suggested, or several wrong tactics get high scores.

– The *tactic dialogs*. Each tactic has its own dialog that can be opened by clicking on its name in the Tactic List Window. This dialog has entries for all the arguments that can be given to the tactic. When possible, the current goal is used to restrict the input to valid values only. When all arguments have been entered, the tactic can be applied from the dialog directly.

  The tactic dialogs are for *intermediate* users. This method of proving is both powerful, because all tactics can be applied this way, and fairly easy, because one does not need to memorize the name or syntax of a tactic, nor the arguments that it requires.

– The *command line interface*. This is a textual interface, located at the bottom of the Proof Window, that is for *advanced users* only. It is powerful, but requires extensive expertise of Sparkle and its tactics. However, once mastered, it is the fastest way of building proofs, because all tactics can be applied this way and it does not require opening additional dialogs at all.

The property of `map` that was given above is very easy and can therefore be proved automatically with the hint mechanism:

Assignment 17. *(Proving the* `map` *property with the hint mechanism)*

(a) Open the Tactic Suggestions Window (Ctrl-H) and set the threshold to 1.
(b) Set the threshold back to 101. Why is this necessary prior to (c)?
(c) Enter ◀`Restart.`▶ at the command-line interface.
(From now on, ◀`cmd.`▶ *will be used to denote textual input to the command-line. For reasons of parsing, these commands have to end with a closing '.', otherwise* Sparkle *will not be able to recognize them.)*
(d) Redo the proof by applying suggestions manually with the hot-keys.

The complete proof tree of the example property has now been stored internally by Sparkle. By means of the Theorem Info Window, this proof tree can be browsed and inspected in detail:

Assignment 18. *(Browsing through the proof)*

(a) Open the Theorem Info Window of the completed proof.
(b) Click 'browse' after the first tactic and then browse through the proof using the 'previous' and 'next' buttons.
(c) Undo the first application of `Reflexive` only.
(d) Click on the brown star to return to the Proof Window.
(e) Use a different tactic to prove the goal.

The hint mechanism has succeeded in completing the proof automatically, and it did not require any expertise at all. The downside to this, unfortunately, is that no understanding of the tactics has been gained in the process. Therefore, below we will present the entire proof again, and this time we will explain each tactic that was applied too.

The initial goal is simply the property to be proved:

$$\boxed{\frac{\text{-}}{\forall_f \forall_{xs} \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys}} \quad (1)$$

Because both `map` and `++` are tail-recursive, structural induction on *xs* is likely to be useful here. This is accomplished by applying the tactic ◀`Induction xs.`▶. Three new goals(1.1,1.2,1.3) are created: one for the case that *xs* is $\perp$; one for the case that *xs* is `Nil`; and one for the case that *xs* is an application of `Cons`. Note that $\perp$ is a base value of any type and is therefore always treated by induction as a constructor case.

$$\boxed{\frac{\text{-}}{\forall_f \forall_{ys}.\texttt{map}\ f\ (\perp\ \texttt{++}\ ys) = \texttt{map}\ f\ \perp\ \texttt{++}\ \texttt{map}\ f\ ys}} \quad (1.1)$$

The current proposition starts with two universal quantifications, on which it does not make sense to perform induction (on *f* it is not possible, and on *ys* it does not help because `++` is not tail-recursive in its second argument). It is therefore best to apply ◀`Introduce f ys.`▶, which removes the quantors and introduces the variables *f* and *xs* in the context of the goal. After this action, the main proposition can be accessed more easily.

$$\boxed{\frac{f :: \texttt{b} \rightarrow \texttt{a},\ ys :: \texttt{[b]}}{\texttt{map}\ f\ (\perp\ \texttt{++}\ ys) = \texttt{map}\ f\ \perp\ \texttt{++}\ \texttt{map}\ f\ ys}} \quad (1.1')$$

Due to the strictness of `map` and `++` and the presence of $\perp$ arguments, redexes are present in the current goal. The tactic ◀`Reduce NF All.`▶ can be used to reduce all redexes in the current goal to normal form. With other parameters, the tactic `Reduce` can also be used for stepwise reduction, reduction to root normal form, reduction of one particular redex and reduction in the goal context.

$$\boxed{\frac{f :: \texttt{b} \rightarrow \texttt{a},\ ys :: \texttt{[b]}}{\perp = \perp}} \quad (1.1'')$$

This is clearly a trivial goal, because equality is a reflexive relation. Such reflexive equalities are proved immediately with the final tactic ◀`Reflexive.`▶.

$$\boxed{\frac{\text{-}}{\forall_f \forall_{ys}.\texttt{map}\ f\ (\texttt{[]}\ \texttt{++}\ ys) = \texttt{map}\ f\ \texttt{[]}\ \texttt{++}\ \texttt{map}\ f\ ys}} \quad (1.2)$$

This is the second goal of induction, created for the case that *xs* is the empty list. Again, induction makes no sense for *f* and *ys*, and they should therefore be introduced in the goal context by means of ◀`Introduce f ys.`▶.

$$\boxed{\dfrac{f :: \mathtt{b} \rightarrow \mathtt{a},\ ys :: \mathtt{[b]}}{\mathtt{map}\ f\ (\mathtt{[]}\ \mathtt{++}\ ys) = \mathtt{map}\ f\ \mathtt{[]}\ \mathtt{++}\ \mathtt{map}\ f\ ys}} \qquad (1.2')$$

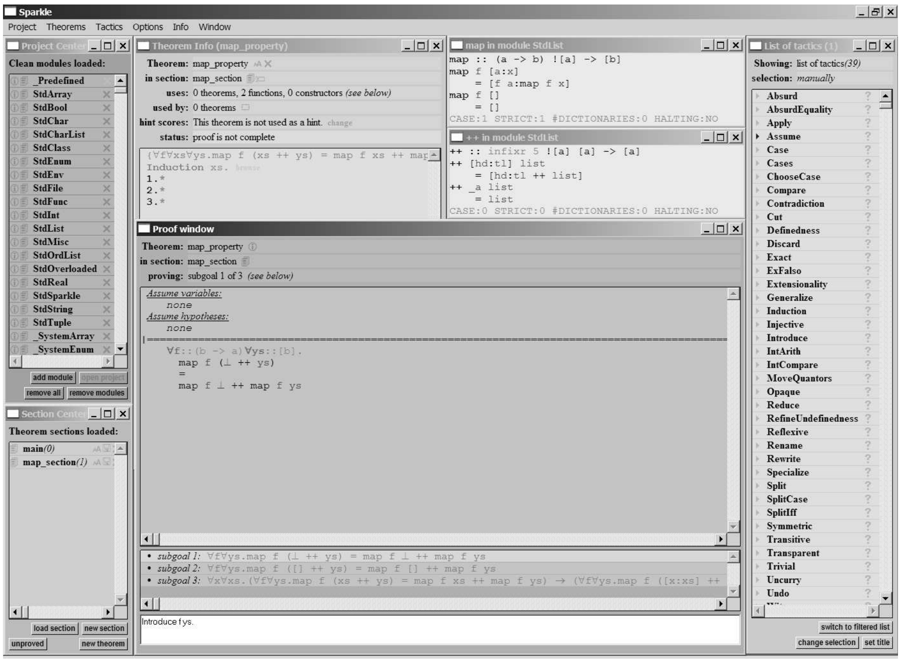
There are again redexes present in the current goal, because both `map` and `++` have patterns that match on the empty list `[]`. Therefore: ◀`Reduce NF All.`▶.

$$\boxed{\dfrac{f :: \mathtt{b} \rightarrow \mathtt{a},\ ys :: \mathtt{[b]}}{\mathtt{[]} = \mathtt{[]}}} \qquad (1.2'')$$

This is another example of a reflexive equality; therefore ◀`Reflexive.`▶.

$$\boxed{\begin{array}{l} \qquad\qquad\qquad\qquad\qquad\qquad - \\ \hline \forall_x \forall_{xs}. \\ \quad (\forall_f \forall_{ys}.\mathtt{map}\ f\ (xs\ \mathtt{++}\ ys) = \mathtt{map}\ f\ xs\ \mathtt{++}\ \mathtt{map}\ f\ ys) \\ \quad \rightarrow (\forall_f \forall_{ys}.\mathtt{map}\ f\ (\mathtt{[}x\mathtt{:}xs\mathtt{]}\ \mathtt{++}\ ys) = \mathtt{map}\ f\ \mathtt{[}x\mathtt{:}xs\mathtt{]}\ \mathtt{++}\ \mathtt{map}\ f\ ys) \end{array}} \qquad (1.3)$$

This is the third goal created by induction for the case that *xs* is a composed list. The current goal looks quite complicated, but introduction can make things a lot clearer. Here, we will not only introduce variables from universal quantors, but we will also introduce hypotheses from implications. This can be performed in one go with ◀`Introduce x xs IH f ys.`▶.



Theorem: map_property
in section: map_section
uses: 0 theorems, 2 functions, 0 constructors *(see below)*
used by: 0 theorems
hint scores: This theorem is not used as a hint. change
status: proof is not complete

```
(∀f∀xs∀ys.map f (xs ++ ys) = map f xs ++ map
Induction xs. browse
1.*
2.*
3.*
```

```
map :: (a -> b) ![a] -> [b]
map f [a:x]
    = [f a:map f x]
map f []
    = []
CASE:1 STRICT:1 #DICTIONARIES:0 HALTING:NO
```

```
++ :: infixr 5 ![a] [a] -> [a]
++ [hd:tl] list
    = [hd:tl ++ list]
++ _a list
    = list
CASE:0 STRICT:0 #DICTIONARIES:0 HALTING:NO
```

Theorem: map_property
in section: map_section
proving: subgoal 1 of 3 *(see below)*

```
Assume variables:
    none
Assume hypotheses:
    none
|==========================================================
   ∀f::(b -> a)∀ys::[b].
     map f (⊥ ++ ys)
     =
     map f ⊥ ++ map f ys
```

- *subgoal 1:* `∀f∀ys.map f (⊥ ++ ys) = map f ⊥ ++ map f ys`
- *subgoal 2:* `∀f∀ys.map f ([] ++ ys) = map f [] ++ map f ys`
- *subgoal 3:* `∀x∀xs.(∀f∀ys.map f (xs ++ ys) = map f xs ++ map f ys) → (∀f∀ys.map f ([x:xs] ++`

```
Introduce f ys.
```

Fig. 3. Screen shot of SPARKLE at proof state (1.1)

$$\boxed{\begin{array}{c} x :: \texttt{b},\ xs :: \texttt{[b]},\ f :: \texttt{b} \rightarrow \texttt{a},\ ys :: \texttt{[b]} \\ \underline{IH : \forall_f \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys} \\ \texttt{map}\ f\ (\texttt{[}x\texttt{:}xs\texttt{]}\ \texttt{++}\ ys) = \texttt{map}\ f\ \texttt{[}x\texttt{:}xs\texttt{]}\ \texttt{++}\ \texttt{map}\ f\ ys \end{array}} \quad (1.3')$$

Again, the current goal contains redexes, because `map` and `++` have patterns that match on constructed lists of the form [x:xs]. Therefore, ◀`Reduce NF All.`▶.

$$\boxed{\begin{array}{c} x :: \texttt{b},\ xs :: \texttt{[b]},\ f :: \texttt{b} \rightarrow \texttt{a},\ ys :: \texttt{[b]} \\ \underline{IH : \forall_f \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys} \\ \texttt{[}f\ x\texttt{:map}\ f\ (xs\ \texttt{++}\ ys)\texttt{]} = \texttt{[}f\ x\texttt{:map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys\texttt{]} \end{array}} \quad (1.3'')$$

The current proposition is now of the form `[X:Y] = [X:Z]`. Using the automatic injectivity of all *lazy* data constructors in Clean, we can simplify this to `X = X` ∧ `Y = Z`. Therefore, ◀`Injective.`▶.

Assignment 19. *(Injectivity and strictness)*
(a) Why does injectivity not hold for strict data constructors?

$$\boxed{\begin{array}{c} x :: \texttt{b},\ xs :: \texttt{[b]},\ f :: \texttt{b} \rightarrow \texttt{a},\ ys :: \texttt{[b]} \\ \underline{IH : \forall_f \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys} \\ f\ x = f\ x \wedge \texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys \end{array}} \quad (1.3''')$$

The current proposition is now of the form `P` ∧ `Q`, and can obviously be split into subgoals `P` and `Q`. Therefore, ◀`Split.`▶, which creates subgoals 1.3.1 and 1.3.2.

$$\boxed{\begin{array}{c} x :: \texttt{b},\ xs :: \texttt{[b]},\ f :: \texttt{b} \rightarrow \texttt{a},\ ys :: \texttt{[b]} \\ \underline{IH : \forall_f \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys} \\ f\ x = f\ x \end{array}} \quad (1.3.1)$$

This is a reflexive equality that can be proved immediately with ◀`Reflexive.`▶.

$$\boxed{\begin{array}{c} x :: \texttt{b},\ xs :: \texttt{[b]},\ f :: \texttt{b} \rightarrow \texttt{a},\ ys :: \texttt{[b]} \\ \underline{IH : \forall_f \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys} \\ \texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys \end{array}} \quad (1.3.2)$$

The current proposition is now an instantiation of the induction hypothesis *IH*. It can therefore be proved immediately by applying *IH* with ◀`Apply IH.`▶.

$$\boxed{\text{Q.E.D.}}$$

There are no more subgoals, which means that the proof is complete!

Assignment 20. *(Manual proof of the* `map` *property)*
(a) Prove the `map` property again, using the tactic dialogs only.
(b) Prove the `map` property again, using the command interface only.
(*Hint:* ◀`Reduce.`▶ abbreviates ◀`Reduce NF All.`▶, and ◀`Intros.`▶ is a variant of introduction that comes up with suitable names on its own)

(c) The automatic proof consists of the application of 13 tactics. It is possible to prove the property in less steps (our shortest proof consists of 9 steps). Try to shorten the proof yourself.

Assignment 21. *(More small proofs)*
Try to prove the following properties, preferably without the hint mechanism:
(a) $\forall_{xs}\forall_{ys}\forall_{zs}.xs$ `++` $(ys$ `++` $zs) = (xs$ `++` $ys)$ `++` $zs.$
(b) $\forall_{xs}.\neg(xs = \perp) \rightarrow \neg(xs =$ `[]`$) \rightarrow$ `[hd` xs`:tl` xs`]` $= xs.$
(c) $\forall_{n}\forall_{xs}.\neg(n = \perp) \rightarrow$ `take` $n\ xs$ `++` `drop` $n\ xs = xs.$
(d) $\forall_{P}\forall_{Q}.(\neg P \leftrightarrow Q) \leftrightarrow (P \leftrightarrow \neg Q).$

5 Tutorial Part II: Specialized Features of SPARKLE

In this section, the tutorial will be continued with advanced information about the dedicated use of SPARKLE in practice, and the features that are specialized for reasoning about CLEAN will be described. The same explanatory style will be used as in part I of the tutorial, and various assignments will again be included.

First, in Section 5.1 the importance of sharing in proofs will be explained. Then, the specification of definedness conditions in properties will be described in Section 5.2. The specialized behavior of four tactics will be introduced next; for 'Extensionality' in Section 5.3, for 'Induction' in Section 5.4, for 'Definedness' in Section 5.5, and for 'Reduce' in Section 5.6. Finally, the specification of properties by means of CLEAN-functions will be discussed in Section 5.7.

5.1 The Influence of Sharing on Reasoning

Sharing is important for the efficiency of functional programs. In CLEAN sharing is explicit, because for every construct it is precisely defined what is shared and what is not shared [29]. The semantics of CLEAN are based on graph rewriting [2,3,27]. This means that during reduction of the `Start` expression to its result, sharing is maintained as much as possible.

In SPARKLE, reduction may be used at many points in proofs as well. This reduction should behave in a semantically equivalent way to reduction in CLEAN, but it does not have to be exactly the same. Note that reduction in SPARKLE is symbolic, because it may encounter free variables that are introduced by logic quantors. In CLEAN, reduction only operates on closed expressions.

Sharing has no influence on semantics, and reduction in SPARKLE is free to either preserve or break it. Currently, the following strategy is realized:

– *Within* the application of reduction sharing is always preserved;
– But *afterwards* sharing is always automatically broken.

The idea behind this strategy is twofold. Firstly, efficiency is important in proofs too, therefore sharing is preserved within reduction. Secondly, after full reduction sharing is often not meaningful anymore and only hinders reduction, therefore it is automatically broken.

Assignment 22. *(The effect of sharing during reduction in proofs)*

(a) Consider in SPARKLE the trivial theorem (`let` n = 1+2+3 `in` n+n) = 12. Prove it using ◀`Reduce NF All.`▶, followed by ◀`Reflexive.`▶.

(b) Undo the proof with `Ctrl-Z` and prove the theorem again, this time using reduction with a fixed number of steps (◀`Reduce 4.`▶).

(c) Undo the proof with `Ctrl-Z` and prove the theorem again, this time using repeated single-step reduction (◀`Reduce 1.`▶).

(d) Explain why more reduction steps are needed in (c) than in (b).

Unfortunately, SPARKLE's current strategy for handling sharing is not optimal. The main problem is that all meaningful sharing, such as for instance recursion that has been expressed by means of cyclic lets, cannot be dealt with at all. Moreover, the current behavior is not very intuitive, as was already demonstrated in the assignment above.

The way sharing is handled in SPARKLE is currently being fixed according to the reduction mechanism described in [12]. In the next release, SPARKLE will always preserve all sharing, and manual reasoning steps will be added that allow users to manipulate, and possibly break, shared expressions at will.

5.2 Definedness Conditions in Properties

SPARKLE makes use of a *total* semantics in which undefinedness is taken into consideration explicitly. This has two consequences for the property language. Firstly, expressions are only equal if they either produce the same defined value, or both produce undefinedness. Secondly, the undefined value $\perp$ is a member of any type, and therefore a valid instantiation of any quantor.

In order to specify properties of CLEAN-programs correctly, one therefore has to know precisely how they behave in case some of their input becomes undefined. This behavior is determined by the lazy rewriting semantics of CLEAN, of which a thorough understanding is required for formal reasoning. Below we present a small example to illustrate the propagation of $\perp$-values through expressions. For a full explanation of computation in CLEAN we refer to [29] and [33].

Example. Consider the following definition of the well-known function `take`:

```
|  take n []     = []
|  take n [x:xs] = if (n>0) [x: take (n-1) xs] []
```

In CLEAN, patterns are evaluated from top to bottom, and right-hand-sides are only evaluated when their pattern matches. Consequently:

- `take` n $\perp = \perp$ for all n, because the first pattern always causes $\perp$ to be matched against `[]`, which fails;
- `take` $\perp$ `[]` = `[]`, because the successful match of the first pattern does not require $\perp$ to be evaluated;
- `take` $\perp$ `[`x`:`xs`]` $= \perp$ for all x and xs, because the second pattern matches, and its right-hand-side requires the computation of $\perp$ `> 0`, which fails.

It is very important that the starting point of formal reasoning is a logically correct property. Therefore, the specification of properties must always involve an analysis of behavior in the undefined case. In some cases, the property turns out to hold automatically for the undefined value, and nothing has to be changed. In other cases, however, the property actually turns out to be false:

Example. Consider the following intuitively true property of `drop` and `take`:

| $\forall_n \forall_{xs}.\texttt{take}\ n\ xs\ \texttt{++}\ \texttt{drop}\ n\ xs = xs.$

This property is falsified by the case $n = \perp$, because then the left-hand-side may become undefined, while the right-hand-side remains xs:

- Assume $xs =$ `[1]`. Then the left-hand-side reduces to $\perp$, as follows:
  `take` $\perp$ `[1] ++ drop` $\perp$ `[1]` $= \perp$ `++ drop` $\perp$ `[1]` $= \perp$.
  But the right-hand-side is `[1]`, which is defined.

Assignment 23. *(More definedness analysis)*

(a) The example property $\forall_n \forall_{xs}.\texttt{take}\ n\ xs\ \texttt{++}\ \texttt{drop}\ n\ xs = xs$ is *not* falsified in the case that $xs = \perp \wedge n \neq \perp$. Argue why this is the case.
(*Hint*: distinguish between $n = 0$ and $n \neq 0$.)

(b) Is the property $\forall_f \forall_{xs} \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = (\texttt{map}\ f\ xs)\ \texttt{++}\ (\texttt{map}\ f\ ys)$ falsified in the undefined case? If so, give example values for f, xs and ys that break the property. If not, argue why.
(*Hint:* see also Section 4.6.)

If definedness analysis shows that a property is falsified by a set of variable values V, then it can be rectified simply by adding conditions that exclude V. These *definedness conditions* are often simple and of the form '$n \neq \perp$', but they can also be more intricate (see Section 5.7).

Rectified Example: The `take-drop` property can be corrected by means of:

| $\forall_n \forall_{xs}.n \neq \perp \rightarrow \texttt{take}\ n\ xs\ \texttt{++}\ \texttt{drop}\ n\ xs = xs.$

Assignment 24. *(Proving the rectified* `take-drop` *example)*

(a) In Sparkle, prove $\forall_n \forall_{xs}.n \neq \perp \rightarrow \texttt{take}\ n\ xs\ \texttt{++}\ \texttt{drop}\ n\ xs = xs.$

Finally, note that Clean supports *strictness annotations*, with which the strict evaluation of certain expressions can be enforced explicitly. These annotations are often placed without much thought with the purpose of improving efficiency. However, strictness annotations change the definedness behavior of the program, and have an effect on properties and reasoning as well. In the context of formal reasoning, they should therefore only be used with care.

The precise effect of strictness annotations on properties is difficult to predict. Adding a strictness annotation can either: **(1)** not change a property at all; or **(2)** falsify a property, requiring additional definedness conditions to be formulated; or **(3)** allow existing definedness conditions to be removed. The third effect in particular is rather surprising.

Example of (1). Consider the following property:

| $\forall_{xs} \forall_{ys} \forall_{zs}.(xs\ \texttt{++}\ ys)\ \texttt{++}\ zs = xs\ \texttt{++}\ (ys\ \texttt{++}\ zs)$

This property holds for the standard definition of `++`, which is strict in its first argument only. Adding strictness to the second argument does not effect the property, however; it remains valid in the strict case as well.

Example of (2). Consider the following property:

| $\forall_{f,g}\forall_{xs}.\texttt{map}\ (f\ \texttt{o}\ g)\ xs = \texttt{map}\ f\ (\texttt{map}\ g\ xs)$

This property is valid for lazy lists, but invalid for element-strict lists.
Suppose $xs = \texttt{[12]}$, $g\ 12 = \bot$ and $f\ (g\ 12) = 7$.
Then $\texttt{map}\ (f\ \texttt{o}\ g)\ xs = \texttt{[7]}$, both in the lazy and in the strict case.
However, $\texttt{map}\ f\ (\texttt{map}\ g\ xs) = \texttt{[7]}$ in the lazy case, but $\bot$ in the strict case.
The property can be adapted to element-strict lists by explicitly enforcing that g produces a defined result for all elements x of xs:

| $\forall_{f,g,xs}.(\forall_{x \in xs}.g\ x \neq \bot) \rightarrow \texttt{map}\ (f\ \texttt{o}\ g)\ xs = \texttt{map}\ f\ (\texttt{map}\ g\ xs)$.

Example of (3). Consider the following property:

| $\forall_{xs}.\mathit{finite}\ xs \rightarrow \texttt{reverse}\ (\texttt{reverse}\ xs) = xs$

This property is valid both for lazy lists and for spine-strict lists.
The condition *finite xs*, however, is satisfied automatically for spine-strict lists, because spine-strict lists can never be infinite. In the spine-strict case, the property can therefore safely be reformulated (or, rather, optimized) by removing the *finite xs* condition:

| $\forall_{xs}.\texttt{reverse}\ (\texttt{reverse}\ xs) = xs$

Note that without the condition, the property is invalid in the lazy case: just choose any infinite list for *xs*.

5.3 Specialized Behavior of Extensionality

The property of *extensionality*, which states that two functions are equal iff they produce the same result for all possible arguments, is often considered to be universal. Unfortunately, there is a (rather obscure) example of two functions for which the property of extensionality does not hold unconditionally in the context of lazy evaluation:

```
H :: a -> b                F :: (a -> b)
H x = H x                  F = F
```

In the definitions above, H is a function of arity 1 that only reduces (to itself) when it is given an argument. F on the other hand is a function of arity 0 that always reduces to itself, regardless of whether it is applied or not. Obviously, $\texttt{F}\ x = \texttt{H}\ x$ now holds for all x, because they both reduce to themselves and are therefore both undefined.

Surprisingly, the property $\texttt{F} = \texttt{H}$ does not hold, because `H` is defined (it is a partial function application, and is thus in head normal form), while the meaning of `F` is undefined. It is therefore not safe to replace `H` by `F` (nor `F` by `H`); such a replacement could namely change the termination behavior of the program.

Fortunately, the problem can be corrected by weakening the property of extensionality as follows:

Definition 5.3.1. *(Revised version of extensionality)*
$\forall_f \forall_g.(f = \bot \leftrightarrow g = \bot) \rightarrow (\forall_x.f\ x = g\ x) \rightarrow f = g$

This revised version of extensionality is correct in the context of CLEAN. It can not be applied to prove $F = H$, because the condition $F = \bot \leftrightarrow H = \bot$ does not hold. SPARKLE defines a reasoning step for extensionality that makes use of the correct behavior.

Assignment 25. *(Extensionality)*
(a) Prove using extensionality that `sum` $\circ$ (`map` (`const` 1)) = `length` holds.

5.4 Specialized Behavior of Induction

An important reasoning step for dealing with recursive functions over algebraic datatypes is *structural induction.* Although induction is not always applicable, it is extremely useful in the context of functional programming, because it can be used successfully on many common data structures (such as for instance lists) and on many common kinds of recursive functions (such as for instance those defined by recursion on the results of pattern matching).

In order to deal with lazy evaluation, induction has to be customized in two different ways. Firstly, an extra base step is required for the undefined value $\bot$. Because $\bot$ is a member of each type, it must namely be treated as a constructor with no arguments. This behavior of induction is actually quite intuitive; for instance, if we want to prove $\forall_{x \in [A]}.P(x)$ with induction on the list structure, we would get the following proof obligations:

– $P(\bot)$;
– $P(\texttt{[]})$;
– $\forall_{x \in A} \forall_{xs \in \texttt{[}A\texttt{]}}.P(xs) \rightarrow P(\texttt{[}x\texttt{:}xs\texttt{]})$

Note that without the case for undefinedness it is possible to prove properties that are not true. For instance, we could easily prove that every lazy list is finite: the empty list is finite, and the extension of a finite list with a single element is always finite as well. The undefined list, on the other hand, is not finite!

The second customization of induction extends it to infinite structures as well. Because an infinite structure does not end with a base case, the induction principle is in general not applicable to it. In [25], however, Paulson has shown that the results of induction may be applied to infinite structures as long as the induction predicate satisfies the criterion of *admissibility.* We claim that Paulson's results may be applied to the context of CLEAN as well.

The admissibility criterion can be lifted to lazy functional languages easily. The basic idea is that equalities on negative positions (behind a negation) within a proposition must be decidable. An equality on type α is decidable if all possible expressions of type α are finite. This can be approximated statically: if α does not contain any recursion, then all its members are certainly finite. An equality on `Bool` is for instance decidable, but an equality on lists is not.

Definition 5.4.1. *(Finite types)*
A type α is *finite* if the set E of all possible expressions of type α is finite.

Definition 5.4.2. *(Decidable equalities)*
An equality between values of type α is *decidable* if α is finite.
We will denote this (informally) with $Decidable(=)$.

Definition 5.4.3. *(Admissibility)*
A proposition P is admissible if $Adm(+1, P)$ holds, by means of:

$$\begin{array}{ll}
Adm(sign,\ True) & = True \\
Adm(sign,\ False) & = True \\
Adm(sign,\ \neg P) & = Adm(-sign, P) \\
Adm(sign,\ P \wedge Q) & = Adm(sign, P) \wedge Adm(sign, Q) \\
Adm(sign,\ P \vee Q) & = Adm(sign, P) \wedge Adm(sign, Q) \\
Adm(sign,\ P \rightarrow Q) & = Adm(-sign, P) \wedge Adm(sign, Q) \\
Adm(sign,\ P \leftrightarrow Q) & = Adm(sign, P \rightarrow Q) \wedge Adm(sign, Q \rightarrow P) \\
Adm(sign,\ \forall.P) & = Adm(sign, P) \\
Adm(sign,\ \exists.P) & = Adm(sign, P) \\
Adm(sign,\ E_1 = E_2) & = Decidable(=) \vee sign = +1
\end{array}$$

Assignment 26. *(Induction on lazy lists)*
For each of the theorems below: prove it or show that it is not admissible.
(a) $\forall_{xs}.\texttt{finite}\ xs \rightarrow \texttt{take}\ (\texttt{length}\ xs)\ xs = xs$
(b) $\forall_{xs}.xs = \texttt{ones} \rightarrow \texttt{finite}\ xs$
(c) $\forall_{xs}\forall_{f}\forall_{p}.\texttt{all}\ p\ (\texttt{map}\ f\ xs) = \texttt{all}\ (p\ \texttt{o}\ f)\ xs$
(d) $\forall_{xs\in\texttt{[a]}}\forall_{ys\in\texttt{[a]}}.xs = ys \rightarrow xs\ \texttt{==}\ ys$

In order to reason about non-admissible predicates and/or non-inductive types several techniques have been developed. The most renowned of them are the take lemma and its improved version the approximation lemma [4] on one hand, and the class of techniques concerning co-induction based on bisimilarity[17] on the other hand. To treat them in further detail is outside the scope of this paper.

5.5 Definedness Analysis and the Special 'Definedness' Tactic

A consequence of the specialized behavior described in Sections 5.2-5.4 is that reasoning in Sparkle often involves properties of the form $E = \perp$ or $E \neq \perp$. Dealing with definedness is cumbersome, and should therefore be supported as much as possible. For this purpose, Sparkle derives definedness information automatically, and offers specialized tactics that make use of this information.

Definedness analysis is the process of deriving definedness information. It is carried out automatically by Sparkle each time a new goal is constructed. The results of definedness analysis are sets D and U, which contain expressions that have been determined to be defined and undefined respectively. The sets D and U are stored with each goal and can be used by various tactics.

The process of definedness analysis starts by assigning all occurring basic values to D and $\perp$ to U. It then repeatedly extends D and U by examining the hypotheses that have been introduced, and by making use of *strictness* and *totality* properties. The following derivation rules are used for this purpose:

- *Definedness by hypothesis equality.*
  If a hypothesis $E_0 = E_1$ is available, and $E_i \in D$, then add E_{1-i} to D.
  If a hypothesis $E_0 = E_1$ is available, and $E_i \in U$, then add E_{1-i} to U.
  If a hypothesis $E_0 \neq E_1$ is available, and $E_i \in U$, then add E_{1-i} to D.
- *Constructor definedness.*
  Assume that C is a constructor of arity n with strict arguments $S \subseteq \{1 \ldots n\}$.
  If the application $A = (C\ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid i \in S\} \subseteq D$, then add A to D.
  If the application $A = (C\ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid i \in S\} \cap U \neq \emptyset$, then add A to U.
- *Total function definedness.*
  Assume that F is a function of arity n which is known to be total.
  If the application $A = (F\ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid 1 \leq i \leq n\} \subseteq D$, then add A to D.
  If the application $A = (F\ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid 1 \leq i \leq n\} \cap U \neq \emptyset$, then add A to U.
- *Normal function definedness.*
  Assume that F is a function of arity n with strict arguments $S \subseteq \{1 \ldots n\}$.
  If the application $A = (F\ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid i \in S\} \cap U \neq \emptyset$, then add A to U.

Note that the strictness information for the definedness analysis is available explicitly in the source program, whereas the totality information is assumed to be made available externally (in SPARKLE, many functions from `StdEnv` are hard-coded to be total). Furthermore, to maximize the effectiveness of the definedness analysis, the negation of the current goal is treated as a hypothesis as well.

An important tactic that makes use of definedness analysis is 'Definedness'. It immediately proves any goal that contains contradictory definedness, which is the case if D and U overlap. Note that because the negation of the current goal is treated as a hypothesis, it also proves any goal in which the definedness information implies the validity of the to prove. Although the rules of definedness analysis are relatively simple, it is surprisingly powerful. The SPARKLE-tactic 'Definedness' is therefore extremely useful, and can be applied often in proofs.

Assignment 27. *(Using the Definedness-tactic)*
Prove each of the following properties in SPARKLE with the Definedness-tactic.
(a) $\forall_f \forall_{xs}.\neg(\texttt{map}\ f\ xs = \bot) \rightarrow \neg(xs = \bot)$
(b) $\forall_n.\texttt{eval}\ (n\ \texttt{+}\ 12) \rightarrow \neg(n = \bot)$
(see Section 5.7 for an introduction of the `eval` function)
(c) $\forall_n \forall_m.(n\ \texttt{/}\ m = 42) \rightarrow \neg(n\ \texttt{+}\ m = \bot)$
(d) $\forall_n.(7\ \texttt{+}\ (12\ \texttt{*}\ (13\ \texttt{-}\ n)) = \bot) \rightarrow n = \bot$

More examples of the use of definedness can be found in [33].

5.6 Specialized Behavior of Reduction

Because of the presence of logic variables that are introduced by quantors on the property level, reduction in Sparkle is *symbolic*. A logic variable may be instantiated with an arbitrary well-typed expression, and its evaluation does not yield anything. Assuming termination, it is therefore no longer possible to reduce every expression to either a weak head normal form or to $\perp$.

It is important that reduction in Sparkle carries on as far as possible. For this purpose, Sparkle realizes two extensions in its reduction mechanism that allow reduction to continue, even when a logic variable is encountered on a strict position.The first extension involves ignoring unnecessary strictness annotations; the second extension involves using the results of definedness analysis.

The idea of the first extension is that some strictness annotations can safely be removed without changing the semantics of the program. To illustrate this, take a look at the following three Clean-functions:

```
id :: !a -> a      K :: !a !b -> a    length :: ![a] -> Int
id x = x           K x y = x          length [x:xs] = 1+length xs
                                      length []     = 0
```

An exclamation mark before the type of an argument indicates strictness. During evaluation, the strict arguments of a function will always be reduced to weak head normal form *before* the function is expanded, whereas the non-strict arguments will not. A strictness annotation always changes the reduction behavior of the program; however, it does not always change the semantics.

The strictness annotation in the function `id` does not change the semantics, because the evaluation of its body immediately requires the evaluation of its argument anyway. The same goes for the `length` function, because the pattern match enforces evaluation. In the function `K`, the first strictness annotation does not change the semantics, but the second one does. In fact, removing the second annotation would cause `K` $x \perp = x$, where in the current situation `K` $x \perp = \perp$.

The reduction system of Sparkle is able to recognize the different kinds of strictness annotations. In case a strict function argument is encountered like in `id` or in `K` (first annotation), it will be reduced first, but the function will *always* be expanded afterwards. This is different from reduction in Clean, but semantically sound, and much more user friendly for reasoning (not expanding '`id x`' would be really inconvenient). The behavior of Sparkle on annotations as in `K` (second annotation) is of course not changed, because that would be semantically unsound. The behavior on annotations as in `length` is not changed either, because the pending pattern match requires its argument to be reduced. Expanding the function therefore does not make much sense, because reduction would be stopped by the pattern match anyway.

Assignment 28. *(Reduction in* Sparkle *(1))*
(a) Build a Clean-module with the functions above and load it into Sparkle.
(b) Prove $\forall_x$.`id` $x = x$
(c) Prove $\forall_x$.`K` x $12 = x$
(d) Attempt to prove $\forall_x$.`K` 12 $x = 12$. Why does this property not hold?

The second extension of reduction is very straightforward: simply make use of the results of the definedness analysis. In case SPARKLE encounters a function argument whose strictness cannot be removed safely, and on which no pattern match is performed, then the function is allowed to be expanded anyway, as long as the argument expression is an element of D. Again, the argument will be reduced as much as possible first. The second extension allows users to influence the reduction mechanism by means of specifying (and later proving) additional definedness properties.

Assignment 29. *(Reduction in* SPARKLE *(2))*
(a) Prove $\forall_x \forall_y . \neg(y = \perp) \rightarrow \texttt{K}\ x\ y = x$

5.7 Property Specification in CLEAN

The property language of SPARKLE is a simple first-order proposition logic only, in which predicates and relations cannot be expressed. However, the possibility to define higher-order functions in the programming language and use them as boolean predicates gives unexpected expressive power. The higher-order of the programming language can be combined with SPARKLE's first order logic.

A good example of a boolean predicate in CLEAN is the function `eval`. The purpose of `eval` is to *fully* reduce its argument and return `True` afterwards. Such an 'eval' function is usually used to express evaluation strategies in the context of parallelism [6,32]. We use `eval` for expressing definedness conditions.

In the module `StdSparkle` of SPARKLE's standard environment, the function `eval` is defined by means of overloading. The instance on `Char` is defined by:

```
class eval a :: !a -> Bool

instance eval Char
where    eval :: !Char -> Bool
         eval x = True
```

In a logical property, (`eval` x = `True`) can now be used as a manual definedness condition. The meaning of this condition is identical to $\neg(x = \perp)$, because:

- If $x = \perp$, then (`eval` x) = (`eval` $\perp$) $= \perp$ on the semantic level, because `eval` is strict in its argument. Therefore, `eval` x = `True` is not satisfied.
- If $x \neq \perp$, then x must be equal to some defined basic character b. Therefore, (`eval` x) = (`eval` b) = `True` on the semantic level.
- Note that `eval` is defined in such a way that it is *never* equal to `False`.

On characters, `eval` is not so interesting. However, by means of overloading, it can easily be defined for lists, and all other kinds of data structures as well. The overloading is used to assume the presence of an `eval` on the element type:

```
instance eval [a] | eval a
where    eval :: ![a] -> Bool | eval a
         eval [x:xs] = eval x && eval xs
         eval []     = True
```

This instance of `eval` fully evaluates both the spine of the list and all its elements, and only returns `True` if this succeeds. It can therefore be used to express the intricate definedness condition that a list is finite and contains defined elements only. This condition cannot be expressed on the property level at all.

Assignment 30. *(Proofs of properties that use* `eval`*)*
Using the function `eval` from `StdSparkle`, prove the following properties:
(a) $\forall_x \forall_{xs}.\texttt{eval}\ xs \rightarrow \texttt{isMember}\ x\ xs \rightarrow \texttt{eval}\ x$
(b) $\forall_{xs}.\texttt{eval}\ xs \rightarrow \texttt{sum}\ (\texttt{map}\ (\texttt{K}\ 1)\ xs) = \texttt{length}\ xs$
(using the strict version of function `K`, see assignment 28)
(c) $\forall_x \forall_p \forall_{xs}.\texttt{eval}\ x \rightarrow \texttt{eval}\ xs \rightarrow \texttt{eval}\ (\texttt{map}\ p\ xs) \rightarrow$
$\texttt{isMember}\ x\ (\texttt{filter}\ p\ xs) = \texttt{isMember}\ x\ xs\ \texttt{\&\&}\ p\ x$

All instances of `eval` have to share certain properties. To prove properties of *all* members of a certain type class, the recently added tool support for general type classes can be used [21]. With this tool, the properties $\forall_x.\texttt{eval}\ x \rightarrow x \neq \bot$ and $\forall_x.\texttt{eval}\ x \neq \texttt{False}$ can be stated and proven in SPARKLE.

A useful variation of `eval` on lists is the function that evaluates the spine of the list only, but leaves the elements alone. This function expresses the condition that a list is finite. It is defined in `StdSparkle` as follows:

```
finite :: ![a] -> Bool
finite [x:xs]     = finite xs
finite []         = True
```

The boolean predicate `finite` allows several useful properties to be stated and proven in SPARKLE:

Assignment 31. *(Proofs of properties that use* `finite`*)*
Using the function `finite` from `StdSparkle`, prove the following properties:
(a) $\forall_{xs}.\texttt{finite}\ xs \rightarrow \texttt{length}\ xs \geq 0$
(b) $\forall_{xs}.\texttt{finite}\ xs \rightarrow \texttt{finite}\ (\texttt{reverse}\ xs)$
(c) $\forall_{xs}.\texttt{finite}\ xs \rightarrow \texttt{reverse}\ (\texttt{reverse}\ xs) = xs$

6 Related Work

Currently, well-known and widely used proof assistants are PVS [24], COQ [31] and ISABELLE [26]. They are all generic provers that are not tailored towards a specific programming language. It is very hard for programmers to reason in them, because they require using a different syntax and a different semantics. For instance, strictness annotations as in CLEAN are not supported by any existing proof assistant. On the other hand, these well established proof assistants offer features that are not available in SPARKLE. Most notably, the tactic language and the logic are much richer than in SPARKLE.

At Chalmers University of Technology, the proof assistant AGDA [1] has been developed in the context of the COVER [9] project. AGDA is dedicated to the lazy functional language HASKELL [20]. As in SPARKLE, the program is translated to a core-version on which the proofs are performed. Being geared towards facilitating the 'average' functional programmer, Sparkle offers dedicated tactics and a dedicated semantics based on graph rewriting. Agda uses standard constructive type theory on λ-terms, enabling independent proof checking.

Also as part of the Cover project, it is argued in [10] that "loose reasoning" is "morally correct", i.e. that the correctness of a theorem under the assumption that every subexpression is strict and terminating implies the correctness of the theorem in the lazy case under certain additional conditions. The conditions that are found in this way, however, may be too restrictive for the lazy case. Sparkle offers good support for reasoning with definedness conditions directly.

Another proof assistant dedicated to Haskell is Era [34], which stands for Equational Reasoning Assistant. This proof assistant builds on earlier work initiated by Andy Gill [15]. It is intended to be used for equational reasoning, and not for theorem proving in general. Additional proving methods, such as induction or logical steps, are not supported. Era is a stand-alone application. Unfortunately, it seems that work on this project has been discontinued for a while. Recently, Andy Gill took up the project again, producing a version with an Ajax based interface, under the name of Hera [16], short for Haskell Equational Reasoning Assistant.

In [22], a description is given of an automated proof tool which is dedicated to Haskell. It supports a subset of Haskell, and needs no guidance of users in the proving process. Induction is only applied when the corresponding quantor has been marked explicitly in advance. The user, however, cannot further influence the proving process at all, and cannot suggest tactics to help the prover in constructing the proof.

Another proof assistant that is dedicated to a functional language is Evt [23], the Erlang Verification Tool. However, Erlang differs from Clean, because it is a strict, untyped language which is mainly used for developing distributed applications. Evt has been applied in practice to larger examples.

The Programatica project of the Pacific Software Research Center in Oregon (www.cse.ogi.edu/PacSoft/projects/programatica) is another project that aims to integrate programming and reasoning. They intend to support a wide range of validation techniques for programs written in different languages. For functional languages they use P-logic, which is based on a modal μ-calculus in which undefinedness can also be expressed. In the Programatica project, properties are mixed with the Haskell source.

Properties about functional programs are proved by hand in many textbooks, for instance in [4]. Also, several articles (for instance [8]) make use of reasoning about functional programs. It seems worthwhile to attempt to formalize these proofs in Sparkle. In programming practice, however, reasoning about functional programs is scarcely used.

7 Conclusions

In this paper, we have presented a thorough description of the dedicated proof assistant Sparkle, which is integrated in the distribution of the lazy functional programming language Clean. We have introduced Sparkle in detail, both on the theoretical and on the practical level. On the theoretical level, we have explained the process of formal reasoning in general, and Sparkle's dedicated support for it in specific. On the practical level, we have provided an extensive tutorial of the actual use of Sparkle.

The tutorial not only covers the fundamental functionality of Sparkle, but also explains several of its advanced features that are specific for reasoning about lazy functional programs. Assignments are included at various points in the tutorial; they allow useful hands-on experience with Sparkle to be obtained in a guided way. After completion of the tutorial, anyone with a basic understanding of functional programming will be able to make effective use of Sparkle in practice, and will be able to prove small to medium properties with little effort.

Furthermore, we also hope to have sparked an interest in making use of formal reasoning to show important properties of functional programs. With the right tool support, this is already feasible for many smaller examples, and provides an enjoyable challenge for bigger programs too!

References

1. Abel, A., Benke, M., Bove, A., Hughes, J., Norell, U.: Verifying Haskell programs using constructive type theory. In: Leijen, D. (ed.) Proceedings of the ACM SIGPLAN 2005 Haskell Workshop, Tallinn, Estonia, pp. 62–74. ACM Press, New York (2005)
2. Barendregt, H.P., van Eekelen, M.C.J.D., Glauert, J.R.W., Kennaway, R., Plasmeijer, M.J., Sleep, M.R.: Term graph rewriting. In: de Bakker, J.W., Nijman, A.J., Treleaven, P.C. (eds.) PARLE 1987. LNCS, vol. 259, pp. 141–158. Springer, Heidelberg (1987)
3. Barendsen, E., Smetsers, S.: Graph rewriting aspects of functional programming. In: Handbook of Graph Grammars and Computing by Graph Transformation, pp. 63–102. World Scientific, Singapore (1999)
4. Bird, R.S.: Introduction to Functional Programming using Haskell, 2nd edn. Prentice-Hall, Englewood Cliffs (1998)
5. Brus, T.H., van Eekelen, M.C.J.D., van Leer, M.O., Plasmeijer, M.J.: Clean: A language for functional graph writing. In: Proceedings of the Functional Programming Languages and Computer Architecture, pp. 364–384. Springer, Heidelberg (1987)
6. Burn, G.L.: Evaluation transformers a model for the parallel evolution of functional languages. In: Proc. of a conference on Functional programming languages and computer architecture, pp. 446–470. Springer, Heidelberg (1987)
7. Butterfield, A., Strong, G.: Proving Correctness of Programs with I/O - a paradigm comparison. In: Arts, T., Mohnen, M. (eds.) IFL 2002. LNCS, vol. 2312, pp. 72–88. Springer, Heidelberg (2002)
8. Butterfield, A., Strong, G.: Proving correctness of programs with io - a paradigm comparison. In: Arts, T., Mohnen, M. (eds.) IFL 2002. LNCS, vol. 2312, pp. 72–87. Springer, Heidelberg (2002)
9. Coquand, T., Dybjer, P., Hughes, J., Sheeran, M.: Combining verification methods in software development. Project proposal, Chalmers Institute of Techology, Sweden (December 2001)

10. Danielsson, N.A., Hughes, J., Jansson, P., Gibbons, J.: Fast and loose reasoning is morally correct. SIGPLAN Not. 41(1), 206–217 (2006)
11. de Mol, M., van Eekelen, M., Plasmeijer, R.: Theorem proving for functional programmers - Sparkle: A functional theorem prover. In: Arts, T., Mohnen, M. (eds.) IFL 2002. LNCS, vol. 2312, pp. 55–72. Springer, Heidelberg (2002)
12. de Mol, M., van Eekelen, M., Plasmeijer, R.: A single-step term-graph reduction system for proof assistants. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) Proceedings of Selected and Invited Papers of Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, pp. 181–197 (2007)
13. de Mol, M., van Eekelen, M., Plasmeijer, R.: The Mathematical Foundation of the Proof Assistant Sparkle. Technical Report ICIS–R07025, Radboud University Nijmegen (November 2007)
14. Dowse, M., Butterfield, A., van Eekelen, M.C.J.D.: Reasoning About Deterministic Concurrent Functional I/O. In: Grelck, C., Huch, F., Michaelson, G., Trinder, P.W. (eds.) IFL 2004. LNCS, vol. 3474, pp. 177–194. Springer, Heidelberg (2004)
15. Gill, A.: The technology behind a graphical user interface for an equational reasoning assistant. In: Turner, D.N. (ed.) Functional Programming. Workshops in Computing, p. 4. Springer, Heidelberg (1995)
16. Gill, A.: Introducing the haskell equational reasoning assistant. In: Haskell 2006: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell, pp. 108–109. ACM Press, New York (2006)
17. Gordon, A.: Bisimilarity as a theory of functional programming. In: Proceedings of the Eleventh Conference on the Mathematical Foundations of Programming Semantics. Electronic Notes in Theoretical Computer Science, vol. 1. Elsevier Science B.V, Amsterdam (1995)
18. Gordon, M., Milner, R., Wadsworth, C.: Edinburgh LCF. LNCS, vol. 78. Springer, Berlin (1979)
19. Horváth, Z., Kozsik, T., Tejfel, M.: Proving invariants of functional programs. In: Kilpeläinen, P., Päivinen, N. (eds.) SPLST. Department of Computer Science, pp. 115–126. University of Kuopio (2003)
20. Hudak, P., Jones, S.L.P., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J.H., Guzmán, M.M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, R.B., Nikhil, R.S., Partain, W., Peterson, J.: Report on the Programming Language Haskell, A Non-strict, Purely Functional Language. SIGPLAN Notices 27(5), R1–R164 (1992)
21. van Kesteren, R., van Eekelen, M., de Mol, M.: Proof support for general type classes. In: Loidl, H.-W. (ed.) Trends in Functional Programming 5: Selected papers from the Fifth International Symposium on Trends in Functional Programming, TFP 2004, München, Germany, Intellect, pp. 1–16 (2004)
22. Mintchev, S.: Mechanized reasoning about functional programs. In: Hammond, K., Turner, D., Sansom, P. (eds.) Proceedings of the Glasgow Functional Progamming Workshop, pp. 151–167. Springer, Heidelberg (1994)
23. Noll, T., Fredlund, L., Gurov, D.: The evt erlang verification tool. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 582–585. Springer, Heidelberg (2001)
24. Owre, S., Shankar, N., Rushby, J., Stringer-Calvert, D.: PVS Language Reference (version 2.4) (2001), `http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf`
25. Paulson, L.C.: Logic and Computation. Cambridge University Press, Cambridge (1987)
26. Paulson, L.C.: The Isabelle Reference Manual. University of Cambridge (2007), `http://isabelle.in.tum.de/doc/ref.pdf`

27. Plasmeijer, R., van Eekelen, M.: Functional Programming and Parallel Graph Rewriting. Addison-Wesley Publishing Company, Reading (1993)
28. Plasmeijer, R., van Eekelen, M.: Keep it clean: a unique approach to functional programming. SIGPLAN Not. 34(6), 23–31 (1999)
29. Plasmeijer, R., van Eekelen, M.: Concurrent CLEAN Language Report (version 2.0) (December 2001), `http://www.cs.ru.nl/~clean/`
30. Tejfel, M., Horváth, Z., Kozsik, T.: Extending the sparkle core language with object abstraction. Acta Cybern. 17(2) (2006)
31. The Coq Development Team. The Coq Proof Assistant Reference Manual (version 7.3). Inria (2002), `http://pauillac.inria.fr/cdrom/www/coq/doc/main.html`
32. Trinder, P.W., Hammond, K., Loidl, H.-W., Jones, S.L.P.: Algorithm + strategy = parallelism. J. Funct. Program. 8(1), 23–60 (1998)
33. van Eekelen, M., de Mol, M.: Proof tool support for explicit strictness. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 37–54. Springer, Heidelberg (2006)
34. Winstanley, N.: Era User Manual, version 2.0. University of Glasgow (March 1998), `http://www.dcs.gla.ac.uk/~nww/Era/Era.html`

A Appendix: Short Description of all Sparkle Tactics

This appendix provides a short description of the tactics that can be used in Sparkle proofs. In total, Sparkle makes a library of 39 tactics available. In the description below, each tactic is briefly categorized as follows:

Equivalence/Strengthening - an equivalence tactic creates new goals that are logically equivalent to the original goal; a strengthening tactic creates goals that are logically stronger.
Forwards/Backwards - a forwards tactic brings hypotheses closer to the current goal; a backwards tactic brings the current goal closer to the hypotheses.
Instantaneous - an instantaneous tactic proves a goal in one single step (and will not be categorized as equivalence/strengthening or forwards/backwards).
Programming/Logic - a programming tactic is based on the semantics of Clean; a logic tactic is based on the semantics of the logical connectives.

Besides the type, for each tactic some *information* about its inner working is stated, and a small *example* is given of its use.

```
Absurd <Hyp1> <Hyp2>.
```

Type: Instantaneous; logic.
Info: Proves a goal that contains contradictory (absurd) hypotheses.
Details: Hypotheses are contradictory if they are each other's exact negation.
Example: $p, \langle \mathsf{H1}{:}\neg(p = 12)\rangle, \langle \mathsf{H2}{:}p = 12\rangle \vdash \texttt{FALSE}$
◀`Absurd H1 H2.`▶
Q.E.D.

```
AbsurdEquality <Hyp>.
```

Type: Instantaneous; programming.
Info: Proves a goal that contains a hypothesis stating an absurd equality.

Details: An equality between two different basic values is absurd, as well as an equality between applications of different lazy constructors.
Example: $\langle \mathsf{H1}{:}\texttt{True} = \texttt{False}\rangle \vdash \texttt{FALSE}$
◀`AbsurdEquality H1.`▶
Q.E.D.
Notes: `True` and `False` are constructors; `FALSE` is a constant proposition.

```
Apply <Fact>.
```

Type: Usually strengthening, depends on fact; backwards; logic.
Info: Applies a fact to the current goal.
Details: A fact is either an earlier proved theorem or an introduced hypothesis, and must be of the form $\forall_{x_1 \ldots x_n}.P_1 \rightarrow \ldots P_m \rightarrow Q$. It is only valid if $r_1 \ldots r_n$ can be found such that $Q[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}]$ equals the current goal. If this is the case, then the current goal is replaced with the conjunction $P_1[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}] \wedge \ldots \wedge P_m[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}]$.
Example: $p, \langle \mathsf{H1}{:}\forall_x \forall_y \forall_z.x > 0 \rightarrow y < z \rightarrow x + y < x + z\rangle \vdash 7 + p < 7 + 12$
◀`Apply H1.`▶
$p, \langle \mathsf{H1}{:}\forall_x \forall_y \forall_z.x > 0 \rightarrow y < z \rightarrow x + y < x + z\rangle \vdash 7 > 0 \wedge p < 12$
Notes: This tactic can also be applied in a forwards manner. In that case, P_1 must match on a hypothesis R, which is then replaced by $P_2 \rightarrow \ldots P_n \rightarrow Q$.

```
Assume <Prop>.
```

Type: Equivalence; forwards; logic.
Info: Assumes the validity of a manually stated proposition.
Details: Two goals are created: one with the assumption as new hypothesis, and one with the hypothesis as goal itself.
Example: $P, Q, R, \langle \mathsf{H1}{:}P \rightarrow R\rangle, \langle \mathsf{H2}{:}\neg P \rightarrow R\rangle \vdash R$
◀`Assume` $P \vee \neg P$.▶
(1) $P, Q, R, \langle \mathsf{H1}{:}P \rightarrow R\rangle, \langle \mathsf{H2}{:}\neg P \rightarrow R\rangle, \langle \mathsf{H3}{:}P \vee \neg P\rangle \vdash R$
(2) $P, Q, R, \langle \mathsf{H1}{:}P \rightarrow R\rangle, \langle \mathsf{H2}{:}\neg P \rightarrow R\rangle \vdash P \vee \neg P$
Notes: A name for the new hypothesis is generated automatically.

```
Case <Hyp>.
```

Type: Equivalence; backwards; logic.
Info: Breaks down an introduced disjunction.
Details: The hypothesis must be of the form $P \vee Q$. Two goals are created: one in which the hypothesis is replaced by P, and one in which it is replaced by Q.
Example: $P, Q, \langle \mathsf{H1}{:}P \vee \neg P\rangle, \langle \mathsf{H2}{:}P \rightarrow Q\rangle, \langle \mathsf{H3}{:}\neg P \rightarrow Q\rangle \vdash Q$
◀`Case H1.`▶
(1) $P, Q, \langle \mathsf{H1}{:}P\rangle, \langle \mathsf{H2}{:}P \rightarrow Q\rangle, \langle \mathsf{H3}{:}\neg P \rightarrow Q\rangle \vdash Q$
(2) $P, Q, \langle \mathsf{H1}{:}\neg P\rangle, \langle \mathsf{H2}{:}P \rightarrow Q\rangle, \langle \mathsf{H3}{:}\neg P \rightarrow Q\rangle \vdash Q$

```
Cases <Expr>.
```

Type: Equivalence; programming.
Info: Performs a case distinction on a given expression.
Details: The expression must be of an algebraic type. New goals are created

for each of its constructors, and one for $\perp$ as well. Each new goal is obtained by replacing *all* occurrences (also in the hypotheses) of the indicated expression with a generic application of the constructor.
Example: $xs, ys, \langle \mathsf{H1}{:}\mathtt{length}\ (xs\ \mathtt{++}\ ys) > 0\rangle \vdash \neg(xs\ \mathtt{++}\ ys = \mathtt{[]})$
◀`Cases (xs ++ ys).`▶
(1) $\langle \mathsf{H1}{:}\mathtt{length}\ \perp > 0\rangle \vdash \neg(\perp = \mathtt{[]})$
(2) $\langle \mathsf{H1}{:}\mathtt{length}\ \mathtt{[]} > 0\rangle \vdash \neg(\mathtt{[]} = \mathtt{[]})$
(3) $x_1, x_2, \langle \mathsf{H1}{:}\mathtt{length}\ \mathtt{[}x_1 {:} x_2\mathtt{]} > 0\rangle \vdash \neg(\mathtt{[}x_1 {:} x_2\mathtt{]} = \mathtt{[]})$
Notes: Names for the newly introduced variables are generated automatically.

`ChooseCase.`

Type: Equivalence; programming.
Info: Simplifies a case distinction in which only one pattern is valid.
Details: The goal must be of the form $E_1 = E_2$, where E_1 is a case distinction and E_2 is a basic value. A pattern is valid if its result is not statically unequal to E_2. The tactic succeeds only if there is exactly one valid pattern. The case is then simplified to the result of the single valid pattern, and its condition is introduced as a conjunction in the goal.
Example: $n \vdash \mathtt{case}\ n\ \mathtt{of}\ (7 \mapsto 13; 13 \mapsto 7; n \mapsto 11) = 13$
◀`ChooseCase.`▶
$n \vdash n = 7 \wedge 13 = 13$

`Compare <Expr1> with <Expr2>.`

Type: Equivalence; backwards; logic.
Info: Distinguishes between the possible compare results of two expressions.
Details: The expressions must both be of type `Int`. Five new goals are created; one for $E_1 = \perp$, one for $E_2 = \perp$, one for $E_1 < E_2$, one for $E_1 = E_2$ (provided that E_1 and E_2 are not $\perp$), and one for $E_2 < E_1$.
Example: $m, n \vdash \mathtt{min}\ m\ n \leq \mathtt{max}\ m\ n$
◀`Compare m with n.`▶
(1) $m, n \vdash m = \perp \rightarrow \mathtt{min}\ m\ n \leq \mathtt{max}\ m\ n$
(2) $m, n \vdash n = \perp \rightarrow \mathtt{min}\ m\ n \leq \mathtt{max}\ m\ n$
(3) $m, n \vdash m < n \rightarrow \mathtt{min}\ m\ n \leq \mathtt{max}\ m\ n$
(4) $m, n \vdash \neg(m = \perp) \rightarrow \neg(n = \perp) \rightarrow m = n \rightarrow \mathtt{min}\ m\ n \leq \mathtt{max}\ m\ n$
(5) $m, n \vdash n < m \rightarrow \mathtt{min}\ m\ n \leq \mathtt{max}\ m\ n$

`Contradiction.`

Type: Equivalence; backwards; logic.
Info: Builds a proof by contradiction.
Details: Replaces the current goal by the absurd proposition `FALSE` and adds its negation as a hypothesis in the context. If a double negation is produced, it will be removed automatically.
Example: $P, \langle \mathsf{H1}{:}P \rightarrow \mathtt{FALSE}\rangle \vdash \neg P$
◀`Contradiction.`▶
$P, \langle \mathsf{H1}{:}P \rightarrow \mathtt{FALSE}\rangle, \langle \mathsf{H2}{:}P\rangle \vdash \mathtt{FALSE}$
Notes: A name for the new hypothesis is generated automatically. This tactic

can also be applied in a forwards manner on a hypothesis. In that case, the negation of the hypothesis simply becomes the new goal to prove.

```
Cut <Fact>.
```

Type: Equivalence; backwards; logic.
Info: Duplicates a fact.
Details: A fact is either an earlier proved theorem or an introduced hypothesis. It is added to the to prove by means of a new implication.
Example: $\langle \mathsf{H1}{:}\forall_P.P \vee \neg P\rangle \vdash \texttt{FALSE}$
◀`Cut H1.`▶
$\langle \mathsf{H1}{:}\forall_P.P \vee \neg P\rangle \vdash (\forall_P.P \vee \neg P) \rightarrow \texttt{FALSE}$

```
Definedness.
```

Type: Instantaneous; logic.
Info: Uses contradictory definedness information to prove a goal.
Details: Two sets of expressions are determined: (1) those that are statically known to be *equal* to ⊥; (2) those that are statically known to be *unequal* to ⊥. These sets are determined by examining equalities in hypotheses and using strictness information. In addition, the totality of certain predefined functions is used. If an overlap between the two sets is found, the goal is proved immediately.
Example: $xs, ys, zs, \langle \mathsf{H1}{:}xs = \perp\rangle, \langle \mathsf{H2}{:}xs \texttt{ ++ } ys = \texttt{[}1 : zs\texttt{]}\rangle \vdash \texttt{FALSE}$
◀`Definedness.`▶
Q.E.D.
Notes: In the example, $xs = \perp$ due to H1, and $\neg(xs = \perp)$ due to the strictness of `++` and the definedness of the result of xs `++` ys by means of H2.

```
Discard <Hyp>.
```

Type: Strengthening; logic.
Info: Deletes an introduced hypothesis.
Example: $x, xs, \langle \mathsf{H1}{:}\texttt{reverse [] = []}\rangle \vdash \texttt{reverse [}x : xs\texttt{]} = \texttt{reverse } xs \texttt{ ++[}x\texttt{]}$
◀`Discard H1.`▶
$x, xs \vdash \texttt{reverse [}x : xs\texttt{]} = \texttt{reverse } xs \texttt{ ++[}x\texttt{]}$

```
Exact <Hyp>.
```

Type: Instantaneous; logic.
Info: Proves a goal that is identical to an introduced hypothesis.
Example: $\langle \mathsf{H1}{:}\forall_P\forall_Q.(P \wedge Q) \rightarrow P\rangle \vdash \forall_P\forall_Q.(P \wedge Q) \rightarrow P$
◀`Exact H1.`▶
Q.E.D.

```
ExFalso <Hyp>.
```

Type: Instantaneous; logic.
Info: Proves a goal that contains a hypothesis stating `FALSE`.

Example: $\langle$`H1:FALSE`$\rangle \vdash 5 = 6$
◀`ExFalso H1.`▶
Q.E.D.

`Extensionality <Name>.`

Type: Equivalence; backwards; logic.
Info: Proves equality of functions by means of extensionality.
Details: The current goal must of the form $E_1 = E_2$, and both E_1 and E_2 must be functions. The goal is then replaced with $\forall_{Name}.(E_1\ Name) = (E_2\ Name)$.
Example: $\vdash$ `(++ [])` $=$ `id`
◀`Extensionality xs.`▶
$\vdash \forall_{xs}.$`[] ++` $xs =$ `id` xs
Notes: To prevent proving $\perp = \lambda x.\perp$, which is not valid, additional definedness conditions are created under certain conditions.

`Generalize <Expr> to <Name>.`

Type: Strengthening; backwards; logic.
Info: Generalizes an arbitrary subexpression.
Details: In the to prove, replaces all occurrences of the indicated expression with the variable *Name*. Then, adds the quantor $\forall_{Name}$ in front of it.
Example: $xs \vdash$ `(reverse` xs`) ++ []` $=$ `reverse` xs
◀`Generalize (reverse xs) to ys.`▶
$\vdash \forall_{ys}.ys$ `++ []` $= ys$

`Induction <Var>.`

Type: Strengthening; backwards; programming.
Info: Performs structural induction on a variable
Details: The type of the indicated variable must be `Int`, `Bool` or algebraic. A goal is created for each root normal form(RNF) the variable may have, which includes $\perp$. The RNFs of an algebraic type are determined by its constructors. In each created goal, the variable is replaced by its corresponding RNF. Universal quantors are created for new variables. Additionally, induction hypotheses are added (as implications) for all recursive variables.
Example: $\vdash \forall_{xs}.xs$ `++ []` $= xs$
◀`Induction xs.`▶
(1) $\vdash \perp$ `++ []` $= \perp$
(2) $\vdash$ `[]++ []` $=$ `[]`
(3) $\vdash \forall_x \forall_{xs}.(xs$ `++ []` $= xs) \rightarrow$ `[`x`:`xs`] ++ []` $=$ `[`x`:`xs`]`

`Injective.`

Type: Strengthening; backwards; logic.
Info: Proves equality of applications by making use of injectivity.
Details: Replaces a goal of the form $(S\ E_1 \ldots E_n) = (S\ E'_1 \ldots E'_n)$, where S is either a function or a constructor, with the conjunction $E_1 = E'_1 \wedge \ldots \wedge E_n = E'_n$.

Example: $xs, ys \vdash xs$ `++` `[]` $= xs$ `++` ys

◀`Injective.`▶

$xs, ys \vdash xs = xs \wedge$ `[]` $= ys$

Notes: This tactic can also be applied in a forwards manner on a hypothesis.

`IntArith.`

Type: Equivalence; backwards; logic.

Info: Built-in simplification of arithmetic expressions.

Details: This tactic operates on expressions containing applications of $+$, $-$ and $*$ on integers. It performs three simplifications: (1) $a * (b + c)$ is replaced with $a * b + a * c$; (2) constants are moved to the right as much as possible; and (3) computations on constants are carried out statically.

Example: $x, y \vdash 3 + 7 * (12 + x) - 100 = y$

◀`IntArith.`▶

$x, y \vdash 7 * x - 13 = y$

Notes: This tactic can also be applied in a forwards manner on a hypothesis.

`IntCompare.`

Type: Instantaneous; logic.

Info: Proves goals with contradictory integer comparisons.

Details: Only hypotheses of the exact form $x < y$ are used as input. If a chain $x < y < \ldots < x$ can be found, then the goal is proved immediately.

Example: $x, y, z, \langle \mathsf{H1}{:}y < x\rangle, \langle \mathsf{H2}{:}z < y\rangle, \langle \mathsf{H3}{:}x < z\rangle \vdash$ `FALSE`

◀`IntCompare.`▶

Q.E.D.

`Introduce <Name1> <Name2> ... <Namen>.`

Type: Equivalence; backwards; logic.

Info: Introduces universally quantified variables and hypotheses in the goal.

Details: The current goal must be of the form $\forall_{x_1 \ldots x_a}.P_1 \to \ldots P_b \to Q$, where $a + b = n$. The quantors and implications may be mixed. The variables $x_1 \ldots x_a$ and the hypotheses $P_1 \ldots P_b$ are deleted from the current goal and are added to the goal context using the names given.

Example: $\vdash \forall_x.(x = 7 \to \forall_y.(y = 7 \to x = y))$

◀`Introduce p H1 q H2.`▶

$p, q, \langle \mathsf{H1}{:}p = 7\rangle, \langle \mathsf{H2}{:}q = 7\rangle \vdash p = q$

`MoveQuantors <Dir>.`

Type: Equivalence; backwards; logic.

Info: Swaps implications and universal quantifications.

Details: The direction argument is either 'In' or 'Out'. When moving inwards, goals of the form $\forall_{x_1 \ldots x_n}.P_1 \to \ldots P_m \to Q$ are transformed to $P_1 \to \ldots P_m \to \forall_{x_1 \ldots x_n}.Q$, provided that none of the x_i occur in any of the P_j. The outwards move is the opposite of the inwards move.

Example: $R \vdash \forall_P \forall_Q . R \to \neg R \to P \wedge Q$
◀`MoveQuantors In.`▶
$R \vdash R \to \neg R \to \forall_P \forall_Q . P \wedge Q$
Notes: This tactic can also be applied in a forwards manner on a hypothesis.

`Opaque <Fun>.`

Type: Special.
Info: Marks a function as non-expandable.
Details: When a function is marked opaque, it will not be expanded by the reduction mechanism. Instead, reduction will stop.
Example: $\vdash$ `zip ([], [])` $=$ `[]`
◀`Opaque zip2; Reduce NF All.`▶
$\vdash$ `zip2 [] []` $=$ `[]`

`Reduce NF All.`

Type: Equivalence; backwards; programming.
Info: Reduces all expressions in the current goal to normal form.
Details: All redexes in the current goal are replaced by their reducts. This full reduction is accomplished by first using standard reduction to root normal form, and then continuing recursively on the top-level arguments.
Example: $\vdash$ `reverse` $[7 * 12, 100 - 12] = [89 - 1, 83 + 1]$
◀`Reduce NF All.`▶
$\vdash [88, 84] = [88, 84]$
Notes(1): An artificial limit is imposed on the maximum number of reduction steps in order to safely handle non-terminating reductions.
Notes(2): This tactic can also be configured to reduce n steps; or to reduce to root normal form; or to reduce a specific redex; or to reduce within a hypothesis.

`RefineUndefinedness.`

Type: Equivalence; backwards; logic.
Info: Refines undefinedness equalities.
Details: Attempts to refine all undefinedness equalities in the current goal of the form $(S\ E_1 \ldots E_n) =\perp$, where S is either a constructor or a halting function. Replaces the equality with the disjunction of all $E_i =\perp$ where E_i is on a strict position and not statically known to be defined.
Example: $x, y \vdash (x + y) - 13 =\perp$
◀`RefineUndefinedness.`▶
$x, y \vdash (x + y) =\perp$
Notes: This tactic can also be applied in a forwards manner on a hypothesis.

`Reflexive.`

Type: Instantaneous; logic.
Info: Utilizes the reflexivity of the built-in operators $=$ and $\leftrightarrow$.
Details: Immediately proves any goal of the form $\forall_{x_1 \ldots x_n} . P_1 \to \ldots P_m \to Q$, where Q is either $E = E$ or $P \leftrightarrow P$.

Example: $\vdash \forall_x \exists_y . x < y \rightarrow x + y = x + y$
◀`Reflexive.`▶
Q.E.D.

```
Rename <Name1> to <Name2>.
```

Type: Special.
Info: Renames an introduced variable or an introduced hypothesis.
Example: $x, y \vdash x < y \rightarrow \neg(x = y)$
◀`Rename x to z.`▶
$z, y \vdash z < y \rightarrow \neg(z = y)$

```
Rewrite <fact>.
```

Type: Usually strengthening, depends on fact; backwards; logic.
Info: Rewrites the current goal using an equality in a fact.
Details: A fact is either an earlier proved theorem or an introduced hypothesis, and must be of the form $\forall_{x_1 \ldots x_n} . P_1 \rightarrow \ldots P_m \rightarrow Q$, where Q is either $L = R$ or $L \leftrightarrow R$. It is only valid if $r_1 \ldots r_n$ can be found such that $L[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}]$ occurs within the to prove. If this is the case, then all occurrences of $L[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}]$ are replaced with $R[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}]$. Furthermore, goals are created for each condition of the fact; the i-th states $P_i[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}]$.
Example: $p, \langle \mathsf{H1}: \forall_x . \neg(x = \perp) \rightarrow x * 0 = 0 \rangle \vdash (p - 7) * 0 = 0$
◀`Rewrite H1.`▶
(1) $p, \langle \mathsf{H1}: \forall_x . \neg(x = \perp) \rightarrow x * 0 = 0 \rangle \vdash 0 = 0$
(2) $p, \langle \mathsf{H1}: \forall_x . \neg(x = \perp) \rightarrow x * 0 = 0 \rangle \vdash \neg(p - 7) = \perp$
Notes: This tactic can also be configured to rewrite from right to left; or to rewrite at one specific location only; or to rewrite within a hypothesis.

```
Specialize <Hyp> with <Expr>/<Prop>.
```

Type: Strengthening; forwards; logic.
Info: Specializes a universally quantified hypothesis.
Details: The hypothesis must be $\forall_x . P$, and the given expression/proposition r must have the same type as x. Then, the hypothesis is replaced with $P[x \mapsto r]$.
Example: $x, y, z, \langle \mathsf{H1}: x < y \rangle, \langle \mathsf{H2}: y < z \rangle, \langle \mathsf{H3}: \forall_a . x < a \rightarrow a < z \rightarrow x < z \rangle \vdash x < z$
◀`Specialize H3 with y.`▶
$x, y, z, \langle \mathsf{H1}: x < y \rangle, \langle \mathsf{H2}: y < z \rangle, \langle \mathsf{H3}: x < y \rightarrow y < z \rightarrow x < z \rangle \vdash x < z$

```
Split.
```

Type: Equivalence; backwards; logic.
Info: Splits a conjunction into separate goals.
Example: $P, Q, \langle \mathsf{H1}: P \rangle, \langle \mathsf{H2}: Q \rangle \vdash P \wedge Q$
◀`Split.`▶
(1) $P, Q, \langle \mathsf{H1}: P \rangle, \langle \mathsf{H2}: Q \rangle \vdash P$
(2) $P, Q, \langle \mathsf{H1}: P \rangle, \langle \mathsf{H2}: Q \rangle \vdash Q$
Notes: This tactic can also be applied in a forwards manner on a hypothesis.

`SplitCase <Num>.`

Type: Strengthening; backwards; programming.
Info: Splits a case expression into its alternatives.
Details: The case expression that will be split is indicated by means of an index (cases are numbered from left to right starting with 1). A new goal is created for each of the alternatives of the case, including one for $\bot$ and one for the default. In each goal, the case expression is replaced by the result of the alternative. Hypotheses are introduced to indicate which alternative was chosen.
Example: $xs, \langle \mathsf{H1}{:}\neg(xs = \bot)\rangle \vdash \texttt{case}\ xs\ \texttt{of}\ (\texttt{[}y{:}ys\texttt{]} \mapsto y; \_ \mapsto 12) > 0$

◀`SplitCase 1.`▶

(1) $xs, \langle \mathsf{H1}{:}\neg(xs = \bot)\rangle, \langle \mathsf{H2}{:}xs = \bot\rangle \vdash \bot > 0$

(2) $xs, y, ys, \langle \mathsf{H1}{:}\neg(xs = \bot)\rangle, \langle \mathsf{H2}{:}xs = \texttt{[}y{:}ys\texttt{]}\rangle \vdash y > 0$

(3) $xs, \langle \mathsf{H1}{:}\neg(xs = \bot)\rangle, \langle \mathsf{H2}{:}xs = \texttt{[]}\rangle \vdash 12 > 0$

`SplitIff.`

Type: Equivalence; backwards; logic.
Info: Splits a $\leftrightarrow$ into a $\rightarrow$ and a $\leftarrow$.
Details: The current goal must be of the form $P \leftrightarrow Q$. Two goals are created, one for with $P \rightarrow Q$ and one for $Q \rightarrow P$.
Example: $P, Q, \langle \mathsf{H1}{:}P \rightarrow Q\rangle, \langle \mathsf{H2}{:}Q \rightarrow P\rangle \vdash P \leftrightarrow Q$

◀`SplitIff.`▶

(1) $P, Q, \langle \mathsf{H1}{:}P \rightarrow Q\rangle, \langle \mathsf{H2}{:}Q \rightarrow P\rangle \vdash P \rightarrow Q$

(2) $P, Q, \langle \mathsf{H1}{:}P \rightarrow Q\rangle, \langle \mathsf{H2}{:}Q \rightarrow P\rangle \vdash Q \rightarrow P$

Notes: This tactic can also be applied in a forwards manner on a hypothesis.

`Symmetric.`

Type: Equivalence; backwards; logic.
Info: Utilizes the symmetry of the built-in operators = and $\leftrightarrow$.
Details: The current goal must be of the form $\forall_{x_1 \ldots x_n}.P_1 \rightarrow \ldots P_m \rightarrow Q$, where Q is either $E_1 = E_2$ or $Q_1 \leftrightarrow Q_2$. If this is the case, then Q is replaced with $E_2 = E_1$ if it was a =, and with $Q_2 \leftrightarrow Q_1$ if it was a $\leftrightarrow$.
Example: $x, \langle \mathsf{H1}{:}x = y\rangle \vdash y = x$

◀`Symmetric.`▶

$x, \langle \mathsf{H1}{:}x = y\rangle \vdash x = y$

Notes: This tactic can also be applied in a forwards manner on a hypothesis.

`Transitive <Expr>/<Prop>.`

Type: Equivalence; backwards; logic.
Info: Utilizes the transitivity of the built-in operators = and $\leftrightarrow$.
Details: If the argument T is an expression, then the current goal must be of the form $E_1 = E_2$; if T is a proposition, then it must be of the form $P_1 \leftrightarrow P_2$. Two goals are then created, one stating $E_1 = T$ (or $P_1 \leftrightarrow T$), and the other stating $T = E_2$ (or $T \leftrightarrow P_2$).

Example: $P \vdash P \leftrightarrow ((P \wedge P) \wedge P)$
◀`Transitive` $(P \wedge P)$.▶
(1) $P \vdash P \leftrightarrow (P \wedge P)$
(2) $P \vdash (P \wedge P) \leftrightarrow ((P \wedge P) \wedge P)$

`Transparent.`

Type: Special.
Info: Marks a function as expandable.
Details: Undos the effect of Opaque.
Example: $\vdash$ `zip ([], [])` $=$ `[]`
◀`Opaque zip2; Transparent zip2; Reduce NF All.`▶
$\vdash$ `[]` $=$ `[]`

`Trivial.`

Type: Instantaneous; logic.
Info: Proves the trivial proposition `TRUE`.
Details: Immediately proves any goal of the form $\forall_{x_1 \ldots x_n}.P_1 \rightarrow \ldots P_m \rightarrow$ `TRUE`.
Example: $\vdash \forall_P.P \rightarrow \neg P \rightarrow$ `TRUE`
◀`Trivial.`▶
Q.E.D.

`Uncurry.`

Type: Equivalence; backwards; programming.
Info: Uncurries all applications in the current goal.
Details: Forces all curried applications $(f\ x_1 \ldots x_i)\ x_{i+1} \ldots x_n$ in the current goal to be uncurried to $f\ x_1 \ldots x_n$.
Example: $\vdash$ `[`$((+)\ 1)\ 1$: `map` $((+)\ 1)$ `[]]` $=$ `[2]`
◀`Uncurry.`▶
$\vdash$ `[`$1 + 1$: `map` $((+)\ 1)$ `[]]` $=$ `[2]`
Notes: This tactic can also be applied in a forwards manner on a hypothesis.

`Undo <num>.`

Type: Special.
Info: Undos the last n steps of the proof.
Details: SPARKLE does not memorize the last actions of the user. Instead, n upwards steps in the *proof tree* are made.
Example: $\vdash \forall_{xs}.xs$ `++ []` $=$ `[]`
◀`Induction xs; Reduce. Undo 2.`▶
$\vdash \forall_{xs}.xs$ `++ []` $=$ `[]`

`Witness <Expr>/<Prop>.`

Type: Strengthening; backwards; logic.
Info: Chooses a witness for an existentially quantified goal.
Details: The current goal must be of the form $\exists_x.P$, and $P[x \mapsto T]$ (where T

is the term argument) must be welltyped. If this is the case, then the goal is replaced with $P[x \mapsto T]$.
Example: $\vdash \exists_x . x * x = x$
◀`Witness 1.`▶
$\vdash 1 * 1 = 1$
Notes: This tactic can also be applied in a forwards manner on a hypothesis.

An Introduction to the Lambda Calculus

Zoltán Csörnyei and Gergely Dévai

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest
{csz,deva}@inf.elte.hu

Abstract. Lambda calculus (λ-calculus) is one of the most well-known formal models of computer science. It is the basis for functional programming like Turing machines are the foundation of imperative programming. These two systems are equivalent and both can be used to formulate and investigate fundamental questions about solvability and computability.

First, we introduce the reader to the basics of λ-calculus: its syntax and transformation rules. We discuss the most important properties of the system related to normal forms of λ-expressions. We present the recursive version of λ-calculus and finally give the classical results that establish the link between λ-calculus, partial recursive functions and Turing machines.

1 Historical Background

In 1924, Moses Schönfinkel introduced *Combinatory Logic*, which was independently reformulated by Haskell B. Curry in 1930. It is a computational model based on *combinators*, which are actually higher order functions. This model can be represented in λ-calculus: we give the λ-expressions corresponding to the three combinators (called I, K and S) as an example in section 2.2.

λ-calculus was originally developed in 1932-33 by the logician Alonzo Church as a foundation for mathematics. In 1936, Stephen C. Kleene showed that the λ-calculus is a universal computing system, that is, the λ-definable numeric functions are exactly the partial recursive functions. One year later, Alan M. Turing proved that the classes of functions defined by λ-calculus and Turing machines coincide.

The original λ-calculus was untyped. The problems arising from the lack of types were solved by the typed λ-calculi developed in the 1940s. Nowadays typed λ-calculus is considered as the more fundamental theory, because the original untyped calculus can be seen as a special case with one single type.

From the 1960s the λ-calculus was used in several research projects related to programming languages. For example, Peter Landin used the λ-calculus to analyse Algol 60 and introduced the *ISWIM* ("If you See What I Mean") language as a framework of future languages. The *SECD* ("Stack, Environment, Control, Dump") interpreter was used to implement the ML language. In the 1970s Christopher Wadsworth developed *graph reduction* as a modern method to implement (lazy) functional languages.

2 Syntax, Notions and Notations

2.1 Syntax of the λ-Calculus

Definition 1. *Let V be a countable set of variables. The set Λ of* ***λ-terms*** *or* ***λ-expressions*** *is a set of words on the alphabet $V \cup \{(,),\ .\ ,\ \lambda\}$ inductively defined as follows:*

- *$x \in V$ implies $x \in \Lambda$,*
- *$E \in \Lambda$ and $x \in V$ implies $(\lambda x.E) \in \Lambda$ (**abstraction**),*
- *$E \in \Lambda$ and $F \in \Lambda$ implies $(EF) \in \Lambda$ (**application**).*

For example, if $x, y \in V$, then x, $(\lambda x.x)$ and $((\lambda x.x)y)$ are λ-terms.

2.2 Notational Conventions

The letters x, y, ... will denote variables, while capital letters E, F, ... will denote arbitrary λ-terms.

We will use the ≡ symbol to denote **syntactic identity** of λ-terms ($(\lambda x.x) \equiv (\lambda x.x)$), to define **simplified notations** for λ-terms and to **define a name** for a λ-term.

Simplifying the syntax
To make the syntax of λ-terms more convenient, we can leave out redundant parentheses:

$(\lambda x.x) \equiv \lambda x.x$
$\lambda x.(\lambda y.E) \equiv \lambda x.\lambda y.E$

Furthermore, there are two bracketing conventions:

- $(\lambda x.EF)$ abbreviates $\lambda x.(EF)$, which was used by Barendregt and others, and
- $(\lambda x.E\ F)$ abbreviates $(\lambda x.E)F$, which will be used in this material.

Finally, we can close up multiple abstractions using the following notation:
$\lambda xy.E \equiv \lambda x.\lambda y.E$

Defining names for λ-terms
We can give names to particular λ-terms. As we will discuss in the next section, $\lambda x.x$ represents the identity function, so we can give it the following name:

$id \equiv \lambda x.x$

We can also define the three combinators of *Combinatory Logic*:

$\mathsf{I} \equiv \lambda x.x$,
$\mathsf{K} \equiv \lambda xy.x$,
$\mathsf{S} \equiv \lambda xyz.(xz(yz))$.

2.3 Functions

Definition 2. *We call a term obtained by λ-abstraction, like $\lambda x.E$ (where E is a λ-term), an* ***unnamed*** *or* ***anonymous function****, where x is the* ***variable*** *or* ***formal argument****, and E is the* ***body*** *of the abstraction.*

If E is a function, in the application EF the term F is called the ***actual argument*** *of the function.*

To understand why we call $\lambda x.x$ a function, let us show via an example how application works. We substitute the formal argument (x) for the actual argument (y) in the body of the function:

$$(\lambda x.x)y \to y$$

That is, $\lambda x.x$ is the *identity* function, as it returns its actual argument. In section 3.1 we formally define how this *reduction step* works.

Definition 3. *A* ***higher-order function*** *accepts functions as arguments and is able to return a function as its result.*

For example, a function of the form $\lambda x.(\lambda y.E)$ returns a function for any argument.

Definition 1 only allows functions with one argument. This does not limit the expressive power of the λ-calculus, as we can use higher order functions instead of functions with more than one argument. This transformation is called **currying**. If we want to encode a (classical) function that takes two arguments, by currying we get a higher order function such that it takes the first argument and returns a function that takes the second argument.

In section 5.5 we will show different ways to encode the natural numbers and addition in λ-calculus. For the sake of this example let us suppose that $\ulcorner n \urcorner$ and $\ulcorner m \urcorner$ denotes λ-terms representing the natural numbers n and m. Furthermore, $\ulcorner n \urcorner +_\lambda \ulcorner n \urcorner$ denotes the λ-term that represents the sum $n+m$, that is,

$$\ulcorner n \urcorner +_\lambda \ulcorner n \urcorner = \ulcorner n + m \urcorner$$

holds. Using these abbreviations, the curried form of the addition function is the following higher-order function:

$$\lambda x.\lambda y.(x +_\lambda y)$$

We use two applications to give the arguments $\ulcorner n \urcorner$ and $\ulcorner m \urcorner$, and using two reduction steps we get the result, which is really the λ-term representing $n+m$:

$$((\lambda x.\lambda y.(x +_\lambda y))\, \ulcorner n \urcorner)\, \ulcorner m \urcorner \to (\lambda y.(\ulcorner n \urcorner +_\lambda y))\, \ulcorner m \urcorner \to \ulcorner n \urcorner +_\lambda \ulcorner m \urcorner = \ulcorner n + m \urcorner$$

The first reduction step results in $\lambda y.(\ulcorner n \urcorner +_\lambda y)$. This λ-term represents the function that takes a single argument and increments it by n.

2.4 Variables

Similarly to classical first order logic, in λ-calculus we also use the notion of free and bound variables. An occurrence of a variable x in a λ-expression is **free** if it is not within the body of an abstraction with formal argument x, otherwise it is **bound**.

Definition 4. *$FV(E)$ is the set of* ***free variables*** *in E and can be defined inductively as follows:*

– $FV(x) = \{x\}$,
– $FV(\lambda x.E) = FV(E) \setminus \{x\}$,
– $FV(EF) = FV(E) \cup FV(F)$.

For example, $FV(\lambda x.x) = \emptyset$ and $FV((\lambda x.x)y) = \{y\}$.

Definition 5. *We say, that E is* ***closed*** *if $FV(E) = \emptyset$.*
The ***set of closed λ-terms*** *is $\Lambda^0 = \{E \in \Lambda \mid E \text{ is closed}\}$.*
A ***closure*** *of $E \in \Lambda$ is $\lambda x_1 x_2 \ldots x_n.E$, where $\{x_1, x_2, \ldots, x_n\} = FV(E)$.*

Definition 6. *An occurrence of a variable is* ***bound*** *if is not free.*
The set of bound variables is $BV(E) = \{x \in \Lambda \mid x \text{ is a bound variable in } E\}$.

For example, $BV(x) = \emptyset$ and $BV((\lambda x.x)y) = \{x\}$. It is not always the case that $FV(E) \cap BV(E) = \emptyset$. Consider the λ-term $((\lambda x.x)x)$, where

$$FV((\lambda x.x)x) = BV((\lambda x.x)x) = \{x\}$$

2.5 Remark about Proofs

The goal of this paper is to introduce the basic consepts and fundamental results of λ-calculus to the reader. Therefore we do not give the proofs of the lemmas and theorems. The interested reader is referred to [1].

3 Semantics

In this section we formally define the semantics of λ-calculus. First we deal with operational semantics: λ-terms can be viewed as expressions to be calculated, and this calculation is performed using reduction steps. These reduction steps consist of the conversions that we define in the following sections.

An other question is the equality of λ-expressions. Section 3.4 defines precisely when are two λ-expressions equal, making the system a real calculus.

3.1 Conversions

Substitution
We denote the substitution of a free variable x by a term F in a λ-term E by $E[x := F]$. (In some other material the notation $E[F/x]$ is also used.)

Definition 7. ***Substitution of variables*** *is defined inductively as follows:*

1. $x[y := G] \equiv \begin{cases} G, \textit{ if } x \equiv y, \\ x, \textit{ otherwise}, \end{cases}$

2. $(EF)[y := G] \equiv (E[y := G])(F[y := G])$,

3. $(\lambda x.E)[y := G] \equiv \begin{cases} \lambda x.E, & \textit{if } x \equiv y, \\ \lambda x.E[y := G], & \textit{if } x \not\equiv y \textit{ and } x \notin FV(G), \\ \lambda x.E, & \textit{otherwise}. \end{cases} \qquad (*)$

Let us substitute the variable y by z in $(\lambda y.y)y$:

$$((\lambda y.y)y)[y := z] \equiv ((\lambda y.y)[y := z])(y[y := z]) \equiv (\lambda y.y)z$$

In the first step we use the second rule of the definition. Then, in the first part of the application we use the third rule: we do not change the bound variable in the abstraction. In the second part the first rule is applied.

Why is the $x \notin FV(G)$ restriction necessary in the $(*)$ condition? In the λ-term vx, both v and x are free variables. If we replaced y by vx in $\lambda x.y$, then we would get $\lambda x.(vx)$, where x is not free any more. We say that in this case the variable binding operation λx captures the (originally free) variable x. This violates the common mathematical intuition, that a variable is a placeholder. In such a case, the restriction in the definition completely forbids the substitution:

$$(\lambda x.y)[y := vx] \equiv \lambda x.y$$

If we want to substitute y anyway, we have to *rename* the variable of the abstraction first:

$$(\lambda z.y)[y := vx] \equiv \lambda z.(vx)$$

α-conversion

Definition 8. *Renaming the bound variable x in a λ-term of the form $\lambda x.E$ is called* ***α-conversion*** *and results in $\lambda y.(E[x := y])$ where y must not occur free in E.*

If we can transform the expression F to F' by performing α-conversions on its subexpressions, then we use the following notation: $F \equiv_\alpha F'$.

Note, that if we rename x to y in $\lambda x.(\lambda x.x)$, the result (according to definitions 7 and 8) is

$$\lambda x.(\lambda x.x) \equiv_\alpha \lambda y.((\lambda x.x)[x := y]) \equiv \lambda y.(\lambda x.x),$$

which is correct, as the inner x is bound to the inner λx and not to be renamed.

There is a restriction on the new variable name in definition 8: it prevents that a free variable become captured in the body of the expression. As a consequence, the following lemma holds:

Lemma 9. *If $E \equiv_\alpha F$, then FV(E) = FV(F).*

Now we give a second version of substitution, which applies alpha conversion when necessary.

Definition 10. ***Substitution of variables with α-conversion*** *is defined inductively as follows:*

$$1.\ x[y := G] \equiv \begin{cases} G, & \text{if } x \equiv y, \\ x, & \text{otherwise,} \end{cases}$$

$$2.\ (EF)[y := G] \equiv (E[y := G])(F[y := G]),$$

$$3.\ (\lambda x.E)[y := G] \equiv \begin{cases} \lambda x.E, & \text{if } x \equiv y, \\ \lambda x.E[y := G], & \text{if } x \not\equiv y \text{ and } x \notin FV(G), \\ (\equiv_\alpha \lambda z.E[x := z])[y := G], & \\ & \text{if } x \not\equiv y \text{ and } x \in FV(G) \end{cases} \qquad (*)$$

Let us examine, how this new definition of substitution solves our previous problem: we want to substitute y by vx in $\lambda x.y$. According to the $(*)$ case of the 3rd rule, we first have to perform an α-conversion:

$$(\lambda x.y)[y := vx] \equiv (\lambda z.(y[x := z]))[y := vx] \equiv (\lambda z.y)[y := vx] \equiv \lambda z.vx$$

Note, that we have to choose the new variable according to the condition of α-conversion, that is $z \notin FV(E)$ has to hold (see definition 8).

Although it is possible to use a variable such that $z \in FV(G)$, but it is not an efficient decision. For example if we use v in the α-conversion, we get:

$$(\lambda x.y)[y := vx] \equiv (\lambda v.(y[x := v]))[y := vx] \equiv (\lambda v.y)[y := vx]$$

Here, we have to perform the $(*)$ case of the 3rd rule again and perform a second α-conversion. We are forced to do this until we choose a variable that satisfies the $z \notin FV(G)$ condition. Effective implementations should consider this as an additional restriction.

The following lemma expresses a rule about changing the order of substitutions:

Lemma 11 (Substitution lemma). *If $x \not\equiv y$ and $x \notin FV(G)$, then* $E[x := F][y := G\ ] \equiv_\alpha\ E[y := G][x := F[y := G]]$.

3.2 β-Reduction

In section 2.3 we have informally presented how to compute an application in λ-calculus. Now, using substitution of definition 8 we can define it formally:

Definition 12. *The* ***β-reduction*** *substitutes the argument F into the abstraction's body E:*

$$(\lambda x.E)F \to_\beta E[x := F].$$

If we apply the identity function on y, we get the result by one β-reduction step:

$$(\lambda x.x)y \ \to_\beta \ y.$$

Note, that in the above definition of β-reduction, substitution with α-conversion is applied, and definition 10 does not allow that free variables in the actual argument become bound. For example

$$(\lambda xy.x)y \not\to_\beta \lambda y.y,$$

because an α-conversion is performed during the substitution:

$$(\lambda xy.x)y \to_\beta (\lambda y.x)[x := y] \equiv (\lambda z.x)[x := y] \equiv \lambda z.y$$

Definition 13. *If $E \to_\beta F$, then E is the result of a **β-abstraction** on F.*
*A **β-reduction** or **β-abstraction** is called a **β-conversion** step.*

Definition 14. *If a λ-term can be β-reduced, we call it a **reducible expression** or **redex**.*

Definition 15. *A **reduction sequence** of a term consists of zero or more reduction steps.*
If E can be transformed to F by a sequence of β-reductions, we write: $E \ \twoheadrightarrow \ F$.

For example

$$(\lambda z.(zy))(\lambda x.x) \ \twoheadrightarrow y$$

is true, because

$$(\lambda z.(zy))(\lambda x.x) \ \to_\beta \ (\lambda x.x)y \ \to_\beta \ y$$

3.3 The de Bruijn Notation of Terms

The de Bruijn notation is an alternate notation of λ-expressions. It uses numbers instead of names to refer to formal parameters. Although it is not very readable, it has advantages: it avoids the possibility of name capture and removes the need for alpha conversion, making the β-reduction easier to implement.

Definition 16. *We map each bound variable to the number of λs, which the variable is in the scope of, from the location of the reference to the binding λ.*
To extend this mapping for free variables, we fix a closure of the λ-expression and use the same rule on it.
*We transform a λ-expression to **de Bruijn notation** by omitting all the formal arguments and replacing all the variables by natural numbers according to the mapping above.*

We give some examples for the de Bruijn notation:

$\lambda x.x$	$\lambda.1$,
$\lambda xy.(xy)$	$\lambda.\lambda.2\ 1$,
$\lambda xy.((xy)z)$	$\lambda.\lambda.2\ 1\ 3$,
$\lambda z.(((\lambda x.(xy))x)z)$	$\lambda.(\lambda.1\ 3)3\ 1$ *or* $\lambda.(\lambda.1\ 4)2\ 1$,
$(\lambda y.(\lambda z.(zy))y)((\lambda t.t)z)$	$(\lambda.(\lambda.1\ 2)1)((\lambda.1)1)$.

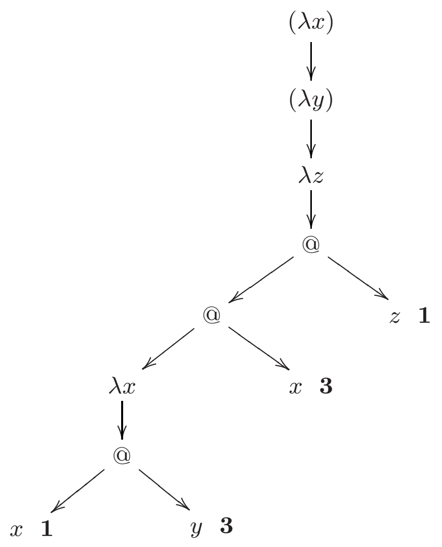


Fig. 1. The de Bruijn-numbers of $\lambda z.(\lambda x.x\ y)x\ z$

Notice that α-equivalent terms are equal in the de Bruijn notation. The terms $\lambda xy.(xy) \equiv_\alpha \lambda zy.(zy)$ both have the same de Bruijn representation: $\lambda.\lambda.2\ 1$

Definition 17. *The **β-reduction** for de Bruijn notation is*

$$(\lambda.P)Q \to_\beta P[1 := Q]$$

where $n[m := N] \equiv \begin{cases} n-1, & \textit{if } n > m, \\ n, & \textit{if } n < m, \\ \mathcal{C}_{n,1}(N), & \textit{if } n = m. \end{cases}$

$(M_1M_2)[m := N] \equiv (M_1[m := N])(M_2[m := N])$

$(\lambda.M)[m := N] \equiv \lambda.(M[m+1 := N])$

and

$\mathcal{C}_{n,i}(j) \equiv \begin{cases} j, & \textit{if } j < i, \\ j+n-1, & \textit{if } j \geq i. \end{cases}$

$\mathcal{C}_{n,i}(N_1N_2) \equiv \mathcal{C}_{n,i}(N_1)\mathcal{C}_{n,i}(N_2)$

$\mathcal{C}_{n,i}(\lambda N) \equiv \lambda(\mathcal{C}_{n,i+1}(N))$

Let us present the de Bruin style β-reduction for the following example:

$$\lambda x.\underline{((\lambda yz.y)x)} \to_\beta \lambda x.(\lambda z.x)$$

The redex of the expression is underlined. According to definition 17 this reduction step is computed as follows:

$$\lambda.\underline{((\lambda.\lambda.2)1)} \to_\beta \lambda.((\lambda.2)[1 := 1]) \equiv \lambda.(\lambda.(2[2 := 1])) \equiv \lambda.(\lambda.(\mathcal{C}_{2,1}(1))) \equiv \lambda.\lambda.2$$

In section 3.2 we saw an example where α-conversion was necessary during the β-reduction step:

$$(\lambda xy.x)y \to_\beta (\lambda y.x)[x := y] \equiv (\lambda z.x)[x := y] \equiv \lambda z.y$$

The de Bruijn representation of $(\lambda xy.x)y$ is $(\lambda.\lambda.2)1$, which is exactly the redex of the example above. The result is $\lambda.2$ again, and this represents both $\lambda z.y$ and $\lambda z.x$. The α-conversion was not necessary at all.

3.4 Equality

In this section we define when are two λ-expressions equal. It is intuitive that α-conversion and β-reduction preserve the meaning of λ-expressions, so we first give a definition based on these conversions. The λ-expressions E and F are **equal** ($E = F$), if they can be transformed into each other by a sequence of α- and β-conversions (see figure 2).

$$\begin{tikzcd}
E \equiv G_1 \arrow[dr, two heads] & & G_3 \arrow[dl, two heads] \arrow[dr, two heads] & & \cdots \arrow[dr, two heads] & & G_{2m-1} \arrow[dl, two heads] \arrow[dr, two heads] & & G_{2m+1} \equiv F \arrow[dl, two heads] \\
& G_2 & & \cdots & & G_{2m-2} & & G_{2m} &
\end{tikzcd}$$

Fig. 2. The equality $E = F$

Definition 18. *$E = F$ holds if exists a sequence of λ-expressions $G_0, G_1, ..., G_n$ such that $E \equiv G_0$, $G_n \equiv F$ and for each $i \in [1..n]$: G_{i-1} can be converted to G_i by a single α- or β-conversion.*

Note that according to definition 13, β-conversion is either a β-reduction or a β-abstraction.

The immediate consequences of this definition are summarised by the following lemma:

Lemma 19. *According to definition 18, equality is*

– *reflexive,*
– *symmetric and*
– *transitive.*

The so called *Leibniz-rule* is a consequence of definition 18. It states that replacing a subexpression by an equal one results in an equal expression.

Lemma 20 (Leibniz-rule). *If $E_1 = F_1$, $E \equiv G_1 E_1 G_2$ and $F \equiv G_1 F_1 G_2$, then $E = F$.*

The following rules are special cases of the Leibniz-rule:

Corollary 21. *If $E = F$ then*
1. $EG \ = FG$,
2. $GE \ = GF$,
3. $\lambda x.E = \lambda x.F$.

To prove that two λ-terms are equal, we can use the following calculus. The consequences of definition 18 that we listed above, are all axioms of the calculus.

Definition 22. *The formulas of **λ-calculus** have the form $E = F$, where $E, F \in \Lambda$ and the calculus is axiomatised by the following axioms and rules:*

I.	$(\lambda x.E)F = E\ [x := F]$	*β-conversion* (β)
II.i.	$E = E$	*reflexivity* (ρ)
II.ii.	$E = F \Rightarrow\ F = E$	*symmetry* (σ)
II.iii.	$E = F,\ F = G\ \Rightarrow\ E = G$	*transitivity* (τ)
II.iv.	$E = F\ \Rightarrow\ EG = FG$	*1. corollary of Leibniz-rule* (μ)
II.v.	$E = F\ \Rightarrow\ GE = GF$	*2. corollary of Leibniz-rule* (ν)
II.vi.	$E = F\ \Rightarrow\ \lambda x.E = \lambda x.F$	*ξ-rule* (ξ)

For example, we prove in the calculus that $\lambda x.((\lambda y.y)z) = \lambda x.z$:

1. $(\lambda y.y)z = z$ (by axiom I.)
2. $\lambda x.((\lambda y.y)z) = \lambda x.z$ (by axiom II.vi. and step 1.)

Note that α-conversion is not an axiom. This implies that, for example

$$\lambda x.x = \lambda y.y$$

is not provable in the calculus. Fortunately this is not a serious limitation, as the following lemma states.

Lemma 23. *If $E = F$ holds by definition 18, then exists F' such that $F \equiv_\alpha F'$ and $E = F'$ can be proved in the calculus of definition 22.*

3.5 η-Conversion and Extensionality

If two functions give the same results for all possible actual arguments, it is natural to consider them equal. However, the above definition and axiomatisation of equality do not express this intuition. For example, $\lambda x.(yx)$ and y are not equal according to definition 18, but for any λ-term F, $(\lambda x.(yx)\ F) = yF$ holds.

Definition 24. ***η-conversion** is the reduction of an abstraction of the form $\lambda x.(Ex)$ to E, if $x \notin FV(E)$.*

We use the following notation for η-conversion: $\lambda x.(Ex)\ \leftrightarrow_\eta\ E$.

It is possible to express this property more directly. This extension of equality is called extensionality and is defined as follows.

Definition 25. *If $Ex = Fx$ and $x \notin FV(E), x \notin FV(F)$ then we say that $E = F$ by* ***extensionality****.*

Now we address the question whether the two formulations (η-conversion and extensionality) are equivalent or not. First we extend the λ-calculus defined in 22 with them.

Definition 26. *The axioms of* ***$\lambda\eta$-calculus*** *are the ones of λ-calculus and the following:*

If $E \leftrightarrow_\eta F$ then $E = F$ (η-rule).

Definition 27. *The axioms of* ***λ-ext-calculus*** *are the ones of λ-calculus and the following:*

If $E = F$ by extensionality then $E = F$ (ext-rule).

We can prove that the η-rule of the $\lambda\eta$-calculus is provable in λ-ext-calculus and conversely, the ext-rule of the λ-ext-calculus is provable in the $\lambda\eta$-calculus.

Theorem 28 (Curry). *The $\lambda\eta$-calculus and the λ-ext-calculus are equivalent.*

4 Normal Forms

Computation in λ-calculus consists of a sequence of reduction steps performed on a λ-term. If this computation results in a λ-term that is not reducible any more, we can consider that as the result of the computation. This result is called a *normal form.*

Definition 29. *A term F which contains no redices is called a* ***normal form****. We also say that F is a term* ***in normal form****.*

If a term E reduces to a term F in normal form, then F is called a ***normal form of E****.*

If $E \twoheadrightarrow F$ and F is a term in normal form, then term E ***has a normal form****.*

The term $\lambda x.x\ y$ has a normal form: y. But not all λ-expressions have one. For example

$$\Omega \equiv (\lambda x.(xx))(\lambda x.(xx)) \rightarrow\ (\lambda x.(xx))(\lambda x.(xx))\ \rightarrow\ \ldots$$

is an infinite sequence, where all elements are reducible.

4.1 Church–Rosser Theorem

Sometimes it is possible to perform different reduction steps on a term. In the following example we show two different reduction sequences:

$$\begin{aligned}\mathsf{K}(\mathsf{I}\ z) &\equiv (\lambda xy.x)((\lambda x.x)z)\\ &\to\ (\lambda xy.x)z\ \to\ \lambda y.z \text{ and}\\ &\to\ \lambda y.((\lambda x.x)z)\ \to\ \lambda y.z.\end{aligned}$$

In this case, the different reduction sequences led to the same normal form. Is this the case in general? In section 4.2 we will see that not all reduction sequences reach a normal form. However, the fundamental theorem of λ-calculus states that reduction is **confluent**: no two sequences of reductions can reach distinct normal forms.

The following lemma is the first step towards this fundamental theorem.

Lemma 30 (The diamond property). *If $E \twoheadrightarrow F_1$ and $E \twoheadrightarrow F_2$ then there exists a term F such that $F_1 \twoheadrightarrow F$ and $F_2 \twoheadrightarrow F$.*

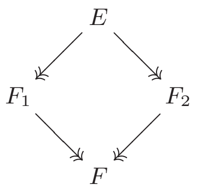

$$\begin{tikzcd}
& E \arrow[dl, two heads] \arrow[dr, two heads] & \\
F_1 \arrow[dr, two heads] & & F_2 \arrow[dl, two heads] \\
& F &
\end{tikzcd}$$

Fig. 3. The diamond property

The *I. Church–Rosser theorem* states that equal terms can be reduced to the same normal form.

Theorem 31 (I. Church–Rosser theorem). *If $E_1 = E_2$ then there exists F such that $E_1 \twoheadrightarrow F$ and $E_2 \twoheadrightarrow F$.*

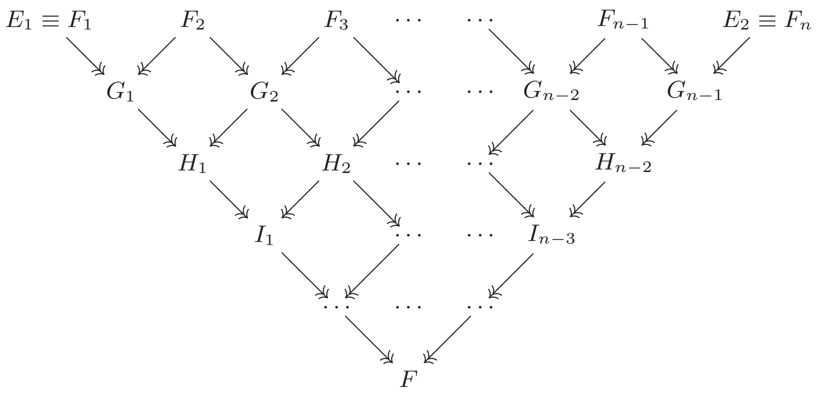

$$\begin{tikzcd}
E_1 \equiv F_1 \arrow[dr, two heads] & & F_2 \arrow[dl, two heads] \arrow[dr, two heads] & & F_3 \arrow[dl, two heads] \arrow[dr, two heads] & \cdots & \cdots \arrow[dr, two heads] & & F_{n-1} \arrow[dl, two heads] \arrow[dr, two heads] & & E_2 \equiv F_n \arrow[dl, two heads] \\
& G_1 \arrow[dr, two heads] & & G_2 \arrow[dl, two heads] \arrow[dr, two heads] & & \cdots \arrow[dl, two heads] & \cdots & G_{n-2} \arrow[dl, two heads] \arrow[dr, two heads] & & G_{n-1} \arrow[dl, two heads] & \\
& & H_1 \arrow[dr, two heads] & & H_2 \arrow[dl, two heads] \arrow[dr, two heads] & \cdots & \cdots \arrow[dr, two heads] & & H_{n-2} \arrow[dl, two heads] & & \\
& & & I_1 \arrow[dr, two heads] & & \cdots \arrow[dl, two heads] & \cdots & I_{n-3} \arrow[dl, two heads] & & & \\
& & & & \cdots \arrow[dr, two heads] & \cdots & \cdots \arrow[dl, two heads] & & & & \\
& & & & & F & & & & &
\end{tikzcd}$$

Fig. 4. I. Church–Rosser theorem

Now we summarise the consequences of the theorem. The following statements show that if two different reduction sequences of a term reach normal forms, then they are essentially the same (congruent).

Corollary 32. *If $E = F$ and E is in normal form, then $E \twoheadrightarrow F$.*

Corollary 33. *If $E = F$, then either E and F do not have normal forms, or E and F both have the* ***same*** *normal form.*

Corollary 34. *Two equal terms in normal form must be congruent.*

4.2 Reduction Strategies

In the previous section we have seen that one can reach the same normal form by performing different reduction steps when more than one is possible. On the other hand, from the point of view of efficiency, it is quite important to find the normal form as soon as possible. That is, we want to minimalize the length of reduction sequences.

For that reason several reduction strategies exist. Some of them are complete, that is, they find the normal form whenever it exists, while others are incomplete but faster in most cases. In this section we observe the most important strategies and their features.

Definition 35. ***Normalising reduction strategy*** *is a strategy that results in the normal form, if the normal form exists.*

Normal order strategy

Definition 36. *In normal order reduction strategy the* ***leftmost-outermost*** *redex is rewritten.*

Normal order reduction strategy is normalising, as the next theorem, the II. Church–Rosser theorem states.

Theorem 37 (II. Church–Rosser theorem). *If E has a normal form F, then there exists a normal order reduction of E to F, $E \twoheadrightarrow_{n.o.} F$*

That is the normal order reduction is optimal in the sense that if a term has a normal form, it always yields a normal form.

In the following example we use the combinator K defined in section 2.2 and the term Ω that we used in section 4.

$$\mathsf{K}\ x\ \Omega \equiv (\lambda xy.x)x((\lambda x.(xx))(\lambda x.(xx))) \rightarrow\ (\lambda y.x)((\lambda x.(xx))(\lambda x.(xx))) \rightarrow\ x.$$

We have seen previously that Ω has no normal forms, so performing reduction steps in the Ω part of the expression is useless. But, using normal order reduction, one can find the normal form of the expression, which is in fact independent of the Ω part.

Applicative order strategy

Definition 38. *In the applicative order reduction strategy the* ***leftmost-innermost*** *redex is rewritten.*

In contrast to normal order reduction, applicative order reduction may not terminate, even when the term has a normal form. We use the same example again:

$$\mathsf{K}\ x\ \Omega \equiv (\lambda xy.x)x((\lambda x.(xx))(\lambda x.(xx))) \rightarrow$$

$$\rightarrow\ (\lambda y.x)((\lambda x.(xx))(\lambda x.(xx))) \rightarrow\ (\lambda y.x)((\lambda x.(xx))(\lambda x.(xx))) \rightarrow\ \ldots$$

The advantage of applicative order evaluation is that it usually uses **less reduction steps** than normal order reduction. We demonstrate this by the following example:

– Normal order reduction:

$$(\lambda x.(+\ x\ x))(*\ 3\ 3) \rightarrow +(*\ 3\ 3)(*\ 3\ 3) \rightarrow (+\ 9)\ (*\ 3\ 3) \rightarrow +\ 9\ 9 \rightarrow 18.$$

– Applicative order reduction:

$$(\lambda x.(+\ x\ x)(*\ 3\ 3) \rightarrow (\lambda x.(+\ x\ x)9 \rightarrow +\ 9\ 9 \rightarrow 18.$$

Lazy evaluation
The efficiency of normal-order reduction can be improved without sacrificing its termination property by using **lazy evaluation**. This strategy delays the computation of a subterm until the result is known to be needed.

5 Representing Things in the λ-Calculus

In the next section we show the equivalence of the λ-calculus and the recursive function theory. In order to do that we first need to encode data using λ-terms. In this section we show how to represent common datastructures like booleans, numerals, lists and other constructions in the λ-calculus.

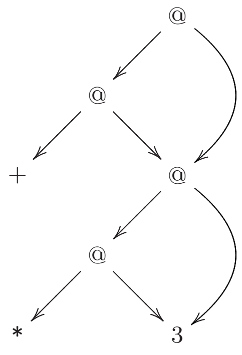


Fig. 5. The simplified graph of the expression $(\lambda x. +\ \ x\ x)(*\ 3\ 3)$

5.1 Booleans

We can represent the *true* and *false* values by functions taking two arguments and returning the first and second one respectively.

$$\begin{aligned}&\mathsf{true} \equiv \lambda xy.x,\\&\mathsf{false} \equiv \lambda xy.y.\end{aligned}$$

This representation can be understood by observing the definitions of some functions working with boolean values. The *if* function takes three arguments: the condition, the then-expression and the else-expression. It applies the condition on the other two arguments. If the condition is true, the application returns its first argument (the then-expression), otherwise the second one (the else-expression) is returned.

$$\mathsf{if} \equiv \lambda pqr.(pqr)$$

To describe the standard logical connectives, we can use the *if* function:

$$\begin{aligned}&\mathsf{and}\ E\ F \approx \mathsf{if}\ E\ F\ \mathsf{false},\\&\mathsf{or}\ E\ F \quad\approx \mathsf{if}\ E\ \mathsf{true}\ F,\\&\mathsf{not}\ E \qquad\approx \mathsf{if}\ E\ \mathsf{false}\ \mathsf{true}.\end{aligned}$$

Based on these rules, the definitions are the following:

$$\begin{aligned}&\mathsf{and} \equiv \lambda xy.(x\ y\ \mathsf{false}),\\&\mathsf{or}\ \ \ \equiv \lambda xy.(x\ \mathsf{true}\ y),\\&\mathsf{not} \equiv \lambda x.(x\ \mathsf{false}\ \mathsf{true}).\end{aligned}$$

5.2 Pairs

This data structure encapsulates two expressions into a pair and provides selector functions to access these expressions. We can use the standard $(E, F) \equiv \mathsf{pair}\ E\ F$ notation for pairs. The constructor and the selector functions are the following:

$$\begin{aligned}&\mathsf{pair} \quad\ \ \equiv \lambda xyz.(zxy),\\&\mathsf{first} \quad\ \ \equiv \lambda x.(x\ \mathsf{true})\ \ \equiv \lambda x.(x(\lambda yz.y)),\\&\mathsf{second} \equiv \lambda x.(x\ \mathsf{false})\ \equiv \lambda x.(x(\lambda yz.z)).\end{aligned}$$

That is $(E, F) = \lambda z.(zEF)$, because

$$(\lambda xyz.(zxy))\ E\ F \to_\beta (\lambda yz.(zEy))\ F \to_\beta \lambda z.(zEF).$$

As an example, let us compute the following expression:

$$\mathsf{second}\ (\mathsf{pair}\ E\ F) = (\lambda x.(x\ \lambda yz.z))\lambda z.(z\ E\ F) =$$

$$= (\lambda z.(z\ E\ F))\ (\lambda yz.z) = (\lambda yz.z)\ E\ F = F$$

5.3 *n*-tuples

The standard notation for tuples is $\langle E_1, E_2, \ldots, E_n \rangle$. The constructor and selector functions are analogous to those of pairs:

$\mathsf{n\text{-}tuple} \equiv \lambda x_1 x_2 \ldots x_n z.(z x_1 x_2 \ldots x_n)$
$\mathsf{select_i} \equiv \lambda x.(x(\lambda x_1 x_2 \ldots x_n.x_i)) \qquad (1 \leq i \leq n)$

5.4 Lists

The list of expressions E_n, E_{n-1}, ..., E_1 is denoted by $[E_n, E_{n-1}, \ldots, E_1]$. It is a recursive data structure. A possible representation uses pairs: a list is a pair, where the first element is true, if it is an empty list and false otherwise. The second element of the pair is relevant only in case of non-empty lists: this is also a pair consisting of the element and the representation of the *tail* of the list, which contains all but the first element of the list. Figure 6 shows the graphical representation of this structure.

This data structure has two constructors: nil to construct an empty list and cons to append a new element to the front of a list. The definitions of these functions according to the above representation are as follow:

$\mathsf{cons} \equiv \lambda xy.(\mathsf{pair\ false}(\mathsf{pair}\ x\ y))$,
$\mathsf{nil} \quad \equiv \mathsf{pair\ true\ true}$.

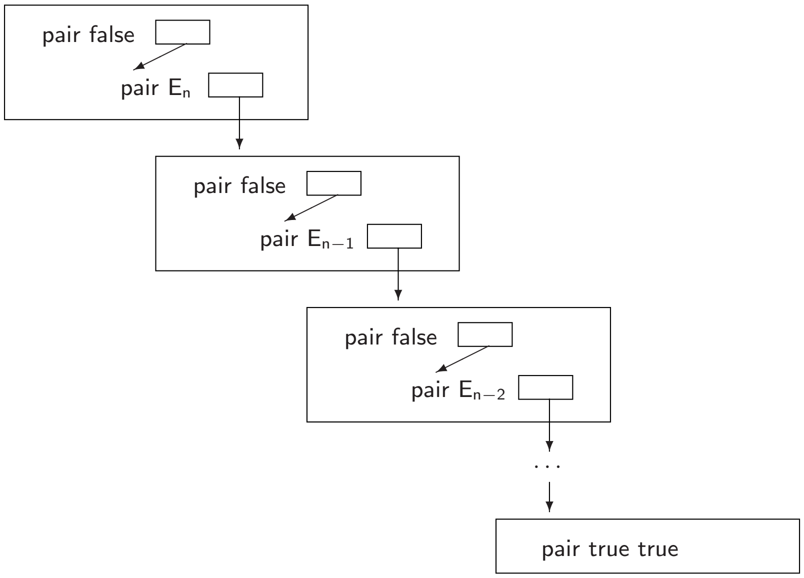


Fig. 6. An implementation of the list

There are two standard selectors: head gives the first element of a non-empty list and the tail function that gives the list of all elements except the first one.

$$\begin{aligned}&\mathsf{head} \equiv \lambda x.(\mathsf{first}\ (\mathsf{second}\ x)),\\ &\mathsf{tail}\ \ \equiv \lambda x.(\mathsf{second}(\mathsf{second}\ x)).\end{aligned}$$

5.5 Numeral Systems

In this section we deal with representation of natural numbers. In order to fix the set of functions we want to use, we first define what is a *numeral system* and an *adequate numeral system.*

Definition 39. *A* ***numeral system*** *consists of a sequence of numbers* $\ulcorner \mathsf{0} \urcorner, \ulcorner \mathsf{1} \urcorner, \ldots$ *and functions* succ *and* zero*, such that*

$$\mathsf{succ}\ \ulcorner \mathsf{i} \urcorner = \ulcorner \mathsf{i}+1 \urcorner, \qquad \textit{if}\ \ 0 \le i,$$

$$\mathsf{zero}\ \ulcorner \mathsf{i} \urcorner = \begin{cases} \mathsf{true}, & \textit{if}\ i = 0,\\ \mathsf{false}, & \textit{otherwise.}\end{cases}$$

Definition 40. *A numeral system is* ***adequate*** *if there exists a* pred *function, such that*

$$\mathsf{pred}\ \ulcorner \mathsf{i} \urcorner = \begin{cases} \ulcorner i-1 \urcorner, & \textit{if}\ i \ge 1,\\ \mathsf{false}, & \textit{otherwise.}\end{cases}$$

There are several ways to represent numeral systems in λ-calculus. In some of them the representation of the numbers is simple and easy to understand, while others are optimised for effective implementation of arithmetic functions. Here we present three well-known systems.

A simple numeral system
This is a simplest representation which encodes the numbers with a list-like data structure: the length of the list is the represented number. Here we do not need to store elements in the list, because only the length of the list is important. The structure can be seen on figure 7.

The corresponding function definitions are the following:

$$\begin{aligned}&\ulcorner 0 \urcorner \ \equiv \lambda x.x \quad \equiv \quad \mathsf{I},\\ &\mathsf{succ} \equiv \lambda y.(\mathsf{pair}\ \mathsf{false}\ y)\\ &\mathsf{zero} \equiv \lambda x.(x\ \mathsf{true}),\\ &\mathsf{pred} \equiv \lambda x.(x\ \mathsf{false}).\end{aligned}$$

The sequence of numbers in this system is the following:

$$\mathsf{I},\ \mathsf{pair}\ \mathsf{false}\ \mathsf{I},\ \mathsf{pair}\ \mathsf{false}\ (\mathsf{pair}\ \mathsf{false}\ \mathsf{I}),\ \ldots$$

Scott numerals
This numeral system uses abstractions to encode the numbers. The Scott numerals are the following:

$$\lambda xy.x,\ \ \lambda xy.(y\ \lambda xy.x),\ \ \lambda xy.(y\ (\lambda xy.y\ \lambda xy.x)),\ \ \ldots$$

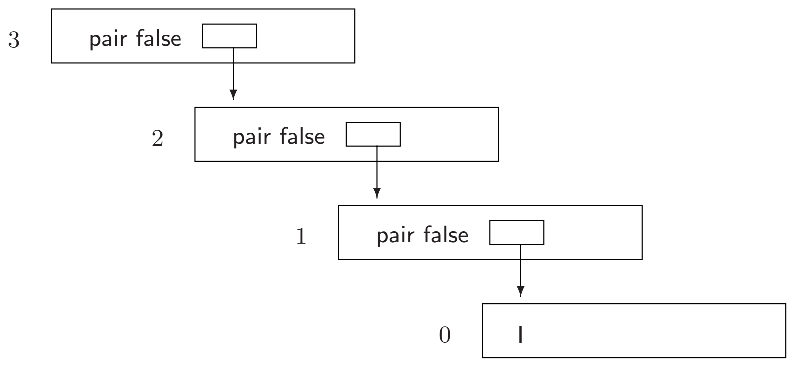


Fig. 7. An implementation of the simple numeral system

The definition of the functions are as follows:

$$\ulcorner 0 \urcorner \equiv \lambda xy.x \equiv \mathsf{true},$$
$$\mathsf{succ} \equiv \lambda zxy.(y\ z),$$
$$\mathsf{zero} \equiv \lambda x.(x\ \mathsf{true}(\lambda y.\mathsf{false})),$$
$$\mathsf{pred} \equiv \lambda x.(x\ \ulcorner 0 \urcorner(\lambda y.y)) \equiv \lambda x.(x\ \ulcorner 0 \urcorner\ \mathsf{I}).$$

Church numerals

This system uses iterated applications to encode numerals. Let us first present the construction functions:

$$\ulcorner 0 \urcorner \equiv \lambda fx.x,$$
$$\mathsf{succ} \equiv \lambda nfx.(f(nfx)).$$

Now we compute the first four numerals to see their representation:

$$\begin{aligned}
\ulcorner 0 \urcorner &\equiv & &\lambda fx.x,\\
\ulcorner 1 \urcorner &\equiv \mathsf{succ}\ \ulcorner 0 \urcorner &= &\lambda fx.(f(x)),\\
\ulcorner 2 \urcorner &\equiv \mathsf{succ}\ \ulcorner 1 \urcorner &= &\lambda fx.(f(f(x))),\\
\ulcorner 3 \urcorner &\equiv \mathsf{succ}\ \ulcorner 2 \urcorner &= &\lambda fx.(f(f(f(x)))),\\
\dots & \quad \dots & &\dots
\end{aligned}$$

We can see, that in general, the representation of the numeral n is $\ulcorner n \urcorner \equiv \lambda f\ x.(f^n(x)) \quad (n = 0, 1, 2, \dots)$, and for any $n \in \mathbb{N}$ and $E, F \in \Lambda$:

$$\ulcorner n \urcorner EF \twoheadrightarrow \underbrace{E(E(\dots E(E}_{n}\ F)\dots)),$$

which is the *n-times iterated application of E and F.*

The corresponding test function can be defined as follows:

$$\mathsf{zero} \equiv \lambda x.(x(\mathsf{true\ false})\mathsf{true}) \equiv \lambda x.(x(\lambda y.\mathsf{false})\mathsf{true}).$$

We note that the same representation can be achieved by another succ function:

$\mathsf{succ}' \equiv \lambda nfx.(n\ f\ (f\ x))$

Finally, we give the definition of the pred function:

$\mathsf{pred} \equiv \lambda n.(\lambda f\ x.(\mathsf{second}(n(\mathsf{pref}\ f)(\mathsf{pair}\ x\ x))))$,
where
$\mathsf{pref} \equiv \lambda fp.(\mathsf{pair}(f(\mathsf{first}\ p))(\mathsf{first}\ p))$.

The Church numerals are of interest because we can define powerful arithmetic functions without recursion:

$\mathsf{add} \equiv \lambda xypq.(x\ p(ypq))$,
$\mathsf{mul} \equiv \lambda xyp.(x(yp))$,
$\mathsf{exp} \equiv \lambda xy.(y\ x)$.

5.6 Extending the λ-Calculus

Although it is possible to represent data with λ-expressions, it is inefficient to do so. The idea is to add the **constants** and then to specify rules. A way of introducing computation rules to the λ-calculus is via **δ-rules**.

For example, instead of using a numeral system to compute the sum of two numbers, we can introduce constants for natural numbers and δ-rules for addition:

$$\mathsf{add}\ \ulcorner 1 \urcorner\ \ulcorner 2 \urcorner \equiv +\ 1\ 2\ \rightarrow_{\delta}\ 3$$

If we implement λ-calculus, these δ-rules can be the basic instructions of the machine we use.

6 Recursion

The usual definition of the factorial function is the following.

$\mathit{fac}(n) = \mathit{if}\ (n = 0)\ \mathit{then}\ 1\ \mathit{else}\ (n\ *\ \mathit{fac}\ (n-1))$

If we transform it to a λ-term, we get:

$\mathsf{fac} \equiv \lambda n.\mathsf{if}(=n\ 0)1(*\ n(\mathsf{fac}(-\ n\ 1)))$.

This is a recursive definition, because the function occurs in the body of its definition. If we perform an abstraction on the fac function itself, we get the following function:

$\mathsf{H} \equiv \lambda f.(\lambda n.\mathsf{if}(=n\ 0)1(*\ n(f(-\ n\ 1))))$

As a consequence, the following statement holds.

$\mathsf{fac} = \mathsf{H}\ \mathsf{fac}$.

6.1 Fixed Points

Definition 41. *If $E = FE$ then E is called a **fixed point** of F.*

According to this definition, fac is a fixed point of H. The question is, whether this fixed point exists in general? The next theorem answers this question positively.

Theorem 42 (Fixed point theorem). *For all E there is an F such that $F = EF$*

This means, that we are allowed to write recursive function definitions of the form

$$fun = D, \text{ where } D \text{ contains } fun,$$

because we can always define

$$\mathsf{H} \equiv \lambda f.D[fun := f],$$

and the fixed point theorem ensures that the equation

$$fun = \mathsf{H}\ fun$$

has a solution. The recursive definition defines the λ-term that is the solution of this equation.

Sometimes we define recursive functions using multiple recursive equations. The next theorem states that these equation systems also have solution.

Theorem 43 (Multiple fixed point theorem). *For all $E_1, E_2, \ldots, E_n$ there are $F_1, F_2, \ldots, F_n$ such that*

$$\begin{array}{lll}
F_1 & = & E_1\ F_1\ F_2 \ldots F_n, \\
F_2 & = & E_2\ F_1\ F_2 \ldots F_n, \\
\ldots & & \\
F_n & = & E_n\ F_1\ F_2 \ldots F_n.
\end{array}$$

6.2 Fixed-Point Operators

Using fixed-point operators, we can "compute" the solution of recursive equations.

Definition 44. *The λ-expression fix, that for all F satisfies $\mathsf{fix}\ F = F(\mathsf{fix}\ F)$, is called a **fixed-point operator**.*

There are many different well-known fixed-point operators, the most famous ones are the following:

Russell:
$\mathsf{Y} \equiv \lambda x.(\lambda y.x(y\ y))(\lambda y.x(y\ y))$.

Turing:
$\Theta \equiv (\lambda x\ y.y(x\ x\ y))(\lambda x\ y.y(x\ x\ y))$.

Klop:
$\pounds \equiv \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$,
$\$ \equiv \pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds\pounds$.

This allows us to transform recursive functions to a non-recursive form. For example, the factorial function can be written as follows:

$$\mathsf{fac} = \mathsf{Y}\ \mathsf{H}, \text{ where } \mathsf{H} \equiv \lambda f.(\lambda n.\mathsf{if}(=n\ 0)1(*\ n(f(-\ n\ 1)))).$$

The next reduction sequence shows, that the transformed expression really computes factorial.

$$\mathsf{fac}\ 2 = \mathsf{Y}\ \mathsf{H}\ 2 \twoheadrightarrow *2(*11)\ \twoheadrightarrow_\delta\ 2.$$

7 λ-Definable Functions

In this section we show the relation of λ-calculus to different function-classes and that λ-calculus has the same expressive power as *Turing machines* have.

7.1 Primitive Recursive Functions

A **numeric function** is a mapping $f : \mathbb{N}^n \to \mathbb{N}$ for some $n \in \mathbb{N}$.

Definition 45. *Let f be a numeric function with n arguments. f is* ***λ-definable*** *if for some $F \in \Lambda$ and for all $x_1, x_2, \ldots, x_n \in \mathbb{N}$*

$F\ulcorner x_1\urcorner\ulcorner x_2\urcorner \ldots \ulcorner x_n\urcorner = \ulcorner f(x_1, x_2, \ldots, x_n)\urcorner$.

In this case f is said to be λ-defined by F.

Definition 46. *The* ***initial functions*** *are the numeric functions*

- $Z(x) = 0$,
- $succ(x) = x + 1$,
- $U_i^n(x_1, x_2, \ldots, x_n) = x_i, \quad 0 \leq i \leq n$.

Definition 47. *Let P be a class of numeric functions. P is closed under* ***substitution*** *(also called* ***composition****) if for all $g, h_1, h_2, \ldots, h_m \in P$*

$$g(h_1(x_1, x_2, \ldots, x_n), h_2(x_1, x_2, \ldots, x_n), \ldots, h_m(x_1, x_2, \ldots, x_n)) \in P.$$

Definition 48. *Let P be a class of numeric functions. P is closed under* ***primitive recursion*** *if for all $g, h \in P$ and*

$$\begin{aligned} f(0, x_2, \ldots, x_n) &= g(x_2, \ldots, x_n) \\ f(succ(x_1), x_2, \ldots, x_n) &= h(f(x_1, x_2, \ldots, x_n), x_1, x_2, \ldots, x_n), \end{aligned}$$

then $f \in P$ holds.

Definition 49. *The class of* ***primitive recursive functions*** *is the least class of numeric functions which contains all of initial functions and is closed under composition and primitive recursion.*

Theorem 50 (Kleene). *The primitive recursive λ-definable numeric functions are exactly the primitive recursive functions.*

7.2 Total Recursive Functions

Definition 51. *Let P be a class of numeric functions. P is closed under* ***minimization*** *(also called* ***inversion****) if for all $g \in P$*

$\mu y[g(y, x_1, x_2, \ldots, x_n) = 0] \in P$ holds,

where

$\mu y[g(y, x_1, x_2, \ldots, x_n)$

denotes the least number y such that $g(y, x_1, x_2, \ldots, x_n) = 0$.

Definition 52. *The class of* ***total recursive functions*** *is the least class of numeric functions which contains all of the initial functions and is closed under composition, primitive recursion and minimization.*

Theorem 53 (Kleene). *The total recursive λ-definable numeric functions are exactly the total recursive functions.*

7.3 Partial Recursive Functions (1. Part)

Partial recursive functions may be *undefined* for some arguments. The question is, how to represent the undefinedness in λ-calculus? The *classical proposal* of Church was, that λ-terms with no normal form should be used for this purpose.

Definition 54. *Let f be a partial numeric function with n arguments. f is* ***λ-definable*** *if for some $F \in \Lambda$ and for all $x_1, x_2, \ldots, x_n \in \mathbb{N}$*

$$F \ulcorner x_1 \urcorner \ulcorner x_2 \urcorner \ldots \ulcorner x_n \urcorner \begin{cases} = \ulcorner m \urcorner, \quad \text{if} \quad f(x_1, x_2, \ldots, x_n) = m, \\ \text{has no normal form}, \\ \qquad\qquad \text{if} \quad f(x_1, x_2, \ldots, x_n) \quad \text{is undefined.} \end{cases}$$

As we have seen in section 4, Ω does not have a normal form, and neither $\lambda x.(x\ \Omega)$ have one. So, according to the previous definition, $\lambda x.(x\ \Omega)$ is undefined. However, if we apply it to the constant function $\lambda x.\ulcorner 0 \urcorner$, the result is $\ulcorner 0 \urcorner$, not undefined. This is somewhat undesirable to get a defined term by applying an undefined function.

Solvability
The previous problem is one of the many *disadvantages* of the definition by Church. For that reason Barendregt and Wadswort in 1971 proposed, that *solvability* should be used: according to their proposal, *unsolvable terms represent the notion "undefined"*.

Definition 55. *$E \in \Lambda^0$ is **solvable** if there exist $F_1, F_2, \ldots, F_n \in \Lambda \quad (n \geq 0)$ such that*

$$EF_1F_2 \ldots F_n = \mathsf{I}.$$

For example it turns out that $\lambda x.(x\ \Omega)$ is solvable, because applying it to $\lambda x.\mathsf{I}$ results in I.

Lemma 56. *E is solvable iff $\lambda x.E$ is solvable.*

That is, $E \in \Lambda$ is solvable if the closure of E is solvable.

Lemma 57. *If EF is solvable then E is solvable.*

Lemma 58. *If E is unsolvable then so are $\lambda x.E,\ EF,\ E[x := F]$ for all F.*

Head normal form

Definition 59. *A term E is a **head normal form** if E is of the form $\lambda x_1x_2 \ldots x_n.xF_1F_2 \ldots F_m \quad (n, m \geq 0)$*

where x is a variable or δ-function and $xF_1F_2 \ldots F_p$ is not a redex for all $p \leq m$.

If $E \equiv \lambda x_1x_2 \ldots x_n.(\underline{(\lambda x.F_0)F_1})F_2 \ldots F_m$, then the underlined $(\lambda x.F_0)F_1$ is called the head redex of E.

Theorem 60 (Wadsworth, 1971). *E has a head normal form iff E is solvable.*

For example an other way of showing that $\lambda x.(x\ \Omega)$ is solvable is to notice that it is in head normal form and then to use the previous theorem.

Weak head normal form

Definition 61. *A term E is a **weak head normal form** if E is of the form $\lambda x.F$*

or

$xF_1F_2 \ldots F_m \quad (n, m \geq 0)$

where x is a variable or δ-function and $xF_1F_2 \ldots F_p$ is not a redex for all $p \leq m$.

For example, $\lambda x.(\lambda y.(x +_\lambda y)\ \ulcorner 2 \urcorner)$ is in weak head normal form, however it is not head normal form. We have to perform a reduction in the body of the expression to get $\lambda x.(x +_\lambda \ulcorner 2 \urcorner)$, which is already in head normal form.

In implementations of functional programming languages, the evaluation is stopped as soon as the result is known to be weak head normal form (see Figure 8).

$$\begin{tikzcd}
\boxed{\lambda\text{-expression}} \arrow[d, "\textit{reduction of top redexes, no } \alpha\textit{-conversion is needed}"] \\
\boxed{\text{weak head normal form}} \arrow[d, "\textit{reduction of head redexes, } \alpha\textit{-conversion}"] \\
\boxed{\text{head normal form}} \arrow[d, "\textit{reduction of inner redexes, } \alpha\textit{-conversion}"] \\
\boxed{\text{normal form}}
\end{tikzcd}$$

Fig. 8. Normal forms and reductions

7.4 Partial Recursive Functions (2. Part)

Definition 62. *Let f be a partial numeric function with n arguments. f is **λ-definable** if for some $F \in \Lambda$ and for all $x_1, x_2, \ldots, x_n \in \mathbb{N}$*

$$F\ulcorner x_1\urcorner\ulcorner x_2\urcorner\ldots\ulcorner x_n\urcorner \begin{cases} = \ulcorner m\urcorner, & \textit{if} \;\; f(x_1, x_2, \ldots, x_n) = m, \\ \textit{is unsolvable}, & \\ & \textit{if} \;\; f(x_1, x_2, \ldots, x_n) \;\; \textit{is undefined.} \end{cases}$$

Using this definition we can eliminate the problem pointed out in section 7.3: $\lambda x.(x\ \Omega)$ is *not* an undefined term. Using this representation of undefinedness, the following fundamental theorems hold.

Theorem 63 (Kleene). *The λ-definable numeric functions are exactly the partial recursive functions.*

Theorem 64 (Turing, 1937). *The classes of Turing computable function is the same as the class of λ-definable functions.*

So the power of Turing machines is the same as the power of the λ-calculus. This means that both models capture the intuitive idea of computation.

References

1. Barendregt, H.P.: The Lambda Calculus, Its Syntax and Semantics. North Holland, Amsterdam (1984)
2. Bird, R., Wadler, P.: Introduction to Functional Programming. Prentice Hall, Englewood Cliffs (1988)
3. Church, A.: A set of postulates for the foundation of logic. Annals of Math. 33, 34, 346–366, 839–864 (1932/1933)

4. Csörnyei, Z.: Lambda-kalkulus (in Hungarian). Typotex (2007)
5. Curry, H.B.: Functionality in combinatory logic. Proc. Nat. Acad. Science USA 20, 584–590 (1934)
6. Felleisen, M., Flatt, M.: Programming Languages and Lambda Calculi. Course Notes, Utah University (2003)
7. Goldberg, M.: An Introduction to the Lambda Calculus. Course Notes. Ben-Gurion University (2000)
8. Hankin, C.: Introduction to Lambda Calculi for Computer Scientists. Imperial College London (2004)
9. Harrison, J.: Introduction to Functional Programming. Course Notes. University of Cambridge (1997)
10. Hindley, J.R., Seldin, J.P.: Introduction to Combinators and λ-Calculus. Cambridge University Press, Cambridge (1986)
11. Ker, A.D.: Lambda Calculus. Course Notes. Oxford University, Oxford (2003)
12. Kleene, S.C., Rosser, J.B.: The inconsistency of certain formal logics. Annals of Math. 36, 630–636 (1936)
13. Kleene, S.C.: λ-definability and recursiveness. Duke Math. J. 2, 340–353 (1936)
14. Kluge, W.: Abstract Computing Machines. Springer, Heidelberg (2005)
15. Mitchell, J.C.: Foundations for Programming Languages. MIT Press, Cambridge (1996)
16. Ong, C.-H.L.: Lambda Calculus. Course Notes. Oxford University, Oxford (1997)
17. Paulson, L.C.: Foundations of Functional Programming. Course Notes. University of Cambridge (1996)
18. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
19. Pierce, B.C. (ed.): Advanced Topics in Types and Programming Languages. MIT Press, Cambridge (2005)
20. Plasmeijer, R., van Eekelen, M.: Functional Programming and Parallel Graph Rewriting. Addison-Wesley, Reading (1993)
21. Roversi, L.: λ-calculus as a Programming Language. Course Notes. Università di Torino (1999)
22. Schönfinkel, M.: Über die bausteine der mathematischen logik. Math. Annalen 92, 305–316 (1924)
23. Turing, A.M.: Computability and λ-definability. J. Symbolic Logic 2, 153–163 (1937)

Abstract λ-Calculus Machines

Werner E. Kluge

Department of Computer Science
University of Kiel
D–24105 Kiel, Germany
wk@informatik.uni-kiel.de

Abstract. This paper is about fully normalizing λ-calculus machines that permit symbolic computations involving free variables. They employ full-fledged β-reductions to preserve static binding scopes when substituting and reducing under abstractions. Abstractions and variables thus become truly first class objects: both may be freely substituted for λ-bound variables and returned as abstraction values. This contrasts with implementations of conventional functional languages which realize a weakly normalizing λ-calculus that is capable of computing closed terms (or basic values) only.

The two machines described in this paper are descendants of a weakly normalizing SECD-machine that supports a nameless λ-calculus which has bound variable occurrences replaced by binding indices. Full normalization is achieved by a few more state transition rules that η-extend unapplied abstractions to full applications, inserting in ascending order binding indices for missing arguments. Updating these indices in the course of performing β-reductions is accomplished by means of a simple counting mechanism that inflicts very little overhead. Both machines realize a head-order strategy that emphasizes normalization along the leftmost spine of a λ-expression. The simpler FN_SECD-machine abides by the concept of saving (and unsaving) on a dump structure machine contexts upon each individual β-reduction. The more sophisticated FN_SE(M)CD-machine performs what are called β-reductions-in-the-large that head-normalize entire spines in the same contexts. It also employs an additional trace stack M that facilitates traversing spines in search for and contracting redices.

The paper also gives an outline of how the FN_SE(M)CD-machine can be implemented as a graph reducer.

1 Introduction

Abstract computing machines are conceptual models of program execution. They exhibit the runtime structures and the basic operating and control mechanisms that are absolutely essential to perform computations specified by particular (classes of) programming languages. They may be considered common interfaces, or intermediate levels of program execution, shared by a variety of real computing machines irrespective of their specific architectural features. The level of abstraction may range from direct interpretation of the constructs of a (class of) language(s) to (compiling to) abstract machine code composed of some minimal set of instructions that suffices to perform some basic operations on the runtime structures and to exercise control over their sequencing.

Our interest in abstract λ-calculus machines derives from the fact that the λ-calculus is at the core of all algorithmic programming languages, procedural or functional, as we know them today. It is a theory of computable functions that talks about elementary properties of and the application of operators to operands and, most importantly, about the role of variables in this game [Chu41, Bar84, HS86]. In its purest form it knows only three syntactical figures – variables, abstractions (of variables from expressions) and applications (of operator to operand expressions) – and a single rule for transforming λ-expressions into others. This β-reduction rule, which specifies the substitution of variables by expressions, tells us in a nutshell the whole story about computing. The runtime structures that are involved in reducing λ-expressions are shared, in one form or another, by abstract machines for all algorithmic languages, particularly in the functional domain, and so are the basic mechanisms that operate on these structures. Understanding λ-calculus machines therefore is fundamental to comprehending the why and how of organizing and performing computations by machinery.

The very first machine of this kind, which has become more or less a standard model, is the SECD-machine proposed by Landin as early as 1964 [Lan64]. It is named after the four runtime structures it employs, of which the most important ones, besides a code structure C, are an environment E and a dump D which facilitate efficient substitutions while maintaining correct binding scopes. The machine is said to be weakly normalizing, meaning that substitutions and reductions under abstractions are outlawed in order to avoid the seeming complexity of full-fledged β-reductions which would be required to resolve potential naming conflicts between free variable occurrences in arguments and variables bound by the abstractions. It is due to this restriction that the SECD-machine cannot really compute abstractions as values but must represent them as closures, i.e., as unevaluated abstractions embedded in the environments that hold instantiations of their (relatively) free variables [1].

It can justifiably be argued that this restriction, for all practical purposes, is of minor relevance if we are mainly interested in computing basic values (or ground terms) only, which is what real-life application programming overwhelmingly is all about. In fact, all implementations of functional languages are based on weakly normalizing machinery with a naive parameter passing (or substitution) mechanism, well known examples being the G-machine, the STG-machine, the Functional Abstract Machine (FAM) or the Categorial Abstract Machine (CAM) [Joh84, PeyJ92, CMQ83, CCM85/87]. Implementations of procedural languages go even one step further by demanding that functions (procedures) be

[1] We refer to a variable as being relatively free if it is free in a particular subexpression under consideration but bound higher up in a larger, surrounding expression.

legitimately applied to full sets of arguments only. Moreover, variables represent values but are not values themselves, as in the λ-calculus.

However, there are some benefits to supporting a fully normalizing λ-calulus based on a full-fledged β-reduction. Resolving naming conflicts between free and bound variable occurrences is the key to correctly performing symbolic computations as both variables and functions (abstractions) can then truly be treated as first class objects in the sense that both may be passed as function parameters and returned as function values.

This quality may be advantegeously employed, for instance, in term rewrite systems or, more specifically, in proof systems where establishing semantic equality between two terms containing free variables is an important proof tactics. Another useful application of full normalization is in the area of high-level program optimizations, e.g., by converting partial function applications into new, specialized functions with normalized bodies. Such optimizations could pay off significantly in terms of runtime performance if the specialized functions are repeatedly called in different contexts.

This paper is to show how fully normalizing abstract λ-calculus machines can be derived from standard SECD-machinery by a few minor extensions and modifications, and how these machines can be taken as blueprints for the design of equivalent graph reduction machines whose runtime efficiencies are competitive with those of its weakly normalizing counterparts.

To do so, we will proceed as follows: In the next section we will look at a very simple program to illustrate some of the shortcomings of current implementations of functional languages in order to make a case for supporting a fully normalizing λ-calculus. Section 3 introduces a normal-order SECD-machine which supports a nameless λ-calculus that has bound variables replaced by binding indices. In section 4 we will first outline the concept of head-order reductions (which is just a particular way of looking at normal-order evaluation) and then introduce in section 5 a fully normalizing FN_SECD-machine that differs from its weakly normalizing counterpart by the addition of a few more state transition rules that primarily deal with unapplied abstractions.

In section 6 we will introduce a more sophisticated FN_SE(M)CD-machine that performs what are called head-order reductions-in-the-large. It engages the dump only when entering (or returning from) the evaluation of so-called suspensions [2] and also employs an additional trace stack M for apply nodes and abstractors encountered while traversing an expression in search for β-redices. Section 7 outlines the workings of a fully normalizing graph reducer that derives from this FN_SE(M)CD-machine.

2 Some Simple Exercises in Functional Programming

To motivate what we are trying to accomplish, let's have a look at several variants of a very simple functional program, written in SCHEME [Dyb87], that exposes

[2] Loosely speaking, these are expressions embedded in their environments whose evaluation has been postponed under the normal-order strategy.

some of the problems of weak normalization. This program consists of the following two function definitions:

```
( define twice ( lambda ( f u ) ( f ( f u ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )
```

The function `twice` applies whatever is substituted for its first parameter `f` twice to whatever is substituted for its second parameter `u`, and the function `square` computes the square of a number substituted for its parameter `v`.

When applying `twice` to `square` and `2`, a SCHEME interpreter returns, as one would expect,

```
( twice square 2 ) --> 16
```

i.e., the square of the square of 2. But if `twice` is just applied to either `square` or to itself, we get

```
( twice square ) --> procedure twice: expects 2 args,
                     given 1 : ( lambda(a1) ... )

( twice twice )  --> procedure twice: expects 2 args,
                     given 1 : ( lambda(a1) ... )
```

i.e., the interpreter notifies us in both cases of attempts to apply a function of two parameters to just one argument, indicating that the result is a function of one parameter that is artificially introduced as `a1`, but it cannot return a full function body in SCHEME notation.

The same happens with the application

```
( twice twice square ) --> procedure twice: expects 2 args,
                           given 1 : ( lambda(a1) ... )
```

though here `twice` is applied to two arguments, so everything should work out. However, the problem now arises in the body of `twice` where the parameter `f` is applied to just one parameter `u`, but `f` is substituted by `twice` itself, which expects two arguments. Again, the result is a function of one parameter `a1`, as one would expect, whose body cannot be made explicit.

We now slightly modify the function `twice`, turning it into curried form (i.e., into a nesting of unary functions), and see what happens then.

```
    ( define twice ( lambda ( f )
                   ( lambda ( u ) ( f ( f u ) ) ) ) )
```

When matching the curried version of `twice` by corresponding nestings of applications, as for instance in

```
    ( ( twice square ) 2 )           -->  16
```

or in

```
( ( ( twice twice ) square ) 2 ) -->  65536
```

we obviously get the expected results. However, when applying `twice` to two arguments, as in

```
( ( twice twice square ) 2 )
                --> procedure twice: expects 1 arg,
                    given 2 : ( lambda(a1) ... )
```

the interpreter complains about a unary function being applied to two arguments, the result of which is a function of one parameter (which is correct) whose body, again, cannot be returned in SCHEME notation.

In the following two applications, we have no mismatching arities,

```
( twice twice )              -->  ( lambda (a1) ... )

( ( twice twice ) square )   -->  ( lambda (a1) ... )
```

Here again we are only told that the result is a function of one parameter, but the function body is not disclosed.

However, what one would wish to see as output of these latter two applications, and what a fully normalizing λ-calculus would readily deliver, is something like this:

```
( twice twice ) --> ( lambda ( u') ( lambda ( u )
                      ( u'( u'( u'( u' u ) ) ) ) ) )

( ( twice twice ) square )
        --> ( lambda ( u )
              ( * ( * ( * ( * u u ) ( * u u ) )
              ( * ( * u u ) ( * u u ) ) ) ( .... ) ) )
```

i.e., the self-application of `twice` should return in high-level notation a function that could be called `double-twice` as it applies four times its first to the second parameter [3]. Applying this self-application to `square` should return a function of one parameter (which is expected to be substituted by a number) that is multiplied 16 times by itself. Both functions may be considered spezialized versions of the original partial applications. They may be applied in different contexts without going repeatedly through the motions of evaluating them as parts of full applications, i.e., these functions are in fact optimized.

The unfortunate state of affairs of not being able to compute functions truly as function values, let alone returning them in the above form as output, is

[3] Note that evaluating this self-application produces a naming conflict between a bound and a relatively free occurrence of the variable `u` which must be resolved by renaming either one of them as `u'`.

common to all current implementations of functional languages, e.g., HASKELL, CLEAN, ML or SCHEME [Bird98, PvE93, Ull98, Dyb87]. Little is accomplished if the programmer is just informed that the result of some computation (that generally may be rather complex) is a function, without telling what the function looks like, i.e., what exactly it computes [4]. This deficiency is a direct consequence of compiling, for reasons of runtime efficiency, programs of these languages to code of some abstract or real machine. Such code being static, it expects the right things (the objects of the computation) to be in the right places (memory locations) at the right time (or state of control). More specifically, it means that, as the above examples indicate, function (abstraction) code can execute correctly if and only if it can access at prefixed locations relative to the top of the runtime stack a full set of arguments (of the right types), i.e., an actual for each of its formal parameters. Otherwise, code execution must either be suspended until missing arguments can be picked up later on, or the user must be notified, as in the above examples, that the computation is getting stuck in a state that cannot be decompiled into a legitimate program expression.

This is to say that, in λ-calculus terminology, these languages in fact feature a weakly normalizing semantics that is more or less imposed by the constraints of compiling to static code: a function application can only be evaluated if the function's arity matches the number of arguments supplied; a partial function application may have its arguments evaluated but nothing can be done beyond that since neither substitutions under the (remaining) abstraction nor evaluation of the abstraction body are permitted.

Static code seems to leave no room for the flexibility that is required to support full normalization, in which case the code would have to deal with partial applications, i.e., with varying numbers of arguments on the stack, and with free variables (which are their own values). Also, new code would have to be generated at runtime for new functions that are being computed by application of existing ones. Though these things can be done in principle, it is generally believed that they are difficult to implement, degrading runtime performance considerably, and therefore considered a luxury that is not really needed.

However, in the following we will show that full normalization can be achieved with little effort, in terms of additional machinery, beyond what is necessary to perform weakly normalizing computations.

3 A Weakly Normalizing λ-Calculus Machine

A good starting point for the design of a fully normalizing λ-calculus machine is Landin's classical SECD-machine [Lan64]. It is an abstract applicative order evaluator that reduces λ-expressions to weak normal forms. The operating principles of this machine are based on the ideas of delayed substitutions, environments and, related to it, the notion of closures.

[4] Typed languages such as HASKELL, CLEAN or ML can at least infer the type of the resulting function which, however, is not of much help either.

The concept of delayed substitutions is to split β-reductions up into two steps that are distributed over space and time. Upon encountering β-redices, generally several in succession, the machine just collects in an environment structure the operand expressions to be substituted. All substitutions are then done in one sweep through the abstraction body by looking the operands up in the environment. Closures are special constructs that, loosely speaking, pair abstractions with the environments in which they may have to be evaluated later on.

We will first show how the SECD-machine can be modified to support normal-order evalution which guarantees termination with weak normal forms, so they exist, and then upgrade it to reduce λ-expressions to full normal forms.

3.1 A Machine-Compatible Syntax for λ-Expressions

We begin the construction of a normal-order SECD-machine with the choice of a suitable syntax for λ-expressions, taking into account that machines have a hard time dealing with variables and parentheses. We therefore use the nameless λ-calculus of deBruijn [Bru72] which replaces λ-bound variable occurrences with binding indices. We also switch to nameless abstractors Λ, replace left parentheses of applications with apply nodes @, and drop right parentheses altogether. The ensuing constructor syntax of what we in the following will refer to as the Λ-calculus thus looks like this:

$$e_\Lambda \ =_s \ \#i \mid \Lambda\ e_b \mid @\ e_f\ e_a$$

Expressions are deBruijn indices $\#i$, abstractions and normal-order applications, respectively. The apply node @ and the abstractor Λ are the constructors of this syntax.

DeBruijn indices may assume values $i \in \{\ 0, \ldots, n-1\ \}$, where n is the number of Λ-abstractors encountered along the path from the root node of the Λ-expression down to the occurrence of the index $\#i$. The index itself measures the distance, in terms of intervening Λs, to the one that binds it (with index $\#0$ being bound to the innermost Λ).

The expressions e_f and e_a are considered operator and operand, respectively, of an application. If the operator happens to be an abstraction, then it may alternatively be referred to as the function and the operand as the argument of the application [5].

In addition to Λ-expressions, the machine also works with two syntactical constructs $[\ E\ \Lambda\, e_b\ ]$ and $[\ E\ e\ ]$ which respectively are called closures and suspensions. They both pair expressions with the environments in which they may have to be evaluated. The difference between the two is that closures are specifically created for abstractions that occur in operator positions of applications,

[5] It should be noted that scanning an application from left to right is equivalent to traversing in pre-order the underlying binary tree structure, i.e., the apply node at the root is inspected first, followed by operator and operand as left and right subtrees, respectively, recursively in pre-order.

whereas suspensions are created for operand expressions, including abstractions, to delay their evaluation until called for by the normal-order regime later on. Syntactically, closures are just special suspensions.

3.2 The Basics of Doing β-Reductions

A brief illustration of how β-reductions are being processed by the abstract machine we are going to design is given in fig.1. It shows how the graph representation of the nested application $@\,@\,@\,\Lambda\Lambda\Lambda\, e_b\; e_1\; e_2\; e_3$ is step by step transformed, beginning in the upper left and following the thick arrows.

We assume that this nested application is part of a larger, surrounding expression, and that β-reductions performed in this expression have produced some

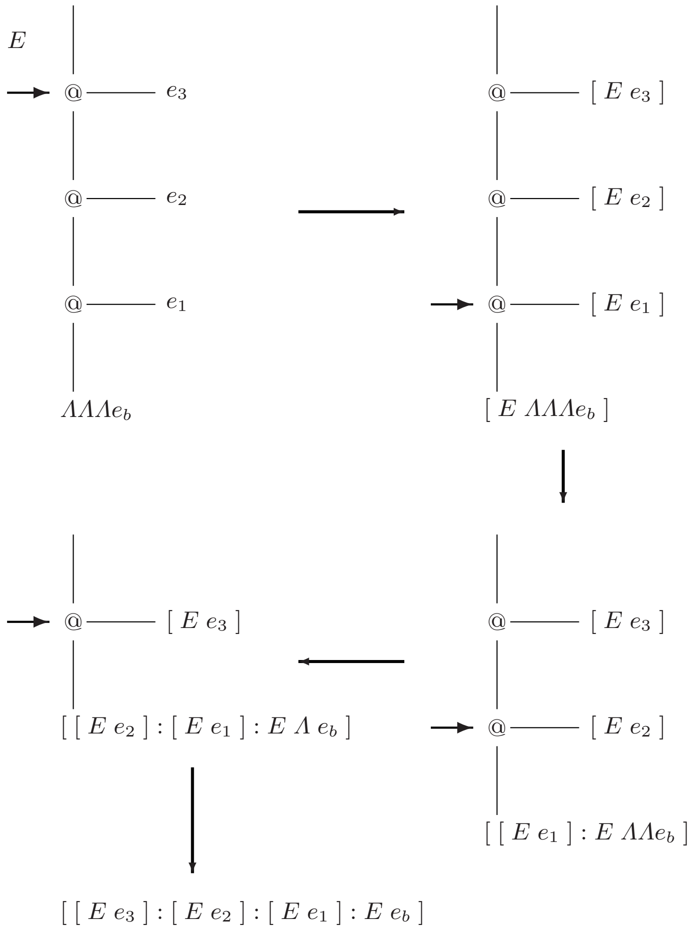


Fig. 1. Sequence of steps that reduces a nested application $@\,@\,@\,\Lambda\Lambda\Lambda\; e_b\; e_1\; e_2\; e_3$

environment E [6] once the focus of control has arrived at the outermost apply node under consideration, as indicated in the upper left graph by the little arrow pointing to it from the left. The entries in this environment are suspensions which may have to be substituted for deBruijn indices that occur free in the operand expressions $e_1,\ e_2,\ e_3$ and in the abstraction $\Lambda\Lambda\Lambda e_b$ in operator position.

As the focus of control moves down the spine of apply nodes, the environment E is distributed over the operand expressions $e_1,\ e_2,\ e_3$, creating suspensions in their places, and over the abstraction $\Lambda\Lambda\Lambda e_b$ in operator position, wrapping it up in a closure, as shown in the upper right graph.

It is important to note that at this point no attempts have been made to evaluate these constructs: the normal-order regime demands that, for the time being, the suspensions in operand positions be left untouched. The closure in operator position cannot be evaluated either as it would require substituting environment entries under an abstraction, which is outlawed under a weakly normalizing regime.

However, with the focus of control now pointing to the innermost apply node, we have an instance of a β-redex with an abstraction embedded in a closure in operator position and a suspension in operand position. Evaluating this application creates a new closure in its place that has one Λ removed from the abstraction and has the operand suspension prepended to the environment (denoted as $[\ E\ e_1\ ] : E$), as depicted in the graph at the lower right. Continuing in this way, the whole spine is consumed from the bottom up, resulting in a closure that has two more entries prepended to the original environment E which is now paired with an abstraction body e_b that is stripped off all Λ-abstractors (at the bottom of fig. 1). This being the case, the closure can now safely be evaluated by substituting all occurrences of deBruijn indices $\#i$ in e_b by the entries found i positions deep in the environment (counting from left to right and beginning with the index $i = 0$) as they are, i.e., without worrying about naming conflicts. We will refer to such substitutions, and in consequence also to β-reductions realized in this form, as being naive.

Note that we have chosen here the ideal case that the number of apply nodes along the spine matches the number of Λs in (or the arity of) the abstraction that is in the head of the spine, but the other cases are covered as well. If the number of apply nodes exceeds the abstraction's arity, then a shorter spine is left over with a closure as at the bottom of fig. 1 in its head. Should the arity of the abstraction exceed the number of apply nodes along the spine, i.e., we have a partial application, then we end up with a closure containing an abstraction of lesser arity that cannot be evaluated any further.

3.3 A Normal-Order SECD-Machine

The workings of an abstract machine are described by a set of machine states and a state transition function that maps (transforms) current into next states.

[6] If the application would be top level, the environment would be empty, denoted as *nil*.

A state, in turn, is described by a collection of dynamically changing data structures on which the machine operates.

The name of the SECD-machine derives from four stack-like structures that make up the machine states. These are

– a code structure C that holds Λ-expressions or fragments thereof in the order in which they need to be evaluated;
– a value stack S into which are pushed the values of expressions (or subexpressions);
– an environment structure E whose entries are suspensions that may have to be substituted for deBruijn indices that pop to the top of C;
– a dump stack D for entire machine states that are pushed and popped when entering and returning from β-reductions, respectively.

Thus, a state of the SECD-machine, to which we will also refer as a configuration, is defined by a quadruple (S, E, C, D), and the state transition function as:

$$\tau_{secd} : (\ S,\ E,\ C,\ D\ ) \ \rightarrow\ (\ S',\ E',\ C',\ D'\ )\ \ .$$

The actual contents of the stack-like runtime structures are specified as

$$stack\ =_s\ nil \mid X \mid item : stack\ \ ,$$

where nil denotes an empty stack, X stands for one of the stack symbols S, E, C, D, and $'\,:'$ separates some specific topmost symbol or expression from the rest of the stack.

The basic operating principle of this machine is to initially set up the entire Λ-expression in the code structure C, to evaluate recursively from innermost to outermost applications popping to the top of C, and to move their values over into S, where the resulting weak normal form is recursively constructed from the bottom up.

More specifically, an application $@\ e_f\ e_a$ on top of C is rearranged in post order as $e_a : e_f : @$ to have the operand evaluated before the operator and before the entire application. Following the normal-order regime, the value of e_a must be moved into S as a suspension $[\ E\ e_a\ ]$, followed by a closure $[\ E\ e_f\ ]$ if e_f happens to be an abstraction. The applicator $@$ then popping to the top of C forces the evaluation of the application, consuming its components from C and S and (eventually) pushing its value into S instead.

However, evaluating β-redices takes a number of intermediate steps that involve the environment and the dump. The operand suspension is prepended to the environment carried along with the closure that contains the abstraction, and the abstraction body is in isolation set up on top of C for further evaluation in this new environment. The latter is accomplished by saving on the dump the machine state that represents the entire surrounding context of the β-redex. This context in fact constitutes the return continuation with which the

Rearranging applications on C and creating suspensions on S
(1) $(S,\ E,\ @\ e_f\ e_a : C,\ D)\ \rightarrow ([\ E\ e_a\ ] : S,\ E,\ e_f : @ : C,\ D)$

Creating closures on S for abstractions on C
(2) $(S,\ E,\ \Lambda e_b : C,\ D)\ \rightarrow ([\ E\ \Lambda e_b\ ] : S,\ E,\ C,\ D)$

Substituting deBruijn indices
(3) $(S,\ E,\ \#i : C,\ D)\ \rightarrow (lookup(\ \#i,\ E\ ) : S,\ E,\ C,\ D)$

Entering naive β–reductions
(4) $([\ E'\ \Lambda e_b\ ] : e_a : S,\ E,\ @ : C,\ D)\ \rightarrow (S,\ e_a : E', e_b : nil,\ (E,\ C,\ D))$

Reducing suspensions not containing abstractions
(5) $([\ E'\ e'\ ] : S,\ E,\ C,\ D) \mid (\ e' \neq \Lambda e_b\ )\ \rightarrow (S,\ E',\ e' : nil,\ (E,\ C,\ D))$

Reconstructing irreducible applications in S
(6) $(\ e_b : e_a : S,\ E,\ @ : C,\ D)\ \rightarrow (@\ e_b\ e_a : S,\ E,\ C,\ D)$

Returning from naive β-reductions
(7) $(S,\ E,\ nil,\ (E',\ C',\ D')) \rightarrow (S,\ E',\ C',\ D')$

Fig. 2. The complete set of state transition rules for the normal-order SECD machine

computation must continue once evaluation of the β-redex is completed, whereupon its value ends up on S and the code structure C becomes empty.

The details of how this machine works are specified by the set of state transition rules given in fig. 2, which realizes the state transition function τ_{secd}. They are listed in the order in which they must be matched against actual machine states.

Rules (1) to (3) identify the machine configurations that have the three syntactical figures of legitimate Λ-expressions appear on top of the code structure C. Rule (1) splits an application up into its three components which are rearranged so that the apply node is squeezed underneath the operator, whereas the operand is embedded in a suspension that is pushed into S. Rule (2) wraps an abstraction up in a closure that is pushed into S. A deBruijn index on top of C accesses, by application of rule (3), the i-th entry relative to the top of the environment E, using a function *lookup*, and pushes it into S, which realizes the substitution that completes a naive β-reduction.

Rule (4) enters a (naive) β-reduction: an applicator @ on top of C in conjunction with a closure on S has the body of the abstraction isolated for evaluation in C together with its environment on E, while the current environment and the current code structure, i.e., the calling context, are saved as return continuation on the dump. The operand retrieved from underneath the closure in S, which is bound to be a suspension, is prepended as a new entry to what has now become the active environment. Evaluating an abstraction body involves traversing it step by step from C to S, thereby substituting deBruijn indices by environment entries or calling for other (naive) β-reductions. Completing this traversal is signified by an empty code structure, at which point rule (7) is called to return to the context saved on the dump.

Isolating in C the body of an abstraction to be evaluated and in E the environment in which evaluation must take place while saving the surrounding context on the dump is a measure that ensures substitution of deBruijn indices in exactly the intended binding scope – the abstraction body – by suspensions that belong to just the relevant environment.

Rule (5) takes care of suspensions that contain expressions other than abstractions, i.e., primarily applications but also deBruijn indices. They are set up for evaluation in basically the same way as by rule (4): the expressions are isolated in C together with the corresponding environments in E, and the surrounding contexts are saved on the dump.

And finally, rule (6) reconstructs from the components spread out over C and S irreducible applications in S.

An initial machine state has the entire expression to be reduced set up in the code structure C, with all other structures empty, and the terminal state, so it exists, has its weak normal form set up in the value stack S, while all other structures are empty.

The machine stops in such a state since none of the rules of fig. 2 matches.

It has to be well understood that this machine can reduce only closed λ-expressions. This is due to the fact that deBruijn indices, by definition, cannot occur free anywhere in the expression and that therefore legitimate reducible expressions can only be top-level applications of closed abstractions to closed abstractions, as a consequence of which the resulting weak normal forms can only be abstractions embedded in closures (which syntactically are indistinguishable from suspensions).

3.4 Reducing Step by Step a Simple Λ-Expression

As an illustration of how this normal-order SECD-machine works, lets have a look at the sequence of machine states in fig. 3 that it brings about when reducing the Λ-expression

$$@\ @\ \Lambda\,\#0\ \Lambda\,\#0\ @\ \Lambda\,\#0\ \Lambda\,\#0$$

to its weak normal form $\Lambda\,\#0$ (which is also its full normal form).

The initial stack configuration at the top of fig. 3 has the entire expression set up in the code structure C while all other structures are empty. This expression being an application, rule (1) takes over to enclose the operand expression in a suspensions that is pushed into S, and the apply node is squeezed underneath the operator in C. The operator thus exposed as the next expression that must be taken care of again is an application which calls once more for rule (1), yielding the third stack configuration from the top. The abstraction now on top of C is by rule (2) wrapped up in a closure and pushed as value into S, thus bringing the inner apply node to the top of C, its operand being underneath the operator in S (fourth configuration).

$$\begin{array}{r}
nil \mid S \\
nil \mid E \\
@\,@\,\Lambda\,\#0\,\Lambda\,\#0\;@\,\Lambda\,\#0\,\Lambda\,\#0 : nil \mid C \\
nil \mid D
\end{array}$$

$$\text{Rule 1} \quad \Downarrow$$

$$\begin{array}{r}
[\, nil\; @\,\Lambda\,\#0\,\Lambda\,\#0 \,] : nil \mid S \\
nil \mid E \\
@\,\Lambda\,\#0\,\Lambda\,\#0 : @ : nil \mid C \\
nil \mid D
\end{array}$$

$$\text{Rule 1} \quad \Downarrow$$

$$\begin{array}{r}
[\, nil\; \Lambda\,\#0 \,] : [\, nil\; @\,\Lambda\,\#0\,\Lambda\,\#0 \,] : nil \mid S \\
nil \mid E \\
\Lambda\,\#0 : @ : @ : nil \mid C \\
nil \mid D
\end{array}$$

$$\text{Rule 2} \quad \Downarrow$$

$$\begin{array}{r}
[\, nil\; \Lambda\,\#0 \,] : [\, nil\; \Lambda\,\#0 \,] : [\, nil\; @\,\Lambda\,\#0\,\Lambda\,\#0 \,] : nil \mid S \\
nil \mid E \\
@ : @ : nil \mid C \\
nil \mid D
\end{array}$$

$$\text{Rule 4} \quad \Downarrow$$

$$\begin{array}{r}
[\, nil\; @\,\Lambda\,\#0\,\Lambda\,\#0 \,] : nil \mid S \\
[\, nil\; \Lambda\,\#0 \,] : nil \mid E \\
\#0 : nil \mid C \\
(nil,\; @ : nil,\; nil) \mid D
\end{array}$$

$$\text{Rule 3} \quad \Downarrow$$

$$\begin{array}{r}
[\, nil\; \Lambda\,\#0 \,] : [\, nil\; @\,\Lambda\,\#0\,\Lambda\,\#0 \,] : nil \mid S \\
[\, nil\; \Lambda\,\#0 \,] : nil \mid E \\
nil \mid C \\
(nil,\; @ : nil,\; nil) \mid D
\end{array}$$

$$\text{Rule 7} \quad \Downarrow$$

Fig. 3. Reducing step by step the expression $@\,@\,\Lambda\,\#0\,\Lambda\,\#0 @\,\Lambda\,\#0\,\Lambda\,\#0$ on the SECD-machine

$$\begin{array}{r|l}
[\,\mathit{nil}\ \Lambda\,\#0\,] : [\,\mathit{nil}\ @\,\Lambda\,\#0\,\Lambda\,\#0\,] : \mathit{nil} & S \\
\mathit{nil} & E \\
@ : \mathit{nil} & C \\
\mathit{nil} & D
\end{array}$$

$$\text{Rule 4} \quad \Downarrow$$

$$\begin{array}{r|l}
\mathit{nil} & S \\
[\,\mathit{nil}\ @\,\Lambda\,\#0\,\Lambda\,\#0\,] : \mathit{nil} & E \\
\#0 : \mathit{nil} & C \\
(\mathit{nil},\ \mathit{nil},\ \mathit{nil}) & D
\end{array}$$

$$\text{Rule 3} \quad \Downarrow$$

$$\begin{array}{r|l}
[\,\mathit{nil}\ @\,\Lambda\,\#0\,\Lambda\,\#0\,] : \mathit{nil} & S \\
[\,\mathit{nil}\ @\,\Lambda\,\#0\,\Lambda\,\#0\,] : \mathit{nil} & E \\
\mathit{nil} & C \\
(\mathit{nil},\ \mathit{nil},\ \mathit{nil}) & D
\end{array}$$

$$\text{Rule 7} \quad \Downarrow$$

$$\begin{array}{r|l}
[\,\mathit{nil}\ @\,\Lambda\,\#0\,\Lambda\,\#0\,] : \mathit{nil} & S \\
\mathit{nil} & E \\
\mathit{nil} & C \\
\mathit{nil} & D
\end{array}$$

$$\text{Rule 5} \quad \Downarrow$$

$$\begin{array}{r|l}
\mathit{nil} & S \\
\mathit{nil} & E \\
@\,\Lambda\,\#0\,\Lambda\,\#0 : \mathit{nil} & C \\
(\ \mathit{nil},\ \mathit{nil},\ \mathit{nil}\ ) & D
\end{array}$$

$$\text{Rule 3} \quad \Downarrow$$

... and so on ...

Fig. 3. (*continued*)

At this point rule (4) detects a β-redex. It removes both the operator closure and the operand suspension from S, isolates the body #0 of the abstraction in C, and also prepends the operand suspension to the empty environment *nil* carried along with the closure, which now becomes active. The old environment and the remaining code structure C, i.e., the outermost apply node, are saved on the dump (fifth configuration from the top). Evaluating the deBruijn index #0 in C calls for rule (3), which copies the environment entry at position 0 relative to its top, which is the suspension [$nil\ \Lambda\,\#0$], on top of S, leaving the control structure C empty (last configuration of fig. 3).

Having thus completed the evaluation of the operator expression of the outermost application, the machine returns, by rule (7), to the surrounding context to continue with the evaluation of the outermost application (top configuration of fig. 3). Going basically through the same motions, it arrives, after three more steps, at a configuration that has the operand of the outermost application enclosed in a suspension set up in S, with all other structures empty. As this suspension contains an application, it is intercepted by rule (5) to enforce its evaluation as well. In doing so, the machine creates a new context which has the application @ $\Lambda\,\#0$ $\Lambda\,\#0$ set up in C and the associated empty environment in E, just as before starting the evaluation of the entire expression. After performing the same four steps that reduced the identical operator expression, the machine terminates with the closure [$nil\ \Lambda\,\#0$] in S and all other structures empty.

4 Toward Fully Normalizing λ-Calculus Machines

Upgrading a weakly to a fully normalizing λ-calculus machine requires (the equivalent of) full-fledged β-reductions to preserve the functional property of the λ-calculus when substituting and reducing under abstractions. A clever implementation that can be mechanically executed almost as efficiently as naive substitutions may be obtained by taking advantage of a few more properties of the λ-calculus beyond the β-reduction rule itself that are well covered in standard textbooks [Bar84, HS86]. They are briefly reviewed in the following subsection.

4.1 β-Reduction, η-Extension, β-Distribution and Head (Normal) Forms

In the nameless Λ-calculus that is of interest here, deBruijn indices measure distances, in terms of numbers of intervening Λs, between the syntactical positions of their occurrences and the Λ-abstractors that bind them. Full-fledged β-reduction requires updating them whenever the number of Λs in between changes. More specifically, when removing intervening Λs, the indices must be decremented, and when squeezing additional Λs in between, the indices must be incremented accordingly.

Consider as a small example that may help to illustrate how this works the expression [7]

$$\Lambda_2\ @\ \Lambda_1\Lambda_0\ @\ \#1\ \#2\ \Lambda_4\ \#1 \quad .$$

In the body of the abstraction $\Lambda_1\Lambda_0\ @\ \#1\ \#2$ the indices $\#1$ and $\#2$ are bound to Λ_1 and Λ_2, respectively, the index $\#1$ occurs free in the abstraction $\Lambda_4\ \#1$ but is also bound to Λ_2; there are no indices that are bound to Λ_0 and Λ_4.

[7] The subscripts attached to the Λs merely serve to facilitate explaining which deBruijn index is bound to which abstractor.

This expression evaluates in two β-reduction steps as follows:

$$\Lambda_2 \, @ \, \Lambda_1 \Lambda_0 \, @ \, \#1 \; \#2 \; \Lambda_4 \, \#1 \quad \rightarrow_\beta \; \Lambda_2 \Lambda_0 \, @ \, \Lambda_4 \, \#2 \; \#1 \;\; \rightarrow_\beta \; \Lambda_2 \Lambda_0 \, \#1$$

Reducing, in the first step, the outer application substitutes the abstraction $\Lambda_4 \, \#1$ for the index #1 in the body of the abstraction $\Lambda_1 \Lambda_0 \, @ \, \#1 \; \#2$, thereby removing the abstractor Λ_1 and decrementing the original index #2 as the distance to the binding Λ_2 is now one less. However, the index in the body of $\Lambda_4 \, \#1$ must be incremented since crossing the abstractor Λ_0 increases the distance to the binding Λ_2 by one.

The second step β-reduces the remaining application. As the abstractor Λ_4 does not bind anything, the operand #1 is simply consumed, but the index #2 in the abstraction body is decremented to #1 since the disappearance of the abstractor Λ_4 has shortened by one the distance to the binding Λ_2.

The troublesome part about performing β-reductions in this way is that deBruijn indices may have to be counted up and down several times, as may be illustrated by the following example:

$$@ \, @ \, @ \, \Lambda\Lambda\Lambda \, @ \, \#2 \, @ \, \#1 \, \#0 \;\; \#3 \;\; \#2 \;\; \#1$$

(here it is assumed that the indices #3, #2, #1 in operand positions of the three nested outer applications are bound by Λ-abstractors somewhere in a surrounding expression). Reducing these applications step by step from innermost to outermost yields:

$$@ \, @ \, @ \, \Lambda\Lambda\Lambda \, @ \, \#2 \, @ \, \#1 \, \#0 \;\; \#3 \; \#2 \; \#1 \;\; \rightarrow_\beta$$
$$@ \, @ \, \Lambda\Lambda @ \, \#5 \, @ \, \#1 \, \#0 \;\; \#2 \; \#1 \;\; \rightarrow_\beta \; @ \, \Lambda @ \, \#4 \, @ \, \#3 \, \#0 \;\; \#1 \;\; \rightarrow_\beta \; @ \, \#3 \, @ \, \#2 \, \#1$$

It is interesting to note that the operand indices are in their places of substitution in the abstraction body first stepped up by the number of Λs whose scopes are being penetrated, but that these indices are decremented again as the Λs are being consumed by subsequent β-reductions, with the net effect that they have not changed at all after all β-reductions are done. Needless to say that this is a special property of full applications which has in fact already been exploited in the weakly normalizing machine of the preceding section.

However, this example also tells us that when β-reducing step by step a partial application, free occurrences of deBruijn indices in operand expressions are, after all redices are done, in their places of substitution effectively stepped up by the number of Λs remaining, i.e., by the arity of the resulting abstraction.

More specifically, a partial application of the general form

$$\underbrace{@ \ldots @}_{k} \; \underbrace{\Lambda \ldots \Lambda}_{n} e_b \;\; e_1 \; \ldots \; e_k \mid k < n$$

β-reduces to an $(n-k)$-ary abstraction that has all occurrences of the deBruijn indices $\#(n-1) \ldots \#(n-k)$ in e_b substituted by the operands $e_1 \ldots e_k$, respectively, in which all occurrences of (relatively) free deBruijn indices are incremented by $(n-k)$. In the special case that $k = n$, i.e., we have a full application as above, the original indices remain unchanged. Of course, all free occurrences of deBruijn indices in the original n-ary abstraction must be decremented by k.

This leads us to conclude that if we can find a way of doing these k β-reductions in one conceptual step, a lot of superfluous index updates could be spared.

As a first step toward this end, we make use of η-extensions as an elegant way of minimizing the number of updates on deBruijn indices when reducing partial applications. η-extension derives from the semantic equivalence

$$@\, e_0\, e_1 \;=\; @\, \Lambda @\, e_0^{(+1)}\, \#0 \;\; e_1 \quad ,$$

where the superscript on $e_0^{(+1)}$ denotes the addition of 1 to all free occurrences of deBruijn indices in e_0, since an additional abstractor Λ has been squeezed between them and the binding Λs that may be found in a larger, surrounding expression. This equivalence also implies that

$$e_0 \;=\; \Lambda\, @\, e_0^{(+1)}\, \#0 \;\; .$$

More generally, when η-extending an abstraction k-fold, we get

$$e \;=\; \underbrace{\Lambda \ldots \Lambda}_{k} \underbrace{@ \ldots @}_{k}\; e^{(+k)}\; \#(k-1) \ldots \#0 \quad .$$

This semantic equivalence may be readily employed to turn partial into full applications that can be reduced by a weakly normalizing machine. All that needs to be done is to extend a partial application by as many applications to deBruijn indices in ascending order as there are missing operands, and to put in front of this extended application the same number of Λ-abstractors:

$$\underbrace{@ \ldots @}_{k}\; \underbrace{\Lambda \ldots \Lambda}_{n} e_b\; e_{k-1}\; \ldots e_0 \;\; =$$
$$\underbrace{\Lambda \ldots \Lambda}_{n-k} \underbrace{@ \ldots @}_{n-k} < \underbrace{@ \ldots @}_{k}\; \underbrace{\Lambda \ldots \Lambda}_{n} e_b\; e_{k-1} \ldots e_0 >^{+(n-k)}\;\; \#(n-k-1) \ldots \#0 \quad .$$

(the construct $< \cdots >^{+(n-k)}$ denotes incrementation by $(n-k)$ of all free occurrences of deBruijn indices in the expressions within the brackets.)

The weakly normalizing SECD-machine augmented by an appropriate mechanism for such η-extension-in-the-large can thus be made to reduce, under an $(n-k)$-ary abstraction, a body composed of the application of an n-ary abstraction to n operand expressions of which the outermost $(n-k)$ are deBruijn indices from the interval $\#0 \ldots \#(n-k-1)$. It creates an environment for the evaluation of the abstraction body e_b which substitutes the indices $\#(n-1) \ldots \#(n-k)$ by the expressions $e_{(k-1)} \ldots e_0$ (with updated indices) and the indices $\#(n-k-1) \ldots \#0$ by themselves.

As a second step, we will make use of the fact that β-redices can be distributed over the components of an abstraction body that is itself an application. For the simple case of distributing just one β-redex we have

$$@\, \Lambda\, @\, e_a\, e_b \;\; e_1 \;=\; @\, @\, \Lambda\, e_a\, e_1\; @\, \Lambda\, e_b\, e_1 \quad .$$

This may be generalized for n nested redices as

$$\underbrace{@\ldots@}_{n}\ \underbrace{\Lambda\ldots\Lambda}_{n}@\,e_a\,e_b\,e_1\ldots e_n\ =$$
$$@\ \underbrace{@\ldots@}_{n}\ \underbrace{\Lambda\ldots\Lambda}_{n}e_a\ e_1\ldots e_n\ \underbrace{@\ldots@}_{n}\ \underbrace{\Lambda\ldots\Lambda}_{n}e_b\ e_1\ldots e_n\ \ ,$$

which we may call a β-distribution-in-the-large. By pushing β-redices in this way recursively in front of the subexpressions of an abstraction body, β-reductions may be delayed until and performed only when and where they are actually needed.

As a third step, we combine both η-extensions-in-the-large and β-distributions-in-the-large with a suitable reduction strategy. It may be derived from looking at the syntax of Λ-expressions from a particular perspective that emphasizes what are called head forms:

$$h\mid t\ =_s\ \#i\mid \underbrace{\Lambda\ldots\Lambda}_{n}\underbrace{@\ldots@}_{r}\ h\ t_1\ldots t_r$$

A head form generally is a (nested) application of a single head expression h to some $r \geq 0$ tail expressions $t_1 \ldots t_r$ which is preceded by some $n \geq 0$ abstractors. Heads and tails are recursively constructed in the same way, i.e., they all have head forms as well. Trivial head expressions are deBruijn indices $\#i$. If the outermost head expression h is a deBruijn index, then we have a head-normal form.

Occurrences of deBruijn indices in Λ-expressions must always be smaller than the total number of Λs preceding them, i.e., there is no notion of such indices being free in the entire head form. However, we may consider indices as being free if they are bound to the outermost leading sequence of Λs because then they may be passed around and updated by β-reductions but they never get substituted by anything else.

Following a normal-order regime, the reduction strategy that lends itself directly to head forms is called head-order reduction as it emphasizes reductions in the head: It first reduces the head expression to head-normal form and then recursively all remaining tails to head normal forms as well, thus eventually arriving at a full normal form of the entire expression, provided the whole process terminates after finitely many β-reductions. The significance of this heads-first strategy derives from the fact that an expression cannot have a full normal form without having a head normal form, which should therefore be determined before evaluating the tails.

4.2 Head-Order Reduction

In this subsection we are going to illustrate, by means of the graphical representation of a typical head form as in fig. 4, how head-order reductions can be organized, closely following an earlier proposal by Berkling [Ber86] [8].

[8] The contents of this subsection are in large parts adopted from the author's monograph on Abstract Computing Machines[Kge05].

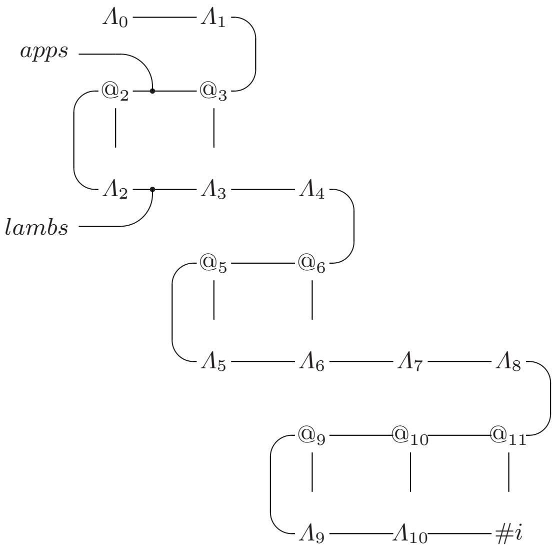


Fig. 4. A typical head form of a λ-expression

Head and tail expressions obviously are the operators and operands (depicted by the downward pointing thin lines), respectively, of applications. On the path from the root node of this graph down to the head index $\#i$ we find alternately only sequences of Λ-abstractors and sequences of applicators @, to which we will refer as *lambs* and *apps* sequences, respectively, and to the entire path as the (leftmost) spine of the head form. All tails along this spine have recursively head forms, or are spines, of their own.[9]

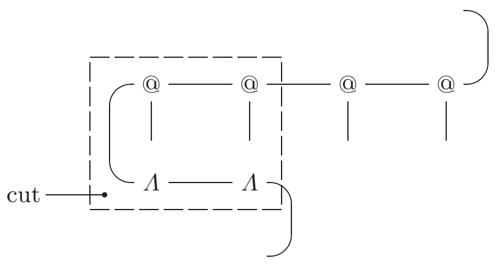
A section of the spine headed by a *lambs* sequence of length n is in fact a curried n-ary abstraction whose body stretches over the entire remaining spine, i.e., the spine of fig. 4 includes four abstractions nested inside each other.

Normal order reduction as effected by the applicator @ demands that β-redices be reduced systematically from top to bottom along such spines until no more β-redices are left, i.e., the spine features a sequence of leading Λs followed by a sequence of applicators (which may be empty) followed by a head index bound by one of the leading Λs, in which case we have arrived at a head-normal form.

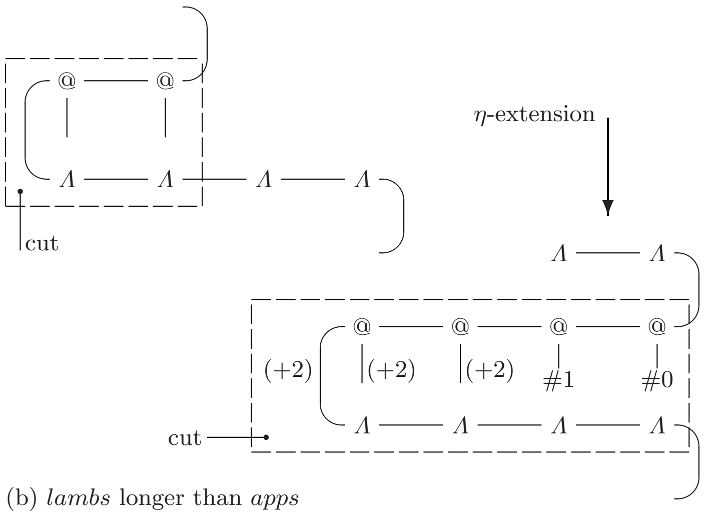
Looking at the meander-like structure of the spine in fig. 4, β-redices can be easily identified in the left-hand corners that connect *apps* and *lambs* sequences and thus pair innermost apply nodes with outermost abstractors. However, rather than actually performing these β-reductions step by step from left to right, the idea of head-order reduction is to take largest possible chunks of β-redices, which we'll call cuts, out of such corners and to distribute them over the head and tail expressions of the *apps* sequence that follows next along the spine, using β-distributions-in-the-large as outlined in the preceding subsection.

[9] Λ-nodes and apply nodes are in this graph enumerated so that one can follow up more easily on what is ending up where when reducing this spine.

Just what these cuts are depends on the relative lengths of the *apps* and *lambs* sequences involved, as depicted in fig. 5 below.



(a) *apps* longer than *lambs*



(b) *lambs* longer than *apps*

Fig. 5. Taking cuts off left-hand corners

The upper part (a) shows the easier case with an *apps* sequence that has at least the same length as the *lambs* sequence. Here we have a full application as the cut matches each abstractor with an apply node.

The lower part (b) shows a corner in which the *lambs* sequence is longer than the *apps* sequence, i.e., we have a partial application that β-reduces to a new abstraction of lesser arity, which would be 2 in the particular case. This can be accomplished by means of an η-extension-in-the-large, as also introduced in the preceding subsection, that transforms the entire *apps* − *lambs* corner into a full application. The added apply nodes have the deBruijn indices #0 and #1 in their tails, and all free occurrences of deBruijn indices in the head and the tails of the original *apps* sequence are stepped up by 2, as annotated at the respective edges, to account for the two Λ-nodes introduced by the η-extension.

Inspecting the spine of fig. 4, we note that this head form includes three *apps* − *lambs* corners, of which the upper two are partial applications that must be η-extended before β-distributing them over the branches of the spine. Fig. 6 below illustrates how this is done.

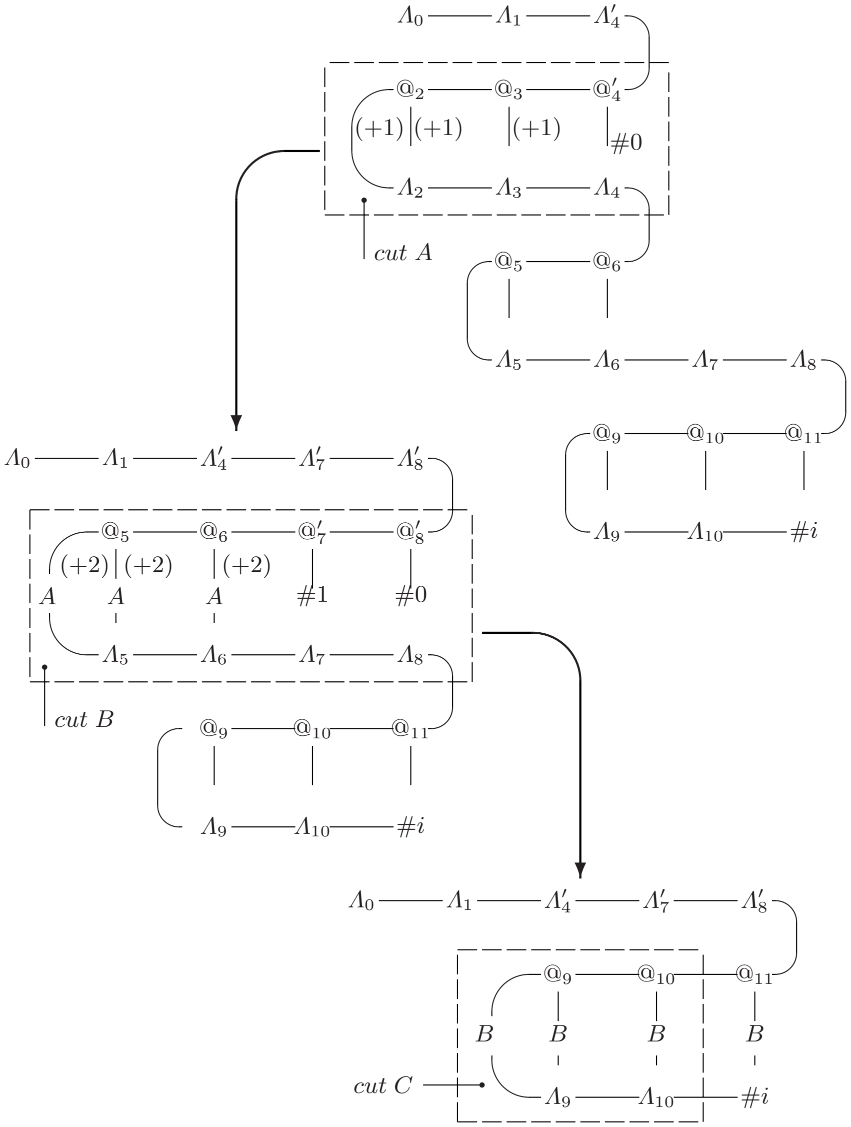


Fig. 6. β-distributing cuts over the branches of the spine

Proceeding from top to bottom along the spine, the first corner that is being encountered must be η-extended by one $\Lambda \,|\, @$ pair to obtain a cut A that represents a full application (the graph in the upper part of the figure). Distributing this cut over the next corner of the spine squeezes it in front of the tails and the head of its *apps*-sequence. This corner must be η-extended by two $\Lambda\,|\,@$ pairs to form another cut B for a full application (the graph in the middle of the figure), which in turn is β-distributed over the head and the tails of the remaining corner of the spine (the graph at the bottom). This corner constitutes a full application as it is, forming a cut C. This cut is trivially distributed just in front of the head index $\#i$, i.e., it remains in place.

If in cut C we now expand the copy of cut B that makes up its left-hand corner and, likewise, in cut B expand the copy of cut A on the left, we obtain the spine shown in fig. 7 below.

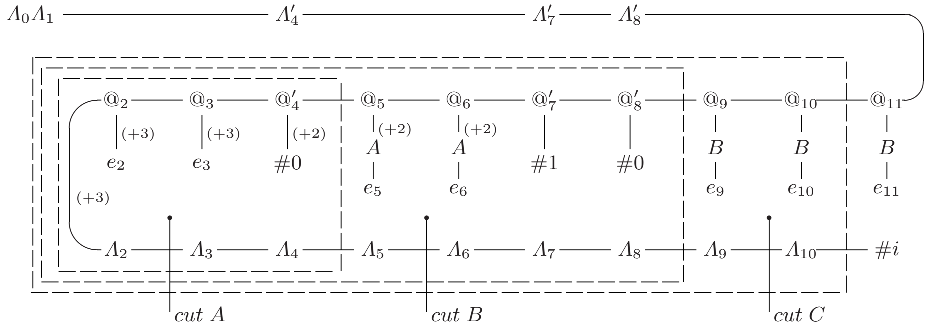


Fig. 7. The spine emerging from the one of fig. 4 after having completed all η-extensions and β-distributions-in-the-large

This spine features a leading *lambs*-sequence to which have been lifted the Λ-abstractors that have been introduced by η-extensions. It is followed by a single left-hand corner that connects an *apps* sequence of length 10 with a *lambs* sequence of length 9, i.e., we have in fact unfolded, by means of repeated η-extensions and β-distributions (...-in-the-large) what was the original cut C to nine β-redices. This new cut C includes cut B which, in turn, includes cut A [10].

Having thus straightened the original spine, we can finally contract, in one conceptual step to which we may refer as β-reduction-in-the-large, all β-redices of the original spine that have now accumulated in a single cut C, thereby completely consuming it. The resulting reductum depends on the deBruijn index $\#i$ in the head of the spine, which happens to be the entire body of the abstraction formed by the *lambs*-sequence preceding it.

If this index is smaller than 9, it is bound to a Λ within the preceding *lambs* sequence, which means that the abstraction is in fact a selector function that picks from the *apps* sequence the tail of the apply node that in the graph is

[10] Note that the apply and Λ nodes that have been introduced by η-extensions are annotated as primed and receive the same indices as the corresponding Λs in the original *lambs* sequences.

opposite to the Λ to which the index is bound. For instance, an index $i = 1$ that is bound to Λ_9 selects the tail of $@_9$, or an index $i = 4$ is bound to Λ_6 and thus returns the tail of $@_6$.

These tails are substituted in the head position of the spine that is left over after the cut C has disappeared, which is just the leading *lambs* sequence $\Lambda_0 \Lambda_1 \Lambda'_4 \Lambda'_7 \Lambda'_8$ followed by the apply node $@_{11}$ whose tail remains intact.

This process of η-extensions, β-distributions and β-reductions (-in-the-large) repeats itself in the head thus expanded until the head position is occupied by an index bound to one of the Λs of the leading *lambs*-sequence, i.e., the spine has become head-normalized.

This is the case if in the original spine of fig. 4 the index was bound either to one of the leading Λs, say $i = 10$, or to one of the unapplied Λs that gave rise to η-extensions, say $i = 6$.

In the former case, the head index is bound to Λ_0 and must remain so after cut C in fig. 7 has been completely β-reduced, i.e., the resulting index should be $i = 4$. We can easily convince ourselves that this is indeed so: there are nine intervening Λs that do disappear due to these β-reductions, decrementing the head index to $i = 1$, but three Λs have been squeezed in between due to η-extensions, resulting in the index $i = 4$.

In the latter case, the original index $i = 6$ is bound to Λ_4, which selects the index $i = 2$ (i.e., $i = 0$ incremented by 2) as the tail of $@'_4$, which in turn is bound to Λ'_4 in what has become the expanded leading *lambs* sequence.

The cuts that build up along the spine in fact define an environment, just as we know it from the SECD-machine, in which the head expression is to be evaluated. This environment just keeps expanding as long as there are *apps–lambs* corners left to be distributed down the spine. With one large *apps–lambs* corner remaining that has accumulated, in nested form, all the others that were preceding it, we have a single contiguous environment. Depending on its value, the head index defines either a single access into this environment to retrieve a tail expression that must be substituted in the head, generally leading to more β-reductions along the spine, or it is bound by one of the Λs of the resulting leading *lambs* sequence, in which case we are done with the head, having arrived at a head-normal form, and may turn to the tails, if there are any left, and recursively reduce them in head-order as well.

The tails of head normal forms are generally unevaluated expressions preceded by cuts, or by their environments, that are equivalent to the suspensions as we know them from the SECD-machine.

5 The FN_SECD-Machine

The runtime structures and the basic mechanisms of the weakly normalizing SECD-machine, not very surprisingly, can be employed in a <u>f</u>ully <u>n</u>ormalizing machine as well. We definitely need a code structure C, an environment that holds suspensions $[\ E\ e\ ]$, some stack S that temporarily holds intermediate values, basically again suspensions but also deBruijn indices that are bound by leading

Λs. Stack S also serves as the destination of full normal forms. Beyond that, it is expedient to include a dump as well that keeps track of nested β-distributions and η-extensions, accommodating the respective return continuations.

The machine must also include an efficient η-extension mechanism that does the equivalent of generating as arguments for unapplied Λs deBruijn indices and of updating those introduced by earlier η-extensions, as outlined in subsection 4.2.

5.1 The Unapplied Lambdas Count

The basic idea of how η-extensions and the ensuing updates on deBruijn indices can be done almost effortlessly may be inferred from a close look at the spine of fig. 7.

We note that after the first η-extension that leads to cut A the deBruijn index in the tail of the apply node $@'_4$ receives the value #0. When doing the second η-extension that brings about cut B, the tails of the new apply nodes $@'_7$ and $@'_8$ receive the indices #1 and #0, respectively, and the index in the tail of $@'_4$ is stepped up by 2, which equals the number of Λs that have been squeezed in between.

Rather than updating in this way earlier deBruijn indices whenever another η-extension must be done along the spine, the very same index values may be obtained by the following method that is decidedly simpler to implement and more efficient to execute [Trou93]:

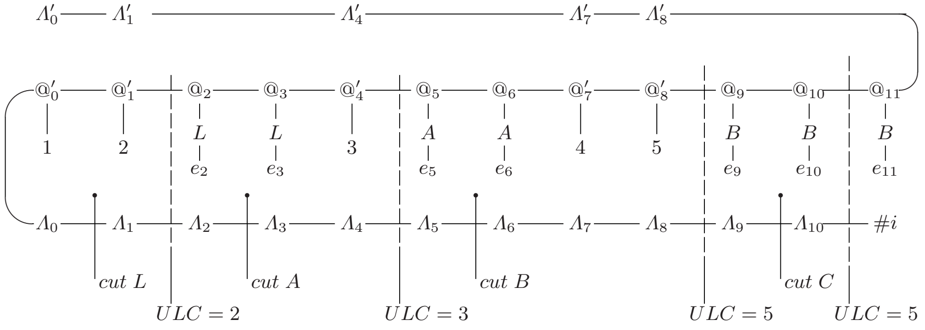
- The number of unapplied Λs introduced by η-extensions while proceeding from top to bottom along the original spine is kept track of in a count variable ULC (which stands for <u>U</u>napplied <u>L</u>ambdas <u>C</u>ount), beginning with the value 0 (though any other non-negative integer value could be chosen as well);
- The tails of the apply nodes introduced by η-extensions are filled with ULC values rather than deBruijn indices in monotonically ascending order;
- When needed, the correct deBruijn indices may be obtained by subtracting from the current value of the ULC counter the ULC values actually found in the η-extended tails (which may be the same or lower).

The interesting properties about this method are that the ULCs put into the η-extended tails are invariant against further η-extensions down the spine, that these values can be generated by a simple counting mechanism, and that correct index values can be calculated by a single integer subtraction, thus minimizing the effort of manipulating them.

However, in order to treat all deBruijn indices, including those that are bound by what originally were the leading Λs of the spine, in a uniform way, these unapplied abstractors must be η-extended as well.

These extensions add another (innermost) cut L to the spine of fig. 7, yielding the spine depicted in fig. 8. It has the tails of cut L filled with the ULC values 1 and 2, followed by the value 3 in the η-extended cut A and by the values 4 and

5 in the η-extended cut B. The ULC values after completion of the cuts L, A, B and C are also shown at the bottom.



Λ'_0 Λ'_1 Λ'_4 Λ'_7 Λ'_8
$@'_0$ $@'_1$ $@_2$ $@_3$ $@'_4$ $@_5$ $@_6$ $@'_7$ $@'_8$ $@_9$ $@_{10}$ $@_{11}$
1 2 L L 3 A A 4 5 B B B
e_2 e_3 e_5 e_6 e_9 e_{10} e_{11}
Λ_0 Λ_1 Λ_2 Λ_3 Λ_4 Λ_5 Λ_6 Λ_7 Λ_8 Λ_9 Λ_{10} $\#i$
cut L
cut A
cut B
cut C
$ULC = 2$
$ULC = 3$
$ULC = 5$
$ULC = 5$

Fig. 8. The spine of fig. 7, η-extended by another cut L for the leading Λs, and showing ULC values replacing all η-extended deBruijn indices

To exemplify calculation from ULCs of correct deBruijn indices, consider environment accesses with the head indices #3, #6 and #10, all of which are bound by unapplied Λs. Index #3 picks the tail of the η-extended apply node $@'_7$, i.e., the ULC value 4. Correcting it with the ULC value 5 reached after having flattened the entire spine yields the deBruijn index #1; likewise the head index #6 selects the value 3 from the tail of $@'_4$ and, after subtracting it from the ULC-value 5, returns the deBruijn index #2. In both cases we obtain exactly the same deBruijn indices as would be selected from the spine of fig. 7. And finally, index #10 which was bound to Λ_0 in the original spine selects 1 from the tail of $@'_0$. Upon subtracting it from the ULC value 5 we get the deBruijn index #4 which remains bound by Λ'_0 in the emerging leading *lambs* sequence $\Lambda'_0\Lambda'_1\Lambda'_4\Lambda'_7\Lambda'_8$ [11].

5.2 The State Transition Rules

The state description of the FN_SECD-machine differs from that of the ordinary SECD-machine only in the addition of the unapplied lambdas count ULC as a plain variable u, i.e., the state transition rules specify mappings of the form:

$$\tau_{fn\_secd} : (S,\ E,\ C,\ D,\ u)\ \rightarrow\ (S',\ E',\ C',\ D',\ u').$$

The full set of these rules is given in fig. 9, again in the order in which they need to be matched against machine states. To facilitate comparison with the state transition rules of the weakly normalizing counterpart as given in fig. 2, the same enumeration of rules has been chosen. The rules that complement existing rules

[11] Note that the entire *apps* $-$ $-$*lambs* corner in between disappears due to the β-reduction-in-the-large that effects the selection.

Returning from β-reductions with closures on S
(7b) $([\,E_b\;\Lambda\,e_b\,] : S,\; E,\; nil,\; (E',\; C',\; D',\; u'),\; u) \;\rightarrow ([\,E_b\;\Lambda\,e_b\,] : S,\; E',\; C',\; D',\; u')$

Rearranging applications on C
(1) $(S,\; E,\; @\,e_f\;e_a : C,\; D,\; u) \;\rightarrow ([\,E\;e_a\,] : S,\; E,\; e_f : @ : C,\; D,\; u)$

Creating closures on S for abstractions on C
(2) $(S,\; E,\; \Lambda\,e_b : C,\; D,\; u) \;\rightarrow ([\,E\;\Lambda\,e_b\,] : S,\; E,\; C,\; D,\; u)$

Substituting deBruijn indices
(3) $(S,\; E,\; \#i : C,\; D,\; u) \;\rightarrow (lookup(\#i,\; u,\; E) : S,\; E,\; C,\; D,\; u)$

Entering the evaluation of β-redices
(4a) $([\,E'\;\Lambda\,e_b\,] : e_a : S,\; E,\; @ : C,\; D,\; u) \;\rightarrow (S,\; e_a : E',\; e_b : nil,\; (E,\; C,\; D,\; u),\; u)$

Dealing with unapplied closures on S
(4b) $([\,E'\;\Lambda\,e_b\,] : S,\; E,\; C,\; D,\; u) \;\rightarrow (S,\; (u+1) : E',\; e_b : \Lambda : nil,\; (E,\; C,\; D,\; u),\; (u+1))$

Entering the normalization of suspensions on S
(5) $([\,E'\;e'\,] : S,\; E,\; C,\; D,\; u) \;\rightarrow (S,\; E',\; e' : nil,\; (E,\; C,\; D,\; u),\; u)$

Putting leading Λs in front of an expression in S
(4c) $(e_b : S,\; E,\; \Lambda : nil,\; (E',\; C',\; D',\; u'),\; u) \;\rightarrow (\Lambda\,e_b : S,\; E',\; C',\; D',\; u')$

Dealing with abstractions on S and apply nodes on C
(8) $(\Lambda\,e_b : S,\; E,\; @ : C,\; D,\; u) \;\rightarrow (S,\; E,\; \Lambda\,e_b : @ : C,\; D,\; u)$

Rearranging applications for the evaluation of tail suspensions
(9) $(e_b : [\,E'\;e_a\,] : S,\; E,\; @ : C,\; D,\; u) \;\rightarrow ([\,E'\;e_a\,] : e_b : S,\; E,\; @^* : C,\; D,\; u)$

Reconstructing applications after normalization of their tail suspensions
(10) $(e_a : e_b : S,\; E,\; @^* : C,\; D,\; u) \;\rightarrow (@\,e_b\,e_a : S,\; E,\; C,\; D\;u)$

Reconstructing irreducible applications in S
(6) $(\,e_b : e_a : S,\; E,\; @ : C,\; D,\; u) \;\rightarrow (@\;e_b\;e_a : S,\; E,\; C,\; D,\; u)$

Returning from β-reductions and η-extensions
(7a) $(S,\; E,\; nil,\; (E',\; C',\; D',\; u'\,)\;u\,) \rightarrow (S,\; E',\; C',\; D',\; u'\,)$

Fig. 9. The state transition rules of a fully normalizing FN_SECD machine

have their numbers tagged by letters $b,\; c$ (with a tagging the original rules), and three entirely new rules receive the numbers $8,\; 9$ and 10.

Rules (1) to (4a), other than for an additional variable u that holds the current ULC value, are exactly the same as those of the weakly normalizing machine. The function *lookup* used in rule (3) is per pattern matching recursively defined as:

$$\begin{aligned} lookup\,(\#0,\; u,\; [\,E'\;e'\,] : E) &\;\rightarrow\; [\,E'\;e'\,] \\ (\#0,\; u,\; un : E) &\;\rightarrow\; \#(u - un) \\ (\#i,\;\; u,\; v : E) &\;\rightarrow\; lookup\,(\#(i-1),\; u,\; E) \end{aligned}$$

i.e., it returns as the i-th environment entry either a suspension or, if this entry contains a ULC value un, the corresponding deBruijn index.

Rule (4b) η-extends the unapplied abstraction contained in a closure that sits on top of stack S. It does so by prepending the current ULC, incremented by one, to the closure's environment that now becomes active, and by setting the isolated abstraction body up in C for evaluation. To complete the η-extension, the Λ is squeezed underneath the abstraction body, from where it may be retrieved once the body is completely evaluated. Also, the machine saves on the dump a return continuation that includes the old ULC, and it continues with the updated ULC in what now has become the current context. Rule (4c) intercepts the complementary stack configuration that has the evaluated abstraction body on top of S and a Λ as the sole entry on top of C. From these components it constructs a head-normalized abstraction on S. The return continuation retrieved from the dump also includes the old ULC value, which happens to be the current value decremented by one.

There are two rules that are complementary to those that save current machine states (or contexts) on the dump. Rule (7a) covers the general case of returning to a calling context whenever the code structure becomes empty, i.e., an instantiated abstraction body has been evaluated and in this form been completely moved from C to S. This rule must be called after all the other rules have failed to match. However, there is also the special case of an empty code structure in conjunction with a closure on top of S. Such configurations may come about when retrieving, by means of the function *lookup*, tail suspensions that happen to contain abstractions (and thus are in fact closures) from the environment. They must be caught before trying any of the other rules that expect closures on top of S, specifically rule (4b); hence rule (7b) as the first of the list.

Of the new rules, rule (8) takes care of the special case that an abstraction may end up as value on top of stack S together with an apply node @ in C, relative to which it is in operator position. This rule simply moves the abstraction back to C so that rule (2) may, in preparation for an application of rule (4a), wrap it up in a closure that is returned to S.

The remaining new rules (9) and (10) are to force and return from (head-) normalizing tail suspensions left over in a head-normalized spine. To figure out what must be done here, we need to understand that a machine that is just head-normalizing would produce in the value stack S an expression of the general form

$$\underbrace{\Lambda \ldots \Lambda}_{n} \; \underbrace{@ \ldots @}_{r} \; \#i \; [\, E_1 \; t_1 \,] \ldots [\, E_r \; t_r \,] \quad ,$$

i.e., from top to bottom we have a leading *lambs* sequence followed by an *apps* sequence followed by a deBruijn index bound by one of the leading Λs followed by a sequence of tail expressions wrapped up in suspensions. But before this terminal state is reached, we have a configuration with the sequence

$$\#i \; [\, E_1 \; t_1 \,] \ldots [\, E_n \; t_r \,] \qquad \text{in } S$$

and with the sequence

$$\underbrace{@\ldots @}_{r}\ \underbrace{\Lambda\ldots\Lambda}_{n}\ ,$$

of which the first @ is on top of C, and the remaining @s and Λs are, as parts of recursively nested contexts, stacked up in the dump D.

From this configuration forward, without rules (9) and (10) all apply nodes would be moved from C to S, using r times rules (6) and (7a), and then the Λs would follow, using n times rules (4c) and again (7a), which in fact means that the head-normalized spine would be assembled in S from the bottom (the head symbol $\#i$) up to the topmost Λ, without doing anything to the tail suspensions.

To evaluate, on the way up, the tails as well, the machine must intercept stack configurations with an apply node on top of C, an expression value other than a suspension (closure) on top of S and a suspension underneath, and to force the evaluation of this suspension. The first such configuration encountered has the head index $\#i$ of the head-normalized spine on top of S; all other configurations have irreducible application on top of S.

Rule (9), upon encountering such configuration, switches the first and the second expression, thus bringing the tail suspension to the top of S, which in turn enables rule (5) to effect its evaluation. At the same time, the apply node on top of C is marked with the superscript * to keep note of the fact that operator and operand have been interchanged. Upon returning the value (normal form) of the suspension to the top of S, rule (10) simply takes the two expressions on S and the apply node on C to construct a syntactically complete (irreducible) application on top of S, with operator and operand in the right order again.

5.3 Head-Normalizing a Λ-Expression: An Example

To illustrate how the FN_SECD-machine goes about doing its job, fig. 10 shows a sequence of representative configurations that it steps through when reducing the Λ-expression

$$\Lambda\Lambda\,@\,@\,\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2\ \#1\ \#0$$

just to head-normal form. All configurations shown (except the last one) are those at which the machine arrives after having processed either successive η-extensions of unapplied abstractions or successive β-redices[12].

The first configuration shown at the top of the figure depicts the situation after having η-extended the leading two Λs. The ULC indices 2 and 1 are stacked up in the environment and the remaining expression is still in C, with one of the abstractors underneath, while the other one is saved on the dump as part of the outer of two nested return continuations. Next follows the configuration after having rearranged the two nested applications on top of C. It has suspensions for their arguments $\#0$ and $\#1$ stacked up in S, and the applicators squeezed underneath the remaining abstraction in C. The third configuration depicts

[12] To accommodate the relevant steps of this sequence on a single page, the initial configuration which has the ULC value initialized with 0, the entire expression set up in the code structure C and all other structures empty is omitted.

Rules 2 and 4b twice $\Downarrow$

$$\begin{array}{r|l}
nil & S \\
2:1:nil & E \\
@\,@\,\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2\ \#1\ \#0 : \Lambda : nil & C \\
(1:nil,\ \Lambda:nil,\ (nil,nil,nil,0),\ 1) & D \\
2 & u
\end{array}$$

Rule 1 twice $\Downarrow$

$$\begin{array}{r|l}
[\,2:1:nil\ \ \#1\,]:[\,2:1:nil\ \ \#0\,]:nil & S \\
2:1:nil & E \\
\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2 : @ : @ : \Lambda : nil & C \\
(1:nil,\ \Lambda:nil,\ (nil,nil,nil,0),\ 1) & D \\
2 & u
\end{array}$$

Rules 2 and 4a twice $\Downarrow$

$$\begin{array}{r|l}
nil & S \\
[\,2:1:nil\ \ \#0\,]:[\,2:1:nil\ \ \#1\,]:2:1:nil & E \\
\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2 : nil & C \\
(2:1:nil,\ \Lambda:nil,(1:nil,\ \Lambda:nil,\ (nil,nil,nil,0),\ 1),\ 2) & D \\
2 & u
\end{array}$$

Rules 2 and 4b once $\Downarrow$

$$\begin{array}{r|l}
nil & S \\
3:[\,2:1:nil\ \ \#0\,]:[\,2:1:nil\ \ \#1\,]:\#2:\#1:nil & E \\
@\,@\,\Lambda\,\#4\ \#3\ \#2 : \Lambda : nil & C \\
(2:1:nil,\ \Lambda:nil,(1:nil,\ \Lambda:nil,\ (nil,nil,nil,0),\ 1),\ 2) & D \\
3 & u
\end{array}$$

Rule 1 twice $\Downarrow$

$$\begin{array}{r|l}
[\,E'\ \ \#3\,]:[\,E'\ \ \#2\,]:nil & S \\
E' \ = \ 3:[\,2:1:nil\ \ \#0\,]:[\,2:1:nil\ \ \#1\,]:2:1:nil & E \\
\Lambda\,\#4 : @ : @ : \Lambda : nil & C \\
(2:1:nil,\ \Lambda:nil,(1:nil,\ \Lambda:nil,\ (nil,nil,nil,0),\ 1),\ 2) & D \\
3 & u
\end{array}$$

Rules 2 and 4a once $\Downarrow$

$$\begin{array}{r|l}
[\,E'\ \ \#2\,]:nil & S \\
[\,E'\ \ \#3\,]:3:[\,2:1:nil\ \ \#0\,]:[\,2:1:nil\ \ \#1\,]:2:1:nil & E \\
\#4 : nil & C \\
(E',\ @:\Lambda:nil,(2:1:nil,\ \Lambda:nil,(1:nil,\ \Lambda:nil,\ (nil,nil,nil,0),\ 1),\ 2),\ 3) & D \\
3 & u
\end{array}$$

Rule 3 once $\Downarrow$

$$\begin{array}{r|l}
\#1:[\,E'\ \ \#2\,]:nil & S \\
[\,E'\ \ \#3\,]:3:[\,2:1:nil\ \ \#0\,]:[\,2:1:nil\ \ \#1\,]:2:1:nil & E \\
nil & C \\
(E',\ @:\Lambda:nil,(2:1:nil,\ \Lambda:nil,(1:nil,\ \Lambda:nil,\ (nil,nil,nil,0),\ 1),\ 2),\ 3) & D \\
3 & u
\end{array}$$

Fig. 10. Snapshots of typical FN_SECD-machine configurations while head-normalizing the Λ-expression $\Lambda\Lambda\,@\,@\,\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2\ \#1\ \#0$. Note that all deBruijn indices are preceded by #, all ULCs are given as plain integers.

the situation after having completed the equivalent of two β-distributions: the argument suspensions are removed from S and prepended to the environment, while the applicators and abstractors involved are being consumed [13]. The following three configurations show the same steps being performed on the abstraction left in C, which return as the sole entry in C the index #4.

This configuration is conceptually equivalent to a 'straightened' spine consisting of a single $apps - lambs$-corner similar to the one in fig. 8. Here we have the original expression completely transformed into an environment whose entries are the tails of the apply nodes in the *apps*-sequence of such corner. Nothing except the head index of this expression is left in the code structure C.

Accessing the environment with this index, which corresponds to β-reducing in one conceptual step (or in-the-large) an equivalent $apps - lambs$-corner, picks the entry 2 which, after correction with the ULC value 3, is pushed into S as deBruijn index #1 (the last configuration).

At this point, the computation has in fact arrived at a head-normal form which, unfortunately, is not immediately obvious. Except for the updated head index in S and a tail suspension underneath, the constructor nodes of the head-normalized spine are recursively hidden in the dump. It contains four nestings of contexts that are being saved along the way, which include, from outermost to innermost, the code fragments $@ : \Lambda : nil$, $\Lambda : nil$, $\Lambda : nil$, nil. When appended to each other, they yield the trace $@ : \Lambda : \Lambda : \Lambda$. Prepending this trace in reverse order to the two entries in S would yield $\Lambda : \Lambda : \Lambda : @ : \#1 : [\ E'\ \#2\ ]$. Except for the separating symbols, this sequence equals the head-normal form of the initial expression.

However, since the machine doesn't stop there but computes full normal forms, it continues to first unsave, by means of rule (7a), the outermost context on D, thus restoring in C the code sequence $@ : \Lambda : nil$. This in turn enables rule (9) to force the evaluation of the tail suspension $[\ E'\ \#2\ ]$, which after several more steps returns the index value #2, and subsequently, after having recursively restored all the other contexts stacked up in the dump, the fully normalized expression $\Lambda\Lambda\Lambda\, @\, \#1\, \#2$.

6 The FN_SE(M)CD-Machine

In this section we introduce a more sophisticated version of the FN_SECD-machine that does away with some of the complications inherited from the original SECD-machine. Prime candidates for improvement are the state transition rules (2) and (4a/b) of the FN_SECD-machine (see fig. 9) which, irrespective of the contexts in which they are applicable, transform abstractions on top of C into closures on top of S, with the consequence that they have to be unwrapped again before reducing applications or η-extending unapplied abstractions, which almost always happens immediately afterwards. Moreover, as a closure on top of S may also

[13] Note that the environment that has built up in this configuration is in all subsequent steps abbreviated as E' to contain the representation of the dump structure in a single line.

originate from an environment access (rule (3)), it is imperative that the general approach be taken to create new contexts (and to save the current contexts in the dump) to ensure that the computation continues in the environment included in the closure. As an unpleasant side effect, head-normalizing only moderately long spines may generate deeply nested dump structures, as exemplified by the state transition sequence of fig. 10, since every single η-extension or β-distribution along a spine pushes another context (or return continuation). These contexts include, in nested form, successively growing environments and, as parts of the codes saved, the apply nodes and abstractors from which normalized spines must be (re)constructed. This is to say that the machine is predominantly busy saving on (and unsaving from) the dump increasingly complex structures that are pretty hard to analyze, e.g., when the machine is used in a step-by-step mode to follow up on some sequence of state transitions, say, for validation purposes.

The FN_SE(M)CD-machine avoids these problems by two fairly simple measures. It evaluates β-redices and η-extends unapplied abstractions directly, i.e., without going through the superfluous motions of creating closures, and thus avoids the excessive use of the dump. In fact, entire spines can be head-normalized in the same contexts, leaving the dump unchanged. New contexts need be created, and current contexts be saved, only when entering the evaluation of suspensions that are being retrieved from the environment, which happens either whenever a suspension is substituted into the head of a spine or whenever the tails of a head-normalized spine need to be normalized.

Moreover, the environments saved on the dump can be replaced by something much simpler. From the conceptual outline of head-order reduction in subsection 4.2 we recall that the $apps - lambs$-corner that builds up when η-extending and β-distributing cuts along a spine (compare figs. 7 and 8) is being consumed when β-reducing it in-the-large. This translates into the environment becoming irrelevant once it has been accessed by a head index which substitutes an environment entry in its place. If this entry happens to be a suspension, it creates a new context in which it is being evaluated, routinely saving a return continuation on the dump. This immediately raises the question of what the environment to be saved must look like now that the one that is part of the calling context has become obsolote. Another problem relates to the questions of what needs to be done about deBruijn indices that belong to the (head-) normalized expression returned after evaluating the suspension, and which role is being played by the ULCs in this new setting.

To find answers to these questions, we simply need to have a closer look at the spine of fig. 8. Once β-reduction-in-the-large has eaten up the entire $apps - lambs$-corner, there is basically only a leading sequence of η-extended Λs left of the original spine. As a head index bound by one of these leading Λs must, upon returning to the calling context, find an environment entry for it, all that needs to be done conceptually is to η-extend this leading $lambs$-sequence once more, which generates an $apps$-sequence of equal length that has in its tails ULCs in ascending order. And this is exactly the environment that must be included in a return continuation.

So, the solution to our problems consists in replacing in our machine state description the plain ULC variable u by a stack U which, beginning with the initial value 0, stacks up ULCs in their order of creation. This stack is made part of the context stored in the dump. Whenever a context is retrieved from the dump, the contents of this stack become the new (initial) environment. The current ULC value that is required to compute correct deBruijn indices is the topmost entry of U.

The machine also employs a special shunting yard mechanism that uses a separate trace stack M to temporarily store the sequence of abstractors and applicators encountered while traversing a spine from the root node down to the current position of activity. In any state of program execution, it contains the Λs that belong to the leading *lambs*-sequence that has built up at that point, and on top of it apply nodes @ that may or may not be consumed by further β-reductions.

The FN_SE(M)CD-machine derives from the weakly normalizing SE(M)CD-machine described in [Kge05] whose specification, unfortunately, is erroneous, but under `www.informatik.uni-kiel.de/inf/Kluge/index-de.html` a corrected version may be found. Relative to the FN_SECD-machine, its specification requires a few more state transition rules as more stack configurations need to be distinguished, specifically with regard to the topmost entries on the trace stack M. The rules are generally simpler and more direct, operating just locally on the stack tops. Except for environment accesses, there is no need to digg deeper than two entries into a stack.

6.1 Traversing the Spine

Traversing the spine of an expression in the FN_SE(M)CD-machine involves just the code structure C, the value stack S and the trace stack M. They are operated like a shunting yard to traverse constructor expressions in pre-order. The expressions are initially set up in pre-order linearized form in C and from there moved to S. To preserve pre-order linearization in S, the constructor symbols Λ and @ are temporarily sidelined in M while their subexpressions, following recursively the same mechanism, are moved from C to S, where they end up with their left and right subexpressions interchanged [Ber75].

To describe how this traversal mechanism works, we consider a very basic machine only whose state is given by a triple $(S,\ M,\ C)$. The expressions to be traversed are assumed to have the general form $\kappa\, e_1\, e_2\, \ldots\, e_n$, where κ is an n-ary constructor and $e_1, \ldots, e_n$ are subexpressions; they are in C set up for traversal as sequences $\kappa : e_1 : e_2 : \ldots : e_n : C$.

The state transition rules of this machine are given below, listed in the order in which they must be matched against machine states:

$$(S,\ \kappa^{(0)} : nil,\ C) \to (\kappa : S,\ nil,\ C)$$

$$(S,\ \kappa_1^{(0)} : \kappa_2^{(i)} : M,\ C) \mid i > 0 \to (\kappa_1 : S,\ \kappa_2^{(i-1)} : M,\ C)$$

$$(S,\ \kappa^{(i)} : M,\ e_{at} : C) \mid i > 0 \rightarrow (e_{at} : S,\ \kappa^{(i-1)} : M,\ C)$$

$$(S,\ M,\ \kappa : C) \rightarrow (S,\ \kappa^{(n)} : M,\ C)$$

The last rule moves a constructor symbol that appears on top of C into the trace stack M and attaches to it an index which initially receives as value the arity n. This index denotes the number of subexpressions hooked up to the constructor that are still lined up in C. The third rule specifies how the index is decremented upon moving an atomic subexpression e_{at} from C to S.

Completing the traversal of a constructor expression is captured by the first two rules. A constructor with arity 0 on top of M indicates that all its subexpressions have been moved to S, i.e., none are left in C, and the traversal of the entire expression can be completed by moving the constructor from M to S. The two rules distinguish between the trace stack underneath being empty and another constructor being underneath, in which case its index must be decremented to notify completion in S of one of its subexpressions. Discriminating between these two trace stack configurations is required in several of the state transition rules of the full FN_SE(M)CD-machine.

The machine terminates with both M and C being empty as there is no rule with which to continue.

The sequence of state transitions shown in fig. 11 illustrates how this machine traverses the expression $\kappa\, a\, \kappa\, b\, c$, where κ is assumed to be a binary constructor,

$$\begin{tikzcd}
\begin{array}{r} \kappa : a : \kappa : b : c : nil \mid C \\ nil \mid M \\ nil \mid S \end{array} \arrow[r, Rightarrow] & \begin{array}{r} a : \kappa : b : c : nil \mid C \\ \kappa^{(2)} : nil \mid M \\ nil \mid S \end{array} \arrow[d, Rightarrow] \\
\begin{array}{r} b : c : nil \mid C \\ \kappa^{(2)} : \kappa^{(1)} : nil \mid M \\ a : nil \mid S \end{array} \arrow[d, Rightarrow] & \begin{array}{r} \kappa : b : c : nil \mid C \\ \kappa^{(1)} : nil \mid M \\ a : nil \mid S \end{array} \arrow[l, Rightarrow] \\
\begin{array}{r} c : nil \mid C \\ \kappa^{(1)} : \kappa^{(1)} : nil \mid M \\ b : a : nil \mid S \end{array} \arrow[r, Rightarrow] & \begin{array}{r} nil \mid C \\ \kappa^{(0)} : \kappa^{(1)} : nil \mid M \\ c : b : a : nil \mid S \end{array} \arrow[d, Rightarrow] \\
\begin{array}{r} nil \mid C \\ nil \mid M \\ \kappa : \kappa : c : b : a : nil \mid S \end{array} & \begin{array}{r} nil \mid C \\ \kappa^{(0)} : nil \mid M \\ \kappa : c : b : a : nil \mid S \end{array} \arrow[l, Rightarrow]
\end{tikzcd}$$

Fig. 11. Traversing the expression $\kappa\ a\ \kappa\ b\ c$ from stack C to S via stack M

as for instance the applicator @, and a, b, c are atomic expressions. The traversal begins with the stack configuration in the upper left, which has the expression set up in C (with M and S being empty), and continues along the double arrows until it terminates with the stack configuration in the lower left, which has the expression in left-right transposed pre-order linearized form reconstructed in S; M and C are empty.

It is interesting to note that this mechanism recursively brings about configurations in which the components of binary constructor expressions are spread out over the tops of the stacks involved, i.e., with a constructor whose index is 1 on top of M, its right subexpression on top of C and its left subexpression on top of S (the third and fifth configurations). If the expressions would be more complex, e.g., applications of abstractions, then such configurations could be easily intercepted to perform β-reductions by popping their components off the tops of the stacks and pushing into S their values instead.

Using stack M as a temporary storage for constructor symbols is the key to performing almost all state transformations in the FN_SE(M)CD-machine as local operations that involve just stack tops, which are fairly straightforward to implement in a real machine.

6.2 The State Transition Rules

A state of the full FN_SE(M)CD-machine is described by a six-tuple

$$(S,\ E,\ M,\ C,\ D,\ U)\ ,$$

which, other than including the trace stack M and replacing the ULC variable u with the stack U, is the same as that of the FN_SECD-machine of section 5. The structures saved on (and unsaved from) the dump accordingly change to $(U,\ M,\ C,\ D)$.

The applicators @ or the abstractors Λ that appear on top of the trace stack, in conjunction with the indices attached to them, play a decisive role in almost all state transition rules. As before, these rules are in the order in which they need to be checked against machine states given in fig. 12[14].

Rule (1) prepares in one step applications for evaluation. It does so by creating in S a suspension for the operand expression e_a and by pushing the applicator with index 1 into the trace stack M, indicating that only its operator expression remains in C. This transformation, which in fact interchanges the positions of operator and operand relative to the applicator, can be split up into the following sequence of simpler state transitions

$$(1a)\ (S,\ E,\ M,\ @\,e_f\,e_a : C,\ D,\ U)\ \rightarrow\ (S,\ E,\ M,\ @ : e_a : e_f : C,\ D,\ U)$$

$$(1b)\ (S,\ E,\ M,\ @ : e_a : e_f : C,\ D,\ U)\ \rightarrow\ (S,\ E,\ @^{(2)} : M,\ e_a : e_f : C,\ D,\ u)$$

$$(1c)\ (S,\ E,\ @^{(2)} : M,\ e_a : e_f : C,\ D,\ U)\ \rightarrow\ ([\,E\,e_a\,] : S,\ E,\ @^{(1)} : M,\ e_f : C,\ D,\ U)$$

[14] The constructor κ that is used in rules (4b) and (5b) stands for either @ or Λ.

which, after having flipped operator and operand, follows more closely the elementary traversal steps specified in the preceding subsection.

Rule (2) applies abstractions directly to operands in S (which due to prior application of rule (1) are bound to be suspensions). Rule (3) has unapplied abstractions prepend current ULC values (incremented by one) to the active environment; it also pushes the ULC into stack U and the abstractor into the trace stack. Rules (4a/b) effect environment accesses for deBruijn indices occuring on top of the code structure C. Since pushing into S an environment entry

Returning from β-reductions_in_the_large
(0) $(S,\ E,\ nil,\ nil,\ (U',\ M',\ C',\ D'),\ U)\ \rightarrow\ (S,\ U',\ M',\ C',\ D',\ U')$

Spreading applications on top of C out over $C,\ M,\ S$
(1) $(S,\ E,\ M,\ @\,e_f\,e_a : C,\ D,\ U)\ \rightarrow\ ([\,E\ e_a\,] : S,\ E,\ @^{(1)} : M,\ e_f : C,\ D,\ U)$

Applying abstractions on top of C to operands on top of S
(2) $(e_a : S,\ E,\ @^{(1)} : M,\ \Lambda e_b : C,\ D,\ U)\ \rightarrow\ (S,\ e_a : E,\ M,\ e_b : C,\ D,\ U)$

η-extending unapplied abstractions on top of C
(3) $(S,\ E,\ M,\ \Lambda e_b : C,\ D,\ u : U)\ \rightarrow\ (S,\ (u+1) : E,\ \Lambda^{(1)} : M,\ e_b : C,\ D,\ (u+1) : u : U)$

Substituting deBruijn indices by environment entries
(4a) $(S,\ E,\ nil,\ \#i : C,\ D,\ u : U)\ \rightarrow\ (lookup(\#i,\ u,\ E) : S,\ E,\ nil,\ C,\ D,\ u : U)$

(4b) $(S,\ E,\ \kappa^{(j)} : M,\ \#i : C,\ D,\ u : U)\mid(j>0)\ \rightarrow\ (lookup(\#i,\ u,\ E) : S,\ E,\ \kappa^{(j-1)} : M,\ C,\ D,\ u : U)$

Applying closures on top of S to operands underneath
(5a) $([\,E'\ \Lambda e_b\,] : e_a : S,\ E,\ @^{(0)} : nil,\ C,\ D,\ U)\ \rightarrow\ (S,\ e_a : E',\ nil,\ e_b : nil,\ (U,\ nil,\ C,\ D),\ U)$

(5b) $([\,E'\ \Lambda e_b\,] : e_a : S,\ E,\ @^{(0)} : \kappa^{(j)} : M,\ C,\ D,\ U)\mid(j>0)\ \rightarrow$
$(S,\ e_a : E',\ nil,\ e_b : nil,\ (U,\ \kappa^{(j-1)} : M,\ C,\ D),\ U)$

Setting suspensions on top of S up for evaluation
(6) $([\,E'\ e_a\,] : S,\ E,\ M,\ C,\ D,\ U)\ \rightarrow\ (S,\ E',\ nil,\ e_a : nil,\ (U,\ M,\ C,\ D),\ U)$

Constructing abstractions on top of S from their components on M and C
(7a) $(e_b : S,\ E,\ \Lambda^{(0)} : nil,\ nil,\ D,\ u : U)\ \rightarrow\ (\Lambda e_b : S,\ E,\ nil,\ nil,\ D,\ U)$

(7b) $(e_b : S,\ E,\ \Lambda^{(0)} : \Lambda^{(1)} : M,\ nil,\ D,\ u : U)\ \rightarrow\ (\Lambda e_b : S,\ E,\ \Lambda^{(0)} : M,\ nil,\ D,\ U)$

Moving abstractions from S to C
(8) $(\Lambda e_b : S,\ E,\ @^{(0)} : M,\ C,\ D,\ U)\ \rightarrow\ (S,\ E,\ @^{(1)} : M,\ \Lambda e_b : C,\ D,\ U)$

Setting tail suspensions up for evaluation
(9) $(e_b : [\,E'\ e'_a\,] : S,\ E,\ @^{(0)} : M,\ C,\ D,\ U)\ \rightarrow\ (S,\ E',\ nil,\ e'_a : nil,\ (U,\ @^{*} : M,\ e_b : C,\ D),\ U)$

Returning from evaluating tail suspensions
(10) $(e_a : S,\ E,\ @^{*} : M,\ e_b : C,\ D,\ U) \rightarrow\ (e_b : e_a : S,\ E,\ @^{(0)} : M,\ C,\ D,\ U)$

Reconstructing irreducible applications on S
(11a) $(e_b : e_a : S,\ E,\ @^{(0)} : nil,\ C,\ D,\ U)\ \rightarrow\ (@\,e_b\,e_a : S,\ E,\ nil,\ C,\ D,\ U)$

(11b) $(e_b : e_a : S,\ E,\ @^{(0)} : \kappa^{(j)} : M,\ C,\ D,\ U)\mid(j>0)\ \rightarrow\ (@\,e_b\,e_a : S,\ E,\ \kappa^{(j-1)} : M,\ C,\ D,\ U)$

Fig. 12. The state transition rules of a fully normalizing SE(M)CD-machine

accessed by a deBruijn index found on C is in fact equivalent to moving this entry from C to S, rule (4b) must also decrement the index attached to the constructor symbol on top of M to signal completion of the traversal of one of its subexpressions.

Environment accesses that return on S suspensions which are substituted for deBruijn indices in head (or operator) positions are taken care of by rules (5a/b) and (6) [15]. The special case that such a suspension is in fact a closure, i.e., contains an abstraction, effects creation of a new context in which the abstraction body is set up for evaluation in the environment that comes with the closure. This environment is prepended by the operand of the particular application that in S is found underneath the closure. The general case of a suspension on top of S leads to the creation of a new context in which it is evaluated. The old context is in either case saved on the dump.

Suspensions in the tails of apply nodes that are left over in a head-normalized spine are by rule (9) set up for evaluation in new contexts. Being in the tail of an application whose head (operator) is already evaluated is identified as being the second-to-top entry in the value stack S relative to an applicator $@^{(0)}$ on top of M. Entering and returning from evaluating a tail suspension necessitates a reverse traversal step which moves the operator expression back to the control structure C in order to bring the suspension to the top of S prior to creating a new context. This can be made more explicit by splitting rule (9) up in two consecutive steps:

$$(9a)\ (e_b : [\,E'\ e'_a\,] : S,\ E,\ @^{(0)} : M,\ C,\ D,\ U)\ \rightarrow\ ([\,E'\ e'_a\,] : S,\ E,\ @^{*} : M,\ e_b : C,\ D,\ U)$$

$$(9b)\ ([\,E'\ e'_a\,] : S,\ E,\ @^{*} : M,\ e_b : C,\ D,\ U)\ \rightarrow\ (S,\ E',\ nil,\ e'_a : nil,\ (U,\ @^{*} : M,\ e_b : C,\ D),\ U)$$

The special applicator $@^{*}$ serves as a marker that signifies evaluation of its tail suspension. Upon returning from the new context, rule (10) uses this applicator to restore the original stack configuration with the suspension, now evaluated to e_a, underneath e_b on S and the applicator $@^{(0)}$ again on top of M.

Returning from evaluating a suspension in another context is accomplished by rule (0). It intercepts a configuration with an empty control structure and an empty trace stack, which signals completion of traversing, and thereby evaluating, an expression from C to S, and restores the calling context saved as return continuation on the dump D. Note that the contents of stack U that are included in the return continuation becomes the new environment.

The remaining rules (7a/b) and (11a/b) are to construct in S from the @s and Λs that have accumulated in the trace stack the *apps* and *lambs* sequences, respectively, of a head-normalized spine.

6.3 Head-Normalizing the Expression of Subsection 5.3

Fig. 13 illustrates, again as a sequence of stack configurations, how the FN_SE(M) CD-machine reduces the spine of a Λ-expression to head-normal form. To expose

[15] Note that a suspension on top of S being in head (operator) position of an application is identified by the index 0 attached to the applicator that sits on top of M.

$$\text{Rule 3 twice} \quad \Downarrow$$

$$\begin{array}{r}
nil \mid S \\
2:1:nil \mid E \\
\Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
@\,@\,\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2\ \#1\ \#0 : nil \mid C \\
nil \mid D \\
2:1:0:nil \mid U
\end{array}$$

$$\text{Rule 1 twice} \quad \Downarrow$$

$$\begin{array}{r}
[\,2:1:nil\ \ \#1\,] : [\,2:1:nil\ \ \#0\,] : nil \mid S \\
2:1:nil \mid E \\
@^{(1)} : @^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2 : nil \mid C \\
nil \mid D \\
2:1:0:nil \mid U
\end{array}$$

$$\text{Rule 2 twice} \quad \Downarrow$$

$$\begin{array}{r}
nil \mid S \\
[\,2:1:nil\ \ \#0\,] : [\,2:1:nil\ \ \#1\,] : 2:1:nil \mid E \\
\Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2 : nil \mid C \\
nil \mid D \\
2:1:0:nil \mid U
\end{array}$$

$$\text{Rule 3 once} \quad \Downarrow$$

$$\begin{array}{r}
nil \mid S \\
3:[\,2:1:nil\ \ \#0\,] : [\,2:1:nil\ \ \#1\,] : 2:1:nil \mid E \\
\Lambda^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
@\,@\,\Lambda\,\#4\ \#3\ \#2 : \Lambda : nil \mid C \\
nil \mid D \\
3:2:1:0:nil \mid U
\end{array}$$

$$\text{Rule 1 twice} \quad \Downarrow$$

$$\begin{array}{r}
[\,E'\ \ \#3\,] : [\,E'\ \ \#2\,] : nil \mid S \\
E' \ = \ 3:[\,2:1:nil\ \ \#0\,] : [\,2:1:nil\ \ \#1\,] : 2:1:nil \mid E \\
@^{(1)} : @^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
\Lambda\,\#4 : nil \mid C \\
nil \mid D \\
3:2:1:0:nil \mid U
\end{array}$$

$$\text{Rule 2 once} \quad \Downarrow$$

$$\begin{array}{r}
[\,E'\ \ \#2\,] : nil \mid S \\
[\,E'\ \ \#3\,] : 3:[\,2:1:nil\ \ \#0\,] : [\,2:1:nil\ \ \#1\,] : 2:1:nil \mid E \\
@^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
\#4 : nil \mid C \\
nil \mid D \\
3:2:1:0:nil \mid U
\end{array}$$

$$\text{Rule } 4b \text{ once} \quad \Downarrow$$

$$\begin{array}{r}
\#1 : [\,E'\ \ \#2\,] : nil \mid S \\
[\,E'\ \ \#3\,] : 3:[\,2:1:nil\ \ \#0\,] : [\,2:1:nil\ \ \#1\,] : 2:1:nil \mid E \\
@^{(0)} : \Lambda^{(1)} : \Lambda^{(1)} : \Lambda^{(1)} : nil \mid M \\
nil \mid C \\
nil \mid D \\
3:2:1:0:nil \mid U
\end{array}$$

Fig. 13. Snapshots of typical FN_SE(M)CD-machine configurations while head-normalizing the Λ-expression $\Lambda\Lambda\,@\,@\,\Lambda\Lambda\Lambda\,@\,@\,\Lambda\,\#4\ \#3\ \#2\ \#1\ \#0$

the differences relative to the FN_SECD-machine of the preceding section, we have chosen as an example again the expression

$$\Lambda\Lambda @ @ \Lambda\Lambda\Lambda @ @ \Lambda \#4 \#3 \#2 \#1 \#0$$

as in fig. 10 in order to be able to compare on a step-by-step basis how both machines are working.

The main difference that immediately catches the eye is that the dump structure D of the FN_SE(M)CD remains empty throughout the entire head-normalizing sequence since none of the η-extensions or β-distributions creates a new context. This has been made possible by eliminating the superfluous steps of creating closures for abstractions that can be directly applied or η-extended, as a consequence of which the machine also accomplishes with one rule application each the same as the FN_SECD-machine with two.

The applicators and abstractors encountered along the spine, minus the @s that are being consumed by β-reductions/distributions, are now building up in the trace stack M, cleanly separated from the code structure that must accommodate them in the FN_SECD-machine. In the head-normalized configuration at the bottom, this trace includes in the form of a flat sequence exactly those constructor symbols from which the spine of the full normal form must be assembled.

This sequence of machine configuration also shows how the stack U up to the point of head-normalization grows to three non-zero entries pushed by the three unapplied Λs along the spine.

Most importantly, it may be noted that the items stacked up in the value stack S and in the environment structure E are in each of the configurations exactly the same as in the equivalent configurations of the FN_SECD-machine (compare fig. 10), just as it should be.

Going through essentially the same sequences of state transformations when head-normalizing the same expression renders it reasonable to assume that both machines also compute the same normal forms. This may be concluded from the fact that reducing left-over tail suspensions of head-normalized spines is nothing but recursively applying head-normalization to the spines of the respective expressions in their own environments.

7 A Fully Normalizing Graph Reducer

We will now briefly outlined a conceivable implementation of the FN_SE(M)CD-machine as a graph reducer [16].

Graph reduction is the standard implementation technique for functional languages, particularly for those with a lazy semantics [Joh84, PeyJ92, PvE93]. The idea is to represent Λ-expressions in a form that hides (sub)expressions of constructor symbols and also environment structures recursively behind pointers and to perform reductions primarily as pointer manipulations, which typically brings down from $O(n^2)$ to $O(n)$ the runtime complexity of programs that

[16] The contents of this section are in parts adopted from the authors monograph on Abstract Computing Machines [Kge05].

operate on data structures of size n. Another important advantage of graph reduction relates to sharing the evaluation of (sub)expressions among several points of substitution. This technique is the key to optimizing normal order reduction strategies with regard to numbers of reduction steps performed.

Under a head-order regime, opportunities for sharing primarily arise whenever suspensions are copied from the environment into head positions of spines. Reducing them in these places may then be shared with the environment and thus with all occurrences of pointers to the suspensions from elsewhere.

7.1 Graphs and Graph Reduction

Following its syntax, graph representations of expressions of the pure Λ-calculus may be composed of just two types of inner nodes, these being the constructors @ and Λ, and of deBruijn indices as the only type of leaf nodes. The nodes are connected by directed edges leading from root nodes (to root nodes) to leaf nodes. In a conceivable implementation, the inner nodes may be represented as cells in a memory section called the heap which, in addition to the node symbols themselves, also include pointers to subgraphs. The leaf node cells just contain deBruijn indices (or other atomic symbols such as constant values, primitive function symbols, etc. of an applied λ calculus). Entire *lambs* sequences of length n may be represented by single node cells of the form $[\ \Lambda,\ n,\ p_h\ ]$ (with p_h being the pointer to the subgraph that represents the abstraction body) as no other graph structures are branching off, and apply nodes by cells of the form $[\ @,\ p_h,\ p_t\ ]$ (with $p_h,\ p_t$ being the pointers to the head and tail subgraphs, respectively). Suspension nodes take the form $[\ sus,\ p_E,\ p_e\ ]$, where sus denotes the node type, p_E is the pointer to the environment, and p_e is the pointer to the tail expression.

For instance, the head form depicted in fig. 4 of section 4 thus translates into the graph shown in fig. 14 below. The tails that are abbreviated here by the symbols e_j, of course, feature similar graph structures of their own.

A graph reducer typically uses such graphs as static (nondestructible) structures, or alternatively equivalent static code, in order to be able to share their application to different sets of nonshareable operands. These graphs are traversed from top to bottom along their spines to build environments from $apps - lambs$ corners encountered and to construct, from the bottom up, new graph structures somewhere else in the heap.

The environment is composed of frames corresponding to $apps - lambs$-corners which in their order of creation are linked by pointers. A frame contains as many pointers to suspensions and as many ULC entries as there are apply nodes in the *apps*-sequence and unapplied Λs in the *lambs*-sequence, respectively, in the particular corner, so that the frame size equals the length of the *lambs*-sequence (or the arity of the abstraction) involved. Each frame is preceded by a header which includes the link pointer to the frame deeper down in the environment, the number of frame entries, and the ULC value that applies in the particular (sub)environment.

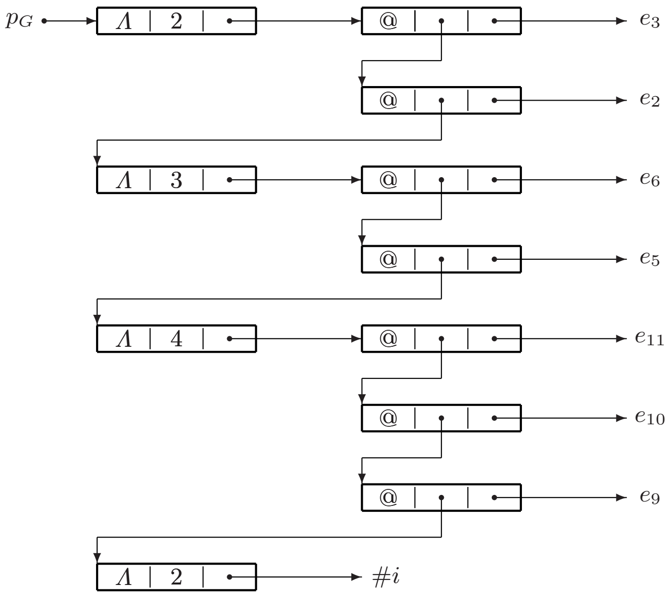


Fig. 14. Graph implementation of the spine of fig. 4

Organizing the environment as a structure of linked frames is due to the need to share, as a matter of efficiency, common parts among environments that belong to different contexts. A typical example is a tail suspension that comes with an environment E_1 contained in a larger environment E_2 of which the suspension is an entry. Substituting this suspension for a head index and evaluating it in this place means that new environment frames must be created on top of E_2 but linked up to the environment E_1 by a pointer that bridges the gap between the tops of E_2 and E_1.

As a minor inconvenience, accessing a particular environment entry with some index $\#i$ therefore entails conparing it with the size of the topmost frame and, if found larger, to subtract this size from the index and proceed down the link pointer to the next frame. This step must be repeated until the remaining index falls inside a frame.

The runtime structures necessary to perform graph reductions are the static graph (or equivalent static code), the environment, and a trace stack that serves a slightly different purpose than the one of the FN_SE(M)CD-machine. In fact, other than maintaining a pointer to the leading Λ-cell it assumes more or less the role of its value stack S: it stacks (and unstacks) pointers to the tail suspensions that are being created on the way down a spine and from which, after evaluating the suspensions left over in the trace stack following head normalization, the spine of a fully normalized graph (or subgraph) is reconstructed.

Reducing a graph sets out with an empty environment, a ULC set to 0, and a graph pointer p_G pointing to the topmost node cell of the graph. The pointer to this cell becomes the first entry of the trace stack.

The graph pointer then advances down the spine along the chain of head pointers p_h and pushes into the trace stack pointers to suspensions for tail expressions until the next Λ-cell is reached. This Λ-cell now controls the creation of an environment frame by filling it from the trace stack either with suspension pointers as the cell's arity parameter demands or until these pointers are prematurely exhausted, in which case the topmost Λ-cell (the pointer to which is at the bottom of the trace stack) pops to the top, indicating unapplied Λs. In this latter case, the remaining frame entries are filled with ULCs and the number of ULCs pushed is added to the arity of the Λ-cell in the trace stack, thus in fact completing an η-extension.

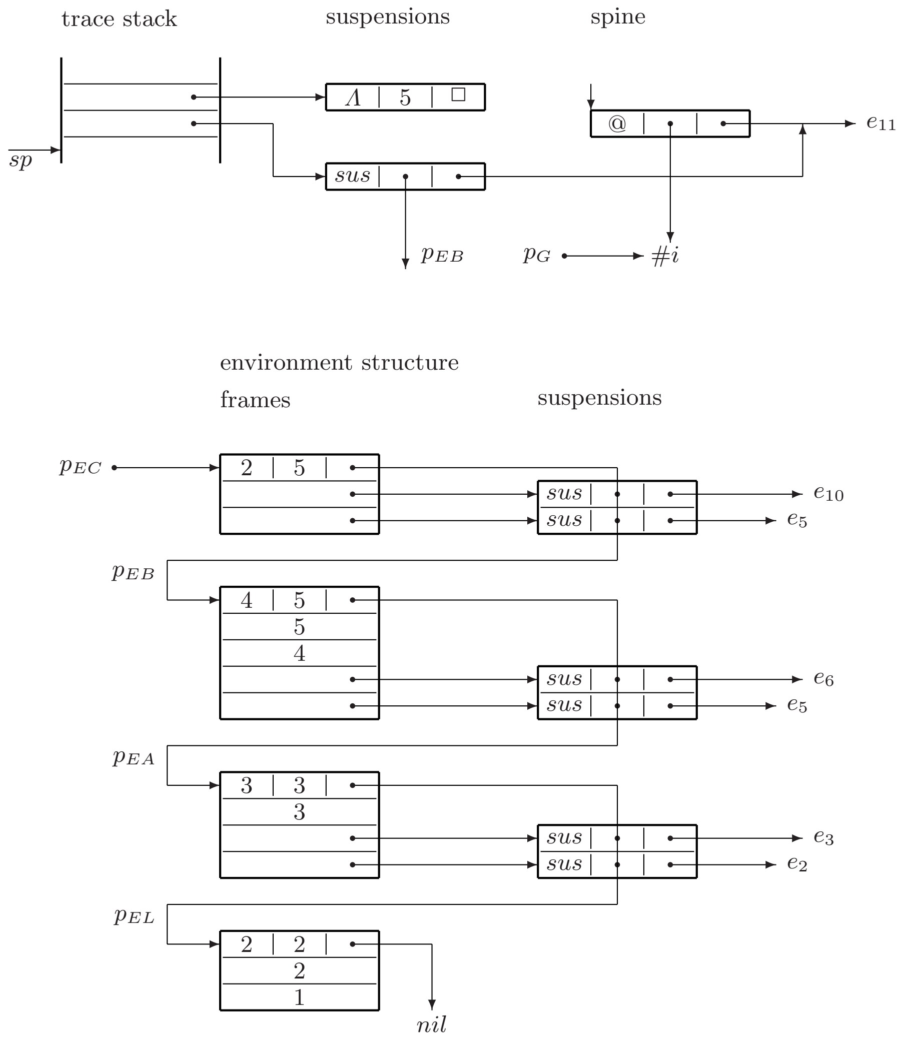


Fig. 15. Trace stack and environment after having reached the head of the spine

Fig. 15 above shows how the trace stack and the environment look like after evaluation has arrived at the head of the spine, as indicated by the position of the graph pointer p_G. The trace stack just has a pointer to the suspension for the tail e_{11} sitting on top of the pointer to the topmost Λ-cell which has its arity index updated to 5. The environment that has built up at this point is composed of four frames corresponding to the three $apps - lambs$ corners of the original spine and a fourth frame at the bottom that is due the η-extension of the leading $lambs$ sequence. Note that the environments pointed to by p_{EC}, p_{EB}, p_{EA} and p_{EL} correspond to cuts C, B, A and L, respectively, in the straightened head form of fig. 8.

7.2 Continuing with Reductions in the Head

If the head index to which p_G is pointing selects from the environment another suspension, then in a straightforward approach its graph would have to be substituted for this head index and head-order reduction would have to continue along the extended spine in the environment carried along with the suspension. This may be accomplished by setting the graph pointer p_G and the environment pointer p_E to those found in the suspension node, and by setting the ULC to that of the topmost frame of the environment that comes with the suspension. If the suspension thus substituted is, or reduces to, an abstraction, then it creates a new environment frame by either consuming suspensions already held in the trace stack or by filling it with ULC entries.

The problem with these old suspensions held in the trace stack is that they constitute specific instantiations of the abstraction that renders reductions further down the extended spine dependent on them, i.e., evaluation of the suspension cannot be shared.

Sharing requires that a suspension selected by the head index be first reduced in isolation and that then the suspension node be overwritten with the resulting graph so that it can be seen by all pointers directed at it from somewhere else. The pointer to this graph may then be substituted for the head index, and reduction may continue along its spine.

The question that remains to be answered is just how far should the suspension be reduced in isolation. The safe thing to do is to reduce it just to head-normal form. If one exists, then we have a chance to reach a full normal form for the entire expression as well, but not necessarily so. If the result of head-normalizing the suspension would be an abstraction and the machine, on its way up the spine, would continue evaluating tail suspensions, some of them might not terminate, i.e., the computation would get trapped in a 'black hole'. However, if evaluating the suspension would stop with a head-normalized abstraction, and this abstraction would be applied to tail suspensions left in the trace stack, there might be a chance that the non-terminating tails are thrown away and the computation could terminate with a full normal form.

Substituting a head-normalized suspension for a head index is depicted in fig. 16. The upper part shows a typical trace stack and the lower end of a static graph with a deBruijn index in its head; the lower part shows the head-normalized spine of the suspension selected by the head index. The substitution

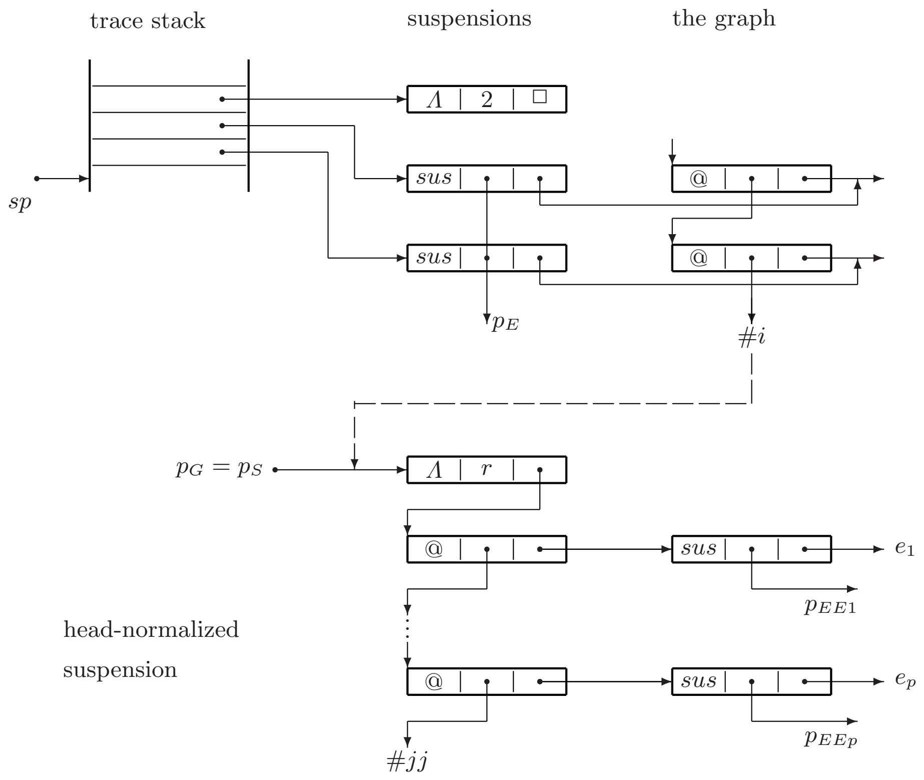


Fig. 16. Linking up to a head-normalized suspension in the head of a spine

is done simply by overwriting the graph pointer p_G with the pointer p_S to the topmost node of the new spine. The evaluation then continues by processing the *apps – lambs* corner formed by the two suspensions held in the trace stack and by the Λ-node to which p_G is now pointing.

Once the computation has arrived at a head-normal form, identified by a *ULC* value retrieved from the environment, the machine finally reverses gear, evaluates the suspensions left on the trace stack, and recursively constructs from the bottom up a spine of apply nodes (with a head index at the bottom) that are linked up by head pointers and whose tail pointers point to the normalized tail graphs. Finally, when arriving at the single Λ-node at the bottom of the trace stack, the spine is completed upwards by as many Λs as specified by the arity index in that node.

8 Related Work and Conclusion

Research on fully normalizing λ-calculus machines dates back to 1975 when Berkling proposed a string reduction machine whose most important operation was a complete and direct implementation of the β-reduction rule [Ber75].

To demonstrate the feasibility of the underlying concept, this proposal led to the construction of an experimental hardware machine that was completed in 1979 [Kge79]. To overcome the performance problems inherent in string reduction, Hommes published in 1982 an early graph reduction version of this machine [Hom82]. Also to mention is Wadsworth's work on a λ-calculus-based graph reducer [Wads71] which represents binding structures by pointers and performs β-reductions by pointer rearrangements.

In 1986 Berkling came up with the more sophisticated head-order reduction concept outlined in section 4 that describes environments completely within the framework of the nameles Λ-calculus. It employs η-extensions-in-the-large to fill in binding indices for missing arguments of abstractions and β-distributions-in-the-large to distribute environments over the components of applications [Ber86]. An alternative theoretical approach which achieves the same ends by slightly different means is the $\lambda\sigma$-calculus of Abadi, Cardelli, Curien and Levi [ACCL90]. It introduces environments through the notion of substitutions as an extension of the nameless Λ-calculus. Full normalization can be achieved by weakly normalizing machinery which must call upon a mechanism that pushes substitutions across unapplied abstractors and updates binding indices accordingly, which is equivalent to η-extension.

Based on Berklings work, Troullinos gives in his PhD thesis a formal specification of an abstract head-order reduction machine that, by a skillful choice of state transition rules, completely avoids a dump, using instead a direction parameter to distinguish between going up or down a spine [Trou93]. It also introduces the notion of an unapplied lambdas count as used by the SECD-machine descendants described in this paper. Crégut describes a fully normalizing abstract machine based on Krivine's weakly normalizing $\mathcal{K}$-machine [Kri85]. In the course of evaluating suspensions, it uses an updating scheme for binding indices that is similar to ULCs. Another fully normalizing machine is due to Grégoire and Leroy [GrLe02]. It augments a weakly normalizing machine by a so-called rollback function which, following the classical definition of the β-reduction rule, α-converts λ-bound variables in order to take them out of naming conflicts.

An early implementation of Berkling's head-order reduction concept is described in the PhD thesis by Hilton [Hil90]. An instruction-based fully-normalizing head-order reducer that makes extensive use of sharing reductions in the head is described in [Kge05]. This B-machine has been reconstructed from various unpublished drafts and handwritten notes by Berkling [Ber96, Ber97], and from the PhD theses by Hilton [Hil90] and Troullinos [Trou93].

Another instruction-based machine described in this monograph and earlier published in [GK96] employs weak normalization to do the routine work of naive substitutions when reducing full applications but switches to a special η-extension mechanism to deal with unapplied abstractions. Rather than filling in ULCs for missing arguments, it re-introduces the original variables, doing the equivalent of the transformation:

$$(\underbrace{\Lambda \ldots \Lambda}_{n}\, e_0\, e_1 \ldots e_k\ ) \mid k < n \ \rightarrow\ \lambda v_{k+1} \ldots \lambda v_n((\underbrace{\Lambda \ldots \Lambda}_{n}\, e_0\, e_1 \ldots e_k\ )\, v_{k+1} \ldots v_n).$$

The variables $\{ \ v_{k+1} \ldots v_n \ \}$ are being retrieved from persistent structures in which have been saved the full parameter sets of all λ-abstractions of a program before converting them into equivalent nameless Λ-abstractions for reduction.

This η-extension mechanism makes use of the fact that unapplied abstractors always end up in a leading λ-sequence that never engages in further β-reductions but becomes part of the full normal form. Different instances of the same variables are distinguished by subscripts that enumerate the η-extension steps by which they have been introduced.

The two abstract machines described in this paper have been devised for the purpose of the summer school to convey the basic message that, starting from the well-known weakly normalizing SECD-machine, fully normalizing machines can be had by supplementing them with a few more state transition rules that η-extend unapplied abstractions, thus elegantly getting by the trouble of implementing full-fledged β-reductions. The key to performing these η-extensions with very little overhead is the use of ULC-indices rather than binding indices. It takes very simple updates to turn ULCs retrieved from the environment into deBruijn indices, in which form they must show up in (head-)normalized Λ-expressions.

Both the FN_SECD-machine and the FN_SE(M)CD-machine have been tested with the same set of about 25 example λ-expressions, all of which include in various contexts unapplied abstractions that require η-extension. Both machines have been found to reduce these expressions correctly to full normal forms. Of course, this is no proof that they work correctly for all Λ-expressions but it raises the level of confidence considerably.

References

[ACCL90] Abadi, M., Cardelli, L., Curien, P.-L., Levi, J.-J.: Explicit Substitutions. In: Proceedings of the 17th ACM Symposium on Principles of Programming Languages, pp. 1–16. ACM Press, New York (1990)

[Bar84] Barendregt, H.P.: The Lambda Calculus, Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam (1984)

[Ber75] Berkling, K.J.: Reduction Languages for Reduction Machines. In: Berkling, K.J. (ed.) Proceedings of the 2nd Annual Symposium on Computer Architecture, pp. 133–140. ACM/IEEE (1975)

[Ber86] Berkling, K.J.: Head-Order Reduction: a Graph Reduction Scheme for the Operational Lambda Calculus. LNCS, vol. 254, pp. 26–48. Springer, Heidelberg (1986)

[Ber96] Berkling, K.J.: The von Neumann-PLUS Architecture: an Evolutionary Development. unpublished draft (1996)

[Ber97] Berkling, K.J.: Privately communicated handwritten notes, Syracuse, NY (Spring 1997)

[Bird98] Bird, R.S.: Introduction to Functional Programming with Haskell, 2nd edn. Prentice-Hall, Englewood Cliffs (1998)

[Bru72] de Bruijn, N.G.: A Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church–Rosser Theorem. Indagationes Mathematicae 34, 381–392 (1972)

[CMQ83] Cardelli, L., McQueen, D.: The Functional Abstract Machine,The ML/LCF/HOPE Newsletter. AT&T Bell Labs, Murray Hill, NJ (1983)

[Chu41] Church, A.: The Calculi of Lambda Conversion. Princeton University Press, Princeton (1941)

[CCM85/87] Cousineau, G., Curien, P.L., Mauny, M.: The Categorial Abstract Machine. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 50–64. Springer, Heidelberg (1985); Science of Computer Programming, No. 8, 173–202 (1987)

[Cre90] Crégut, P.: An Abstract Machine for the Normalization of λ-Terms. In: Proceedings of the ACM Conference on LISP and Functional Programming, pp. 333–340 (1990)

[Dyb87] Dybvig, R.K.: The SCHEME Programming Language. Prentice-Hall, Englewood Cliffs (1987)

[GK96] Gaertner, D., Kluge, W.E.: π-RED$^+$ – an Interactive Compiling Graph Reduction System for an Applied λ-Calculus. Journal of Functional Programming 6(5), 723–757 (1996)

[GrLe02] Grégoire, B., Leroy, X.: A Compiled Implementation of Strong Reduction. In: Proceedings of the ACM International Conference on Functional Programming, pp. 235–246 (2002)

[Hil90] Hilton, M.L.: Implementation of Declarative Languages, PhD thesis, CASE Center Technical Report No. 9008. Syracuse University, Syracuse, NY (1990)

[HS86] Hindley, J.R., Seldin, J.P.: Introduction to Combinators and λ-Calculus. London Mathematical Society Student Texts. Cambridge University Press, Cambridge (1986)

[Hom82] Hommes, F.: The Heap/Substitution Concept - an Implementation of Functional Operations on Data Structures for a Reduction Machine. In: Proc. 9th Annual Symposium on Computer Architecture, Austin (Texas), pp. 248–256 (1982)

[Joh84] Johnsson, T.: Efficient Compilation of Lazy Evaluation. ACM Conference on Compiler Construction, Montreal, Que., pp. 58–69 (1984)

[Kge79] Kluge, W.E.: The Architecture of the Reduction Machine Hardware Model, Internal Report GMD ISF-79-3, St. Augustin, Germany (1979)

[Kge05] Kluge, W.: Abstract Computing Machines – A Lambda Calculus Perspective. Springer, New York (2005)

[Kri85] Krivine, J.-L.: Un interpréte du λ-calcul. (1985)

[Lan64] Landin, P.J.: The Mechanical Evaluation of Expressions. The Computer Journal 6(4), 308–320 (1964)

[PeyJ92] Peyton Jones, S.L.: Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-Machine. Journal of Functional Programming 2(2), 127–202 (1992)

[PvE93] Plasmeijer, R., van Eekelen, M.: Functional Programming and Parallel Graph Rewriting. Addison-Wesley, Reading (1993)

[Trou93] Troullinos, N.B.: Head-Order Techniques and Other Pragmatics of Lambda Calculus Graph Reduction, PhD thesis, CASE Center Technical Report No. 9322, Syracuse University, Syracuse, NY (1993)

[Ull98] Ullman, J.U.: Elements of MLProgramming, 2nd edn. Prentice-Hall, Englewood Cliffs (1998)

[Wads71] Wadsworth, C.P.: Semantics and Pragmatics of the Lambda Calculus, PhD thesis, Oxford University (1971)

Programming in Ωmega

Tim Sheard and Nathan Linger

Computer Science Department
Maseeh College of Engineering and Computer Science
Portland State University

Abstract. This report was originally prepared as notes for a short course on Ωmega taught at the Central-European Functional Programming School held in Cluj, Romania, between 25-30 June, 2007. It can be viewed as a tutorial on the use of the Ωmega programming language.

It introduces readers to the *types as propositions* notion based upon the Curry-Howard isomorphism. Such types can express precise properties of programs. The Ωmega language allows us to use a single language for the specification of designs, the definition of properties, the implementation of programs, and the production of proofs that programs adhere to their properties. Ωmega bundles all these in a coherent manner into a single unified system that appears to the user to be a programming language.

1 Introduction

Ωmega is a language with an infinite hierarchy of computational levels: value, type, kind, sort, etc. Data, and functions manipulating data, can be introduced at any level. Data is introduced by declaring the type of constructors, and functions are introduced by writing (possibly recursive) pattern matching equations.

Terms at each level are classified by terms at the next level. Thus values are classified by types, types are classified by kinds, kinds are classified by sorts, etc. As discussed earlier, programmers are allowed to introduce new terms and functions at every level, but any particular program will have terms at only a finite number of levels. We illustrate the level hierarchy for the many of the examples given in this paper in Figure 1.

We maintain a strict phase distinction — the classification of a term at level n cannot depend upon terms at lower levels. For example, no types can depend on values, and no kinds can depend on types. We formalize properties of programs by exploiting the Curry-Howard isomorphism. Terms at computational level n, are used as proofs about terms at level $n+1$. We use indexed types to maintain a strict and formal connection between the two levels, and singleton types to maintain the strict separation between values and types.

← *value name space*	*type name space* →		
value	*type*	*kind*	*sort*
	Tree	:: *0 ~> *0	:: *1
Fork	:: Tree a -> Tree a -> Tree a	:: *0	:: *1
Node	:: a -> Tree a	:: *0	:: *1
Tip	:: Tree a	:: *0	:: *1
	Z	:: Nat	:: *1
	S	:: Nat ~> Nat	:: *1
	plus	:: Nat ~> Nat ~> Nat	:: *1
	{plus 1t 3t }	:: Nat	:: *1
	Seq	:: *0 ~> Nat ~> *0	:: *1
Snil	:: Seq a Z	:: *0	:: *1
Scons	:: a -> Seq a b -> Seq a (S b)	:: *0	:: *1
app	:: Seq a n -> Seq a m -> Seq a {plus n m}	:: *0	:: *1
	Tp	:: Shape	:: *1
	Nd	:: Shape	:: *1
	Fk	:: Shape	:: *1
	Tree	:: Shape ~> *0 ~> *0	:: *1
Tip	:: Tree Tp a	:: *0	:: *1
Node	:: a -> Tree Nd a	:: *0	:: *1
Fork	:: Tree x a -> Tree y a -> Tree (Fk x y) a	:: *0	:: *1
find	:: (a -> a -> Bool) -> a -> Tree sh a -> [Path sh a]	:: *0	:: *1
	T	:: Boolean	:: *1
	F	:: Boolean	:: *1
	le	:: Nat ~> Nat > Boolean	:: *1
	{le 0t 2t}	:: Boolean	:: *1
	LE	:: Nat ~> Nat > *0	:: *1
LeZ	:: LE Z a	:: *0	:: *1
LeS	:: LE n m -> LE (S n) (S m)	:: *0	:: *1
	Even	:: Nat ~> *0	:: *1
EvenZ	:: Even Z	:: *0	:: *1
EvenSS	:: Even n -> Even (S(S n))	:: *0	:: *1

Fig. 1. The level hierarchy for some of the examples in the paper

2 A Simple Example

To illustrate the hierarchy of computational levels we give the following two-level example which uses natural numbers as a type index to lists that record their length in their type.

First, we introduce tree-like data (the natural numbers, `Nat`) at the *type level* by using the `data` introduction form. This form is a generalization over the `data` declaration in Haskell [21].

```
data Nat :: *1 where
  Z :: Nat
  S :: Nat ~> Nat
```

The line "`data Nat :: *1 where`" indicates that `Nat` is classified by `*1` (rather than `*0`), which tells the programmer that `Nat` is a kind (rather than a type), and that `Z` and `S` are types (rather than values) that are classified as indicated. Think of the operator `~>` as the operator that classifies functions at the type level. I.e. it is similar in use to the operator `->`, but used on kinds rather than types. Thus, `S :: Nat ~> Nat` indicates a type constructor that takes a `Nat` as input and produces a `Nat` as output.

The classifiers `*0`, `*1`, `*2`, etc. indicate the level of a term. All values are classified by types that are classified by `*0`. All types are classified by kinds that are classified by `*1`. All kinds are classified by sorts that are classified by `*2`, etc. This is illustrated with great detail in Figure 1.

Second, we write a function at the type level over this data (`plus`). At the type level and higher, we distinguish function application from constructor application by surrounding function application by braces (`{` and `}`). For example, we write `S x` for constructor application, and `{plus x y}` for function application.

```
plus:: Nat ~> Nat ~> Nat
{plus Z m} = m
{plus (S n) m} = S {plus n m}
```

Third, using the `data` introduction form at the *value level*, we introduce the algebraic data structure (`Seq`). The types of such values are indexed by the natural numbers. These indexes describe an invariant about the constructed values — their length appears in their type — consider the type of `l1` below.

```
data Seq:: *0 ~> Nat ~> *0 where
  Snil :: Seq a Z
  Scons:: a -> Seq a n -> Seq a (S n)

l1 = (Scons 3 (Scons 5 Snil)) :: Seq Int (S(S Z))
```

Finally, we introduce an append function at the value level over `Seq` values (`app`). The type of `app` describes one of its important properties — there is a functional relationship between the lengths of its two inputs, and the length of its output.

```
app:: Seq a n -> Seq a m -> Seq a {plus n m}
app Snil ys = ys
app (Scons x xs) ys = Scons x (app xs ys)
```

To see that the `app` is well typed, the type checker does the following. The expected type is the type given in the function prototype. We compute the type of both the left- and right-hand-side of the equation defining a clause. We compare the expected type with the computed type for both the left- and right-hand-sides. This comparison generates some necessary equalities (for each side) to make the expected and computed types equal. We assume the left-hand-side equalities to prove the right-hand-side equalities. To see this in action, consider the second clause of the definition of `app`.

expected type	`Seq a n` $\rightarrow$	`Seq a m` $\rightarrow$	`Seq a {plus n m}`
equation	`app (Scons x xs)`	`ys`	`= Scons x (app xs ys)`
computed type	`Seq a (S b)` $\rightarrow$	`Seq a m` $\rightarrow$	`Seq a (S {plus b m})`
equalities		`n = (S b)` $\Rightarrow$	`{plus n m}= S{plus b m}`

The expected types are taken from the type declaration accompanying the function definition. The computed type is computed[1] from the known types of the constructors and functions in the definition. The equalities are generasted by equating the expected type and the computed type. The left-hand-side equalities (to the left of the ⇒) let us assume `n` = `S b`. The right-hand-side equalities, require us to establish that `{plus n m}` = `S{plus b m}`. Using the assumption that `n` = `S b`, we are left with the requirement that `{plus (S b) m}` = `S{plus b m}`, which is easy to prove using the definition of `plus`.

The different levels of the objects introduced in this example (and elsewhere in the paper) are plotted in Figure 1. The reader may wish to consult the figure to help visualize the relationships involved.

Exercise 1. Write an Ωmega function that defines the length function over sequences. `length:: Seq a n -> Int`. You will need to create a file, and paste the definition for `Seq` into the file, as well as write the length function. The `Nat` kind is predefined. You will need to include the function prototype, above, in your file (type inference is limited in Ωmega). How might we reflect the fact that the resulting `Int` should have size `n`? See Section 3.7.

Exercise 2. After you complete Exercise 1, create a table, as we did for `app` above, with expected type, equations, computed type, and equations to be discharged. How might we solve the equations produced?

3 Features of the Ωmega Language

Ωmega is modelled after the Haskell language. There are several important differences between Ωmega and Haskell that give Ωmega its unique power of expression. These include.

– **Data Structures at All Levels.** Kinds are a type system for classifying types. Sorts are a type system for classifying kinds. There is no practical limit to this hierarchy. In Ωmega, programmers can introduce new tree-like structures at any level. In Haskell all introduced datatypes are classified by `*0`. I.e. the introduced types classify only values. In Figure 1, Haskell types are illustrated by `Tree`, which is a type constructor which classifies its constructor functions (`Fork`, `Node`, and `Tip`) which are values. In Ωmega, the `data` declaration is generalized to all levels.

[1] Using an inference algorithm based upon algorithm-W.

– **GADTs.** Generalized Algebraic Datatypes allow constructor functions to have more general types than the types supported by the data declaration in Haskell. GADTs are important because the additional generality allows the programmer to express properties of types as witness types, proof objects, or singleton types. GADTs are the machinery that support the Curry-Howard isomorphism in *Ω*mega. In Figure 1, the types Seq, LE, and Even require the generality introduced by GADTs.
– **Functions at All Levels.** *Ω*mega supports functions over tree- structured data at all levels. Such functions are written by pattern matching equations, in much the same manner one writes functions over data at the value level in Haskell. We restrict the form of such definitions to be inductively sequential (See Appendix B). This ensures a sound and complete strategy for answering certain type-checking time questions by the use of narrowing. The class of inductively sequential functions is a large one, in fact every Haskell function has an inductively sequential definition. The inductively sequential restriction affects the form of the equations, and not the functions that can be expressed. In Figure 1, plus and le are functions at the type level.
– **Code Constructing Quasi-Quotes.** *Ω*mega supports the run-time generation of code, along the lines of MetaML [24] and Template Haskell [25]. The meta-programming ability of code generation allows us to remove a layer of interpretation from our programs, that makes them efficient as well as general.

Some of the following sections are labeled with *Feature* if they are an addition to Haskell, *Pattern* if they are a paradigmatic use of the features to accomplish a particular end, or *Example* if they illustrate an important concept.

3.1 Feature: Kinds

We can introduce new tree-like data at any level, including the type level and higher. The data declaration introduces both the constructors for tree-like data and the object that classifies these structures. We indicate the level where these objects reside using *0, *1, *2, etc. in the data declaration. Consider the kinds Nat (introduced earlier), and Boolean:

```
data Shape :: *1 where
  Tp:: Shape
  Nd:: Shape
  Fk:: Shape ~> Shape ~> Shape
```

```
data Boolean:: *1 where
  T:: Boolean
  F:: Boolean
```

Like the kind Nat defined earlier, Shape and Boolean also define new kinds, and new types classified by these kinds. The new tree-like data at the type level are constructed by the type-constants (Tp, Nd, T, F, Z), and type constructors (Fk and S). The kinds Shape and Boolean classify these structures, as shown explicitly in the declaration. For example T is classified by Boolean, and Fk is a constructor from Shape to Shape to Shape. Note that while Tp, Nd, T, and F live at the type level, there are no values classified by them. Again, see Figure 1 to see where these objects reside in the larger picture.

Even though there are no values classified by the types introduced by `Nat`, `Shape`, and `Boolean`, they are very useful. Instead of using them to classify values, we use them as indexes to value level data, i.e. types like `Proof {even n}` and `Seq a (S Z)`. The indexes like `{even n}` and `S z` indicate static (type-checking time) properties of values. For example, a value with type `Seq a (S Z)` is statically guaranteed to have length 1.

Exercise 3. Write a data declaration introducing a new kind called `Color` with types `Red` and `Black`. Are there any values with type `Red`? Now write a data declaration introducing a new type `Tree` which is indexed by `Color` (this will be similar to the use of `Nat` in the declaration of `Seq`). There should be some values classified by the type `(Tree Red)`, and others classified by the type `(Tree Black)`.

3.2 Feature: Type Functions

Kind declarations allow us to introduce new tree-like structures at the type level. We can use these structures to parameterize data at the value level as we did with `Nat` indexing `Seq`. We may also compute over these tree-like structures. Such functions are written by pattern matching equations, in much the same manner one writes functions over data at the value level. Several useful functions over types defined earlier are:

```
even :: Nat ~> Boolean
{even Z} = T
{even (S Z)} = F
{even (S (S n))} = {even n}

le:: Nat ~> Nat ~> Boolean
{le Z n} = T
{le (S n) Z} = F
{le (S n) (S m)} = {le n m}
```

```
plus:: Nat ~> Nat ~> Nat
{plus Z m} = m
{plus (S n) m} = S {plus n m}

and:: Boolean ~> Boolean ~> Boolean
{and T x} = x
{and F x} = F
```

Like functions at the value level, the type functions `plus`, `and`, `even`, and `le` are expressed using equations. The function `and` is a binary function that combines two `Booleans`. The property `even` is a unary predicate that distinguishes odd from even numbers, and the property `le` is a binary less-than-or-equal-to predicate. All the functions are strict total (terminating) functions at the type level. Termination is a necessary property of type functions, though this is not currently checked by the system.

Exercise 4. Write an Ωmega function `mult`, which is the multiplication function at the type level over natural numbers. It should be classified by the kind `mult:: Nat ~> Nat ~> Nat`.

Exercise 5. Write the `odd` function classified by the kind `Nat ~> Boolean`.

Exercise 6. Write the `or` and `not'` functions, that are classified by the kinds `(Boolean ~> Boolean ~> Boolean)` and `(Boolean ~> Boolean)`. Use `not'` rather than `not` since the name `not` is already predefined. Which arguments of `or` should you pattern match over? Does it matter? Experiment, *Ω*mega won't allow some combinations. See Appendix B on inductively sequential definitions and narrowing for the reason why.

3.3 Feature: GADTs

Generalized Algebraic Datatypes allow constructor functions to have more general types than the types supported by `data` declaration in Haskell. GADTs are important because the additional generality allows the programmer to express properties of types using type indexes and witnesses (or proof objects). The `data` declaration in *Ω*mega defines generalized algebraic datatypes (GADT). These are characterized by explicitly classifying constructors in a `data` declaration with their full types. The additional generality arises because the range of a constructor in a GADT is not constrained to be the type constructor applied to only type variables. For example consider the value level GADTs `Seq`, `Path` and `Tree`:

```
data Seq:: *0 ~> Nat ~> *0 where
  Snil :: Seq a Z
  Scons:: a -> Seq a n -> Seq a (S n)

data Path:: Shape ~> *0 ~> *0 where
  None :: Path Tp a
  Here :: b -> Path Nd b
  Left :: Path x a -> Path (Fk x y) a
  Right:: Path y a -> Path (Fk x y) a

data Tree :: Shape ~> *0 ~> *0 where
  Tip:: Tree Tp a
  Node:: a -> Tree Nd a
  Fork::  Tree x a -> Tree y a -> Tree (Fk x y) a
```

Note that instead of ranges like `(Seq a b)`, and `(Path a b)` where only type variables like `a`, and `b` can be used as parameters, the ranges contain sophisticated instantiations such as `(Seq a (S n))` and `(Path Nd)`. Note that the second index to `Seq` (the one of kind `Nat`) is used to describe an invariant about the length of the sequence, and the `Shape` index to `Path`, indicates the shape of a tree in which that path is legal. This is one of the many uses of GADTs – to enforce invariants about the structure of data. Notice how the shape of `tree1` appears in its type.

```
tree1 :: Tree (Fk   (Fk   Tp  Nd)       (Fk   Nd       Nd))        Int
tree1 =       Fork (Fork Tip (Node 4)) (Fork (Node 4) (Node 3))
```

We can write pattern matching functions over GADTs just as we can over algebraic datatypes. The only caveat is that we must specify the type of the function using a prototype. Ωmega does type checking of functions over GADTs rather than type inference.

Suppose we wanted to search a tree, returning all paths that lead to a particular element. It would be nice to know that every path returned was a legal path within the tree. For example `(Left (Here 2))` is not a legal path within the tree `Tip`. The `Shape` index allows us to specify that our searching function always returns a value that obeys this legal path invariant.

```
find:: (a -> a -> Bool) -> a -> Tree sh a -> [Path sh a]
find eq n Tip = []
find eq n (Node m) =
  if eq n m then [Here n] else []
find eq n (Fork x y) =
  map Left (find eq n x) ++
  map Right (find eq n y)
```

The type of `find` guarantees that every path returned is a legal path within the tree searched, because both the tree and every path in the list has the same `Shape`, namely `sh`.

Exercise 7. Write an Ωmega function with type `extract:: Path sh a -> Tree sh a -> a`, which extracts the value of type `a`, stored in the tree at the location pointed to by the path. This function will pattern match over two arguments simultaneously. Some combinations of patterns are not necessary. Why? See section 3.10 for how you can document this fact.

Exercise 8. Replicate the shape index pattern for lists. Write two Ωmega GADTs. One at the kind level which encodes the shape of lists, and one at the type level for lists indexed by their shape. Also, write a find function for your new types. `find:: (a -> a -> Bool) -> a -> List sh a -> Maybe(ListPath sh a)`, which returns the first path, if one exists.

Since every GADT is comprised of a sum of products, can you define a single shape kind, that could be used for all parametric datatypes?

Exercise 9. Consider the GADT below.

```
data Rep :: *0 ~> *0 where
   Int :: Rep Int
   Prod :: Rep a -> Rep b -> Rep (a,b)
   List :: Rep a -> Rep [a]
```

Construct a few terms. Do you note anything interesting about this type? Write a function with the following prototype: `showR:: Rep a -> a -> String`, which given values of type `Rep a` and `a`, displays the second as a string. Extend this GADT with a few more constructors, then extend your `showR` function as well.

3.4 Pattern: Witnesses

GADTs can be used to witness relational properties between types. This is because the parameters to types introduced using the GADT mechanism can play different roles. The natural number argument of the type constructor `Seq` (from Section 2) plays a qualitatively different role than type arguments in ordinary ADTs. Consider the declaration for a binary tree datatype in Haskell:

```
data HTree a = HFork (HTree a) (HTree a) | HNode a | HTip
```

In this declaration the type parameter `a` is used to indicate that there are sub components of `HTrees` that are of type `a`. In fact, `HTrees` are parametric. Any type of value can be placed in the “sub component” of type `a`. The type of the value placed there is reflected in the `HTree`’s type. Contrast this with the `n` in `(Seq a n)`, and the `sh` in `(Tree sh a)`. Instead, the parameter `n` is used to stand for an abstract property (the length of the list represented), and the parameter `sh` is used to stand for the shape of the tree. When we use a type parameter in this way we call it a type index [40, 43] rather than a type parameter.

We can use indexes to GADTs to define value level data that we can think of as proofs, or witnesses to type level properties. This is a powerful idea. Consider the introduction of several new indexed types `Proof`, `Plus`, `LE` and `Even`. Note that these are ordinary data structures that exist at the value level, but describe properties at the type level.

```
data Proof:: Boolean ~> *0 where
  Triv:: Proof T

data Plus:: Nat ~> Nat ~> Nat ~> *0
      where
  PlusZ:: Plus Z m m
  PlusS:: Plus n m z ->
          Plus (S n) m (S z)
```

```
data LE:: Nat ~> Nat ~> *0 where
  LeZ:: LE Z n
  LeS:: LE n m ->
        LE (S n) (S m)

data Even:: Nat ~> *0 where
  EvenZ:: Even Z
  EvenSS:: Even n -> Even (S (S n))
```

These declarations introduce value-level constants (`Triv`, `EvenZ`, `PlusZ`, and `LeZ`) and constructor functions (`EvenSS`, `PlusS`, and `LeS`). Values of these types can be used as proofs about the natural numbers.

To make it easier to enter and display types of kind `Nat`, in *Ω*mega, we have special syntactic sugar for them: `Z` = `0t`, `S Z` = `1t`, and `S(S Z)` = `2t`, etc. We may also write `(1+x)t` for `S x`, and `(2+x)t` for `S(S x)`, etc. We introduce this notation here (see Section 3.13 for more detail) to emphasize that we should view `LE`, `Plus` and `Even` as relationships between natural numbers. To emphasize this, let’s examine the types of several values constructed with these constructors.

```
EvenZ:: Even 0t
(EvenSS EvenZ):: Even 2t
(EvenSS (EvenSS EvenZ)):: Even 4t

p1 ::Plus 2t 3t 5t
p1 = PlusS (PlusS PlusZ)
```

```
LeZ:: LE 0t a
(LeS LeZ):: LE 1t (1+a)t
(LeS (LeS LeZ)):: LE 2t (2+a)t

even2 :: Proof {even 2t}
even2 = Triv
```

The important thing to notice is that we may view ordinary values with types `(LE n m)`, `(Even n)`, and `(Proof {even n})` as proofs, since the types of all legally constructed values witness only true statements about `n` and `m`. For example we cannot build a term of type `(Even 1t)`. This is the essence of the Curry-Howard isomorphism.

We can view `(EvenSS EvenZ):: Even 2t` as either the statement that `(EvenSS EvenZ)` has type `(Even 2t)`, or that `(EvenSS EvenZ)` is a proof of the property `(Even 2t)`. In the same fashion, the type system will reject ill-typed terms that witness false statements. For example, consider the response when we try to type the term `Triv` with the type `(Proof {even 1t})`

```
bad:: Proof {even 1t}
bad = Triv

While type checking in the scope of:
   bad
We need to prove:
   Equal {even 1t} T
From the truths:
And the rules:S,Z,
But, The equations: (F=T) =>  have no solution
```

All this follows directly from the introduction of new types as GADTs and the ability to define them, and to compute over them, at arbitrary levels.

Exercise 10. Construct terms with the types `(Plus 2t 3t 5t)`, `(Plus 2t 1t 3t)`, and `(Plus 2t 6t 8t)`. What did you discover?

Exercise 11. Write an Ωmega function with the following prototype:
`summandLessThanOrEqualToSum:: Plus a b c -> LE a c`. Hint: it is a recursive function. Can you write a similar function with type `(Plus a b c -> LE b c)`?

3.5 Pattern: Witness vs. Type Function

The reader may have noticed that `(Proof {even n})` and `(Even n)` are two different ways to express the same notion. Either we write a (`Boolean`) function at the type level (`even`), or introduce a witness type (`Even`) at the value level.

The general principle of replacing a boolean function at the type level with a witness object at the value level, can be further generalized (you can try this in Exercise 12). The type function does not have to have `Boolean` as its range. Instead, for every n-ary function at the type level, we can build an $(n+1)$-ary witness type. We express the equality between a function call and its result: `{function a b} = c` as a relation: `{Relation a b c}`.

The witness type turns the n-ary function into an $(n+1)$-ary type constructor. Each clause in the function definition is named by a constructor function in the witness. If the right-hand-side of a clause has m recursive calls, the constructor function becomes an m-ary constructor. The right-hand-side of each clause becomes the $(n+1)^{st}$ argument of the range, where every recursive call to the function in the right-hand-side, is replaced with a variable. Each recursive call becomes one of the m arguments. The $(n+1)^{st}$ argument to these calls is the new variable replacing the corresponding recursive call in the $(n+1)^{st}$ argument of the range. For example: The clause of the binary function `{plus (S n) m} = S {plus n m}`, becomes a ternary predicate `Plus (S n) m (S {plus n m})`. By replacing the recursive call with `z`, and making `z` be the $(n+1)^{st}$ parameter to the first argument, we get the type of the unary constructor

```
PlusS:: Plus n m z -> Plus (S n) m (S z).
```

Exercise 12. Use the pattern above to define a GADT (a type constructor with 2 arguments) that witnesses the `even` type function.

Witnesses and type functions express the same ideas, but can be used in very different ways. Type functions are only useful at compile-time (they're static) and their structure cannot be observed (they can only be applied, so we say they are extensional). Witnesses, on the other hand, are actual data that is manipulated at run time (they're dynamic). Their structure can also be observed and taken apart (we say they're intensional). They are true data. A big difference between the two ways of representing properties is the computational mechanisms used to ensure that programs adhere to such properties.

3.6 Pattern: Singleton Types

Sometimes it is useful to direct computation at the type level, by writing functions at the value level. Even though types cannot depend on values directly, this can be simulated by the use of singleton types. The idea is to build a completely separate isomorphic copy of the type in the value world, but still retain a connection between the two isomorphic structures. This connection is maintained by indexing the value-world type with the corresponding type-world kind. This is best understood by example. Consider reflecting the kind `Nat` into the value-world by defining the type constructor `SNat` using a `data` declaration.

```
data SNat:: Nat ~> *0 where
  Zero:: SNat Z
  Succ:: SNat n -> SNat (S n)

three = (Succ (Succ (Succ Zero))):: SNat(S(S(S Z)))
```

Here, the value constructors of the `data` declaration for `SNat` mirror the type constructors in the `kind` declaration of `Nat`. We maintain the connection between the two isomorphic structures by the use of `SNat`'s natural number index. This type index is in one-to-one correspondence with the shape of the value. Thus, the type index of `SNat` exactly mirrors its shape. For example, consider the example `three` above, and pay particular attention to the structure of the type index, and the structure of the value with that type.

This kind of relationship between values and types is called a *singleton type* because there is only one element of any singleton type. For example only `Succ`

`(Succ Zero)` inhabits the type `SNat(S (S Z))`. It is possible to define a singleton type for any first order type (of any kind). All singleton types always have kinds of the form `I ~> *0` where `I` is the index we are reflecting into the value world. We sometimes call singleton types *representation types*. We cannot over emphasize the importance of the singleton property. Every singleton type completely characterizes the structure of its single inhabitant, and the structure of a value in a singleton type completely characterizes its type. Thus we can compute over a value of a singleton type, and the computation at the value level can express a property at the type level. By using singleton types we completely avoid the use of dependent types where types depend on values [23, 32]. The cost associated with this avoidance is the possible duplication of data structures and functions at several levels.

3.7 Pattern: A Pun: Nat'

We now define the type `Nat'`, which is identical structurally to the type `SNat`. As such, the type `Nat'` is also a singleton type representing the natural numbers, but it relies on a feature of the Ωmega type system. In Ωmega (as in Haskell) the name space for values is separate from the name space for types. Thus it is possible to have the same name stand for two things. One in the value space, and the other in the type space. The pun is because we use the names `S` and `Z` in both the value and type name spaces. We exploit this ability by writing:

```
data Nat':: Nat ~> *0 where
  Z:: Nat' Z
  S:: Nat' n -> Nat' (S n)
```

The value constructors `Z:: Nat' Z` and `S:: Nat' n -> Nat' (S n)` are ordinary values whose types mention the type constructors they pun. The name space partition, and the relationship between `Nat` and `Nat'` is illustrated below.

⟵ *value name space*	*type name space* ⟶		
value	*type*	*kind*	*sort*
	Z	:: Nat	:: *1
	S	:: Nat ~> Nat	:: *1
Z	:: Nat' Z	:: *0	:: *1
S	:: Nat' m -> Nat' (S m)	:: *0	:: *1

In `Nat'`, the singleton relationship between a `Nat'` value and its type is emphasized even more strongly, as witnessed by the example `three'`.

```
three' = (S(S(S Z))):: Nat'(S(S(S Z)))
```

Here the shape of the value, and the type index appear isomorphic. We further exploit this pun, by extending the syntactic sugar for writing natural numbers at the type level (`0t`, `1t`, etc.) to their singleton types at the value level. Thus we may write `(2t:: Nat' 2v)`. See Section 3.13 for details.

Exercise 13. Write the two *Ω*mega functions with types: `same:: Nat' n -> LE n n`, and `predLE:: Nat' n -> LE n (S n)`. Hint: they are simple recursive functions.

Exercise 14. Write the *Ω*mega function which witnesses the implication stating the transitivity of the less-than-or-equal-to predicate. `trans:: LE a b -> LE b c -> LE a c`. By the curry-Howard isomorphism a total function between two witnesses

Hint: it is a recursive function with pattern matching over both arguments. One of the cases is not reachable. Which one? Why? See Section 3.10 for how you can document this fact.

Exercise 15. In Exercise 11 we proposed writing a function with type `(Plus a b c -> LE b c)`. This turned out to be not possible given our current knowledge. But, it is possible to write a function with type `(Nat' b -> Plus a b c -> LE b c)`. Write this function. What benefit does the first `Nat' b` argument provide? Hint: both the functions `same` and `predLE` come in useful.

3.8 Pattern: Leibniz Equality

Terminating terms of type `(Equal lhs rhs)` are values witnessing the equality of `lhs` and `rhs`. The type constructor `Equal` is defined as:

```
data Equal :: a ~> a ~> *0 where
  Eq:: Equal x x
```

The type constructor `Equal` can be applied to any two types, as long as both are classified by the same classifier `a`. The classifier `a` is largely unconstrained. In Section 3.12 we discuss this in greater depth. Intuitively, given a term `w` with type `(Equal x y)`, we can think of `w` as a proof that `x` and `y` are equal.

Note that `Equal` is a GADT, since in the type of the constructor `Eq` the two type indexes are the same, and not just polymorphic variables (i.e. the type of `Eq` is *not* `(Equal x y)` but is rather `(Equal x x)`). The single constructor (`Eq`) has a polymorphic type `(Equal x x)`. Ordinarily, if the two arguments of `Equal` are type-constructor terms, the two arguments must be the same (or they couldn't be equal). But, if we allow type functions as arguments (see Section 3.2), since many functions may compute the same result (even with different arguments), the two terms can be syntactically different (but semantically the same). For example `(Equal 2t {plus 1t 1t})` is a well formed equality type since 2 is semantically equal to 1+1. The `Equal` type allows the programmer to reify the type checkers notion of equality, and to pass this reified evidence around as a value. The `Equal` type plays a large role in the `theorem` declaration (see Section 6).

Exercise 16. Singleton types allow us to construct Equal objects at runtime. Because of the one-to-one relationship between singleton values and their types, knowing the shape of a value determines its type. In a similar manner knowing the type of a singleton determines its shape. Write the function in Ωmega that exploits this fact: `sameNat:: Nat' a -> Nat' b -> Maybe(Equal a b)`. We have written the first clause. You can finish it.

```
sameNat:: Nat' a -> Nat' b -> Maybe(Equal a b)
sameNat Z Z = Just Eq
```

If one wonders how this function is typed, it is very instructive to construct the typing box (as we did for `app` in Section 2) with expected types, equations, computed types, and generated equalities.

3.9 Computing Programs and Properties Simultaneously

We can write programs that compute an indexed value along with a witness that the value has some additional property. For example, when we add two static length lists, the resulting list has a length that is related to the lengths of the two input lists, and we can simultaneously produce a witness to this relationship.

```
data Plus:: Nat ~> Nat ~> Nat ~> *0 where
  PlusZ:: Plus Z m m
  PlusS:: Plus n m z -> Plus (S n) m (S z)

app1:: Seq a n -> Seq a m -> exists p . (Seq a p,Plus n m p)
app1 Snil ys = Ex(ys,PlusZ)
app1 (Scons x xs) ys = case (app1 xs ys) of
                        Ex(zs,p) ->  Ex(Scons x zs,PlusS p)
```

The keyword `Ex` is the "pack" operator of Cardelli and Wegner [6]. Its use turns a normal type `(Seq a p,Plus n m p)` into an existential type `exists p.(Seq a p,Plus n m p)`. The Ωmega compiler uses a bidirectional type checking algorithm to propagate the existential type in the signature inwards to the `Ex` tagged expressions. This allows it to abstract over the correct existentially quantified variables.

In a similar manner, given a proof that $a \leq b$ we can always find a c such that $a + c = b$.

```
smaller :: Proof {le (S a) (S b)} -> Proof {le a b}
smaller Triv = Triv

diff:: Proof {le a b} -> Nat' a -> Nat' b ->
       exists c .(Nat' c,Equal {plus a c} b)
diff Triv Z m = Ex (m,Eq)
diff Triv (S m) Z = unreachable
diff (q@Triv) (S x) (S y) =
  case diff (smaller q) x y of
   Ex (m,Eq) -> Ex (m,Eq)
```

Exercise 17. The filter function drops some elements from a list. Thus, the length of the resulting list cannot be known statically. But, we can compute the length of the resulting list along with the list. Write the Ωmega function with prototype:

```
filter :: (a->Bool) -> Seq a n -> exists m . (Nat' m,Seq a m)
```

Since filter never adds elements to the list, that weren't already in the list, the result-list is never longer than the original list. We can compute a proof of this fact as well. Write the *Ω*mega function with prototype:

```
filter :: (a->Bool) -> Seq a n -> exists m . (LE m n,Nat' m,Seq a m)
```

Hint: You may find the functions `predLE` from Exercise 13 useful.

3.10 Feature: Unreachable Clauses

The keyword `unreachable` in the second clause of the definition for `diff` states that type considerations preclude the flow of control ever reaching the clause labeled unreachable. This is because the type information in the function prototype for `diff` is propagated into the patterns of each clause. In the second clause the following information is propagated.

```
Triv  :: Proof {le a b}
(S m) :: Nat' a
Z     :: Nat' b
```

We compute the type of `(S m)` to be `(Nat' (S m))`, and we compute the type of `Z` to be `(Nat' Z)`, combining this with the propagated type information we see that `a = (S m)` and `b = Z`. Thus the type of `Triv` must be `Proof {le (S m) Z}`. The type function application `{le (S m) Z}` reduce to `F`, but the argument to `Proof` must be `T`. These sets of assumptions are inconsistent. So the clause in the scope of these patterns is unreachable. There are no well-typed arguments, to which we could apply `diff`, that would exercise the second clause. The keyword `unreachable` indicates to the compiler that we recognize this fact. The reachability of all unreachable clauses is tested. If they are in fact reachable, an error is raised. An unreachable clause, without the `unreachable` keyword also raises an error.

The point of the unreachable clause is to document that the author of the code knows that this clause is unreachable, and to help document that the clauses exhaustively cover all possible cases. The function `extract` from exercise 7 and the function `trans` from exercise 14 could use an unreachable clause.

3.11 Feature: Staging

*Ω*mega supports staging annotations: brackets (`[| _ |]`), escape (`$( _ )`), and the two staging functions `lift::(forall a . a -> Code a)` and `run::(forall a . (Code a) -> a)` for building and manipulating code. *Ω*mega uses the Template Haskell [25] conventions for creating code. Brackets (`[| _ |]`) are a quasi-quotation mechanism, and escape (`$( _ )`) escapes from the effects of quasi-quotation. For example.

```
inc x = x + 1
c1a = [| 4 + 3 |]
c2a = [| \ x -> x + $c1a |]
```

```
c3 = [| let f x = y - 1 where y = 3 * x in f 4 + 3 |]
c4 = [| inc 3 |]
c5 = [| [| 3 |] |]
c6 = [| \ x -> x |]
```

In the examples above, `inc` is a normal function. The variable `c1a` names a piece of code with type `Code Int`. The variable `c2a` names a piece of code with type `Code(Int -> Int)`. It is constructed by splicing the code `c1a` into the body of the lambda abstraction. The variable `c3` names a piece of code with type `Code Int`. It illustrates the ability to define rich pieces of code with embedded `let` and `where` clauses. The variable `c4` names a piece of code with type `Code Int`. It illustrates that functions defined in earlier stages (`inc`) can be lifted (or embedded) in code. The variable `c5` names a piece of code with type `Code (Code Int)`. It illustrates that code can be nested.

The purpose of the staging mechanism is to have finer control over evaluation order, which is exactly what we want to do when removing the interpretive overhead of generic programming. Ωmega supports many of the features of MetaML [24, 36].

Exercise 18. The traditional staged function is the power function. The term `(power 3 x)` returns x to the third power. The unstaged power function can be written as:

```
power:: Int -> Int -> Int
power 0 x = 1
power n x = x * power (n-1) x
```

Write a staged power function: `pow:: Int -> Code Int -> Code Int` such that `(pow 3 [|99|])` evaluates to `[| 99 * 99 * 99 * 1 |]`. This can be written simply by placing staging annotations in the unstaged version.

3.12 Feature: Level Polymorphism

Sometimes we wish to use the same structure at both the value and type level. One way to do this is to build isomorphic, but different, data structures at different levels. In Ωmega, we can define a structure to live at many levels. We call this level polymorphism. For example a `Tree` type that lives at all levels can be defined by:

```
data Tree :: level n . *n ~> *n where
  Tip :: a ~> Tree a
  Fork :: Tree a ~> Tree a ~> Tree a
```

Levels are *not* types. A level variable can only be used as an argument to the `*` operator. Level abstraction can only be introduced in the kind part of a `data` declaration, but level polymorphic functions can be inferred from their use of constructor functions introduced in level polymorphic `data` declarations.

In the example above, Ωmega adds the type constructor `Tree` at all type levels, and the constructors `Tip` and `Fork` at the value level as well at all type levels. We can illustrate this by evaluating a tree at the value level, and by asking Ωmega for the kind of a similar term at the type level.

```
prompt> Fork (Tip 3) (Tip 1)
(Fork (Tip 3) (Tip 1)) : Tree Int

prompt> :k Tip Int
Tip Int :: Tree *0
```

Another useful pattern is to define normal (`*0`) datatypes indexable by types at all levels. For example consider the kind of the type constructor `Equal` and the type of its constructor `Eq` from Section 3.8. Its type can be more verbosely expressed as follows where the level polymorphism is explicit (rather than inferred, as it is in Section 3.8).

```
Equal :: level b . forall (a:*(1+b)).a ~> a ~> *0

Eq :: level b . forall (a:*(1+b)) (c:a:*(1+b)).Equal c c
```

For all levels `b`, the type `a` is classified by a star at level `1+b`. Some legal instances are:

```
Equal :: forall (a:*1).a ~> a ~> *0   -- when b=0
Equal :: forall (a:*2).a ~> a ~> *0   -- when b=1
```

Without level polymorphism, the `Equal` type constructor could only witness equality between types at a single level, i.e. types classified by `a:: *1` but not `a:: *2`. So `(Equal Int Bool)` is well formed but `(Equal Nat Tag)` would not be, since both `Nat` and `Tag`[2] are classified by `*1:: *2`. For a useful example, the type of `labelEq` could not be expressed using a level-monomorphic `Equal` datatype.

```
labelEq:: forall (a:Tag) (b:Tag). Label a -> Label b -> Maybe (Equal a b)
```

This is because the `a` and `b` are classified by `Tag`, and are not classified by `*0`.

Exercise 19. A row is a list-like structure that associates a pair of objects. In *Ω*mega we write `{`a=Int,`z=Bool}r` for the row classified by `(Row Tag *0)`, which associates the `Tag `a` with `Int`, and `` `z`` with `Bool`. In general we'd like not to restrict rows to any single level. Level polymorphism comes in handy here. Define a GADT, `MyRow`, that defines a level polymorphic row type at level 1, but which is indexed by a pair of types from any level. I.e. `MyRow` should be classified as follows:

```
MyRow :: level d b . forall (a:*(2+b)) (c:*(2+d)). a ~> c ~> *1
```

3.13 Feature: Syntactic Extension

Many languages supply syntactic sugar for constructing homogeneous sequences and heterogeneous tuples. For example in Haskell lists are often written with bracketed syntax, `[1,2,3]`, rather than a constructor function syntax,

[2] See Section 3.14.

`(Cons 1 (Cons 2 (Cons 3 Nil)))`, and tuples are often written as `(5,"abc")` and `(2,True,[])` rather than `(Pair 5 "abc")` and `(Triple 2 True [])`. In Ωmega we supply special syntax for four different kinds of data, and allow users to use this syntax for data they define themselves. Ωmega has special syntax for list-like, natural-number-like, pair-like, and record-like types. Some examples in the supported syntax are: `[4,5]i`, `(2+n)j`, `(4,True)k`, and `{"a"=5, "b"=6}h`. In general, the syntax starts with list-like, natural-number-like, record-like, or pair-like syntax, and is terminated by a tag. A programmer may specify that a user defined type should be displayed using the special syntax with a given tag. Each tag is associated with a set of functions (a different set for list-like, natural-number-like, record-like, and pair-like types). Each term written using the special syntax (with tag *i*) expands into a call of the functions specified by tag *i*. For example 2*i* expands to `S(S Z))` if the functions associated with *i* are `S` and `Z`. We now explian the details for each case.

The list-like syntax associates two functions with each tag. These functions play the role of `Nil` and `Cons`. For example if the tag "`i`" is associated with the functions `(C,N)`, then the expansion is as follows.

```
[]i          ---> N
[x,y,z]i     ---> C x(C y (C z N))
[x;xs]i      ---> (C x xs)
[x,y ; zs]i ---> C x (C y zs)
```

The semicolon may only appear before the last element in the square brackets. In this case, the last element stands for the tail of the resulting list.

The natural-number-like syntax associates two functions with each tag. These functions play the role of `Zero` and `Succ`. For example if the tag "`i`" is associated with the functions `(Z,S)`, then the expansion is as follows.

```
4i     ---> S(S(S(S Z)))
0i     ---> Z
(2+x)i ---> S(S x)
```

In earlier versions of Ωmega, before the addition of syntactic extensions, values of the built in types `Nat` and `Nat'`, could be specified using the syntax `#4`. For backward compatibility reasons, this is currently still supported and is equivalent to either `4t` (i.e. `S(S(S(S Z)))`) in the type name space, and `4v` (i.e. `S(S(S(S Z)))`) in the value name space.

The tuple-like syntax associates one function with each tag. This function plays the role of a binary constructor. For example if the tag "`i`" is associated with the function `P`, then the expansion is as follows.

```
(a,b)i       ---> P a b
(a,b,c)i     ---> P a (P b c)
(a,b,c,d)i   ---> P a (P b (P c d))
```

The record-like syntax associates two functions with each tag. These functions play the role of the constant `RowNil` and the ternary function `RowCons`. For example, if the tag "`i`" is associated with the functions `(RN,RC)`, then the expansion is as follows.

```
{}i             ---> RN
{a=x,b=y}i     ---> RC a x (RC b y RN)
{a=x;xs}i      ---> (RC a x xs)
{a=x,b=y ; zs}i ---> RC a x (RC b y zs)
```

Syntactic extension can be applied to any GADT, at either the value or type level. The new syntax can be used by the programmer for terms, types, or patterns. Ωmega uses the new syntax to display such terms. The constructor based mechanism can also still be used. The tags are specified using a deriving clause in a GADT. See Section 5.9 for an example use of this feature that makes Ωmega code easy to read and understand.

Exercise 20. Consider the GADT with syntactic extension "`i`".

```
data Nsum:: *0 ~> *0 where
    SumZ:: Nsum Int
    SumS:: Nsum x -> Nsum (Int -> x)
  deriving Nat(i)
```

What is the type of the terms `0i`, `1i`, and `2i`? Can you write a function with prototype: `add:: Nsum i -> i`, where `(add n)` is a function that sums n integers. For example: `add 3i 1 2 3` $\longrightarrow$ `6`.

3.14 Feature: Tags and Labels

Many object languages have a notion of name. To make representing names in the type system easy we introduce the notion of Tags and Labels. As a *first approximation*, consider the finite kind `Tag` and its singleton type `Label`:

```
data Tag:: *1 where
  A:: Tag
  B:: Tag
  C:: Tag

data Label:: Tag ~> *0 where
  A:: Label A
  B:: Label B
  C:: Label C
```

Here, we again deliberately use the value-name space, type-name space overloading. The names `A`, `B`, and `C` name different, but related, objects at both the value and type level. At the value level, every `Label` has a type index that reflects its value. I.e. `A::Label A`, and `B::Label B`, and `C::Label C`. Now consider a countably infinite set of tags and labels. We can't define this explicitly, but we can build such a type as a primitive inside of Ωmega. At the type level, every legal identifier whose name is preceded by a back-tick (`) is a type classified by the kind `Tag`. For example the type `` `abc `` is classified by `Tag`. At the value level, every such symbol `` `abc `` is classified by the type ``(Label `abc)``.

There are several functions that operate on labels. The first is `labelEq` which compares two labels for equality. Since labels are singletons, a simple true or false answer would be useless. Instead `labelEq` returns a Leibniz proof of equality (see Section 3.8) that the `Tag` indexes of identical labels are themselves equal.

```
labelEq :: forall (a:Tag) (b:Tag).Label a -> Label b -> Maybe (Equal a b)

prompt> labelEq ‘w ‘w
(Just Eq) : Maybe (Equal ‘w ‘w)

prompt> labelEq ‘w ‘s
Nothing : Maybe (Equal ‘w ‘s)
```

Fresh labels can be generated by the function `freshLabel`. Since the `Tag` index for such a label is unknown, the generator must return a structure where the `Tag` indexing the label is existentially quantified. Since every call to `freshLabel` generates a different label, the `freshLabel` operation must be an action in the `IO` monad. The function `newLabel` coerces a string into a label. It too, must existentially hide the Tag indexing the returned label. But, because it always returns the same label when given the same input it can be a pure function.

```
freshLabel :: IO HiddenLabel
newLabel:: String -> HiddenLabel

data HiddenLabel :: *0 where
 Hidden:: Label t -> HiddenLabel
```

We illustrate this at the top-level loop. The Ωmega top-level loop executes `IO` actions, and evaluates and prints out the value of expressions with other types.

```
prompt> freshLabel
Executing IO action                -- An IO action
(Hidden ‘#cbp) : IO HiddenLabel


prompt> temp <- freshLabel         -- An IO action
Executing IO action
(Hidden ‘#sbq) : HiddenLabel
prompt> temp
(Hidden ‘#sbq) : HiddenLabel

prompt> newLabel "a"               -- A pure value
(Hidden ‘a) : HiddenLabel
```

Exercise 21. A common use of labels is to name variables in a data structure used to represent some object language as data. Consider the GADT and an evaluation function over that object type.

```
data Expr:: *0 where
  VarExpr :: Label t -> Expr
  PlusExpr:: Expr -> Expr -> Expr
```

```
valueOf:: Expr -> [exists t .(Label t,Int)] -> Int
valueOf (VarExpr v) env = lookup v env
valueOf (PlusExpr x y) env = valueOf x env + valueOf y env
```

Write the function: `lookup:: Label v -> [exists t .(Label t,Int)] -> Int`.

4 Maintaining Structural Invariants of Data

Both `Seq` and `Tree` use kinds as indexes (`Nat` for `Seq`, and `Shape` for `Tree`) to maintain an invariant about the shape of the data. This is quite common. In this section we illustrate this in more detail by examining the world of balanced trees.

4.1 AVL Trees

Binary search trees are a classic data structure for implementing finite maps or sets in a purely functional way. To guarantee efficient operations, we want our trees to be somewhat balanced. There are several ways to define what it means for a tree to be balanced, each leading to different data structures such as Red-Black trees, AVL trees, B-trees, etc. In this section we implement AVL trees in such a way that *Ω*mega's type system guarantees compliance with the balancing invariant.

Types Expressing Invariants. The balancing invariant for AVL trees is simple: any internal node in the tree has children whose heights differ by no more than one. In this section, we define types that express this invariant. Here is our core data structure for AVL trees (indexed by tree height).

```
data Avl :: Nat ~> *0 where
  Leaf :: Avl Z
  Node :: Balance hL hR hMax -> Avl hL -> Int -> Avl hR -> Avl (S hMax)
```

A binary tree has two constructors – one for (empty) leaves and one for internal nodes carrying data. An auxiliary type captures the balancing constraints.

```
data Balance:: Nat ~> Nat ~> Nat ~> *0 where
  Less :: Balance h     (S h) (S h)
  Same :: Balance h     h     h
  More :: Balance (S h) h     (S h)
```

Think of the type `Balance hL hR hMax` as a predicate stating (1) that `hL` and `hR` differ by at most one, and (2) that `hMax` is the maximum of `hL` and `hR`. For any given internal node, there are only three possibilities for the relative heights of its subtrees:

$$1 + \texttt{hL} = \texttt{hR} \quad \text{or} \quad \texttt{hL} = \texttt{hR} \quad \text{or} \quad \texttt{hL} = \texttt{hR} + 1$$

These three possibilities correspond to the three constructors of the datatype `Balance`. Under this interpretation of `Balance`, we see that the `h` in `(Avl h)` really does capture the height of the tree (leaves have height zero and the height of an internal node is the successor of the maximum of the heights of its children).

Finally, we would like to protect users of our library from having to deal with height indices in their own code. To this end, we define a wrapper type that hides away the height index.

```
data AVL:: *0 where
  AVL:: (Avl h) -> AVL
```

In this type the `h` is existentially quantified. This is the type that users will see.

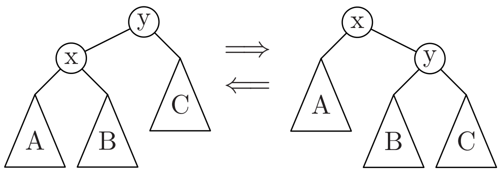
The `data` declarations are all the code we ever need write to guarantee that every AVL tree in our library is well-balanced. Because these type declarations express the balancing invariants, the problem of deciding whether our implementation respects those invariants reduces to the problem of deciding type-correctness, which the Ωmega type-checker does for us automatically.

Basic operations. The two most basic operations are constructing an empty tree and testing an element for membership in the tree.

```
empty :: AVL
empty = AVL Leaf

element :: Int -> AVL -> Bool
element x (AVL t) = elem x t

elem :: Int -> Avl h -> Bool
elem x Leaf = False
elem x (Node _ l y r)
  | x == y  = True
  | x < y   = elem x l
  | x > y   = elem x r
```

The remaining operations of insertion and deletion are much more interesting.

Balancing Constructors. The algorithms for insertion and deletion each follow the same basic pattern: First do the insertion (or deletion) as you would for any other binary search tree. Then re-balance any subtree that became unbalanced in the process. The tool used for re-balancing is tree rotation, which is best described visually.



The transformation of the tree on the left to the tree on the right is *right rotation* and the opposite transformation is called *left rotation*. This operation preserves the BST invariant. However, they do *not* preserve the balancing invariant, which is precisely why they are useful for rebalancing.

It turns out that we can package up all necessary rotations into a couple of *smart constructors*, `rotr` and `rotl`. Think of `rotr lc x rc` as a smart version of `Node ? lc x rc` where

1. We don't have to say (or know) how the resulting tree is balanced, and
2. The subtrees, `lc` and `rc`, don't quite balance out because `height(lc)` = `height(rc) + 2` and therefore we must do some rightward rebalancing rotation(s).

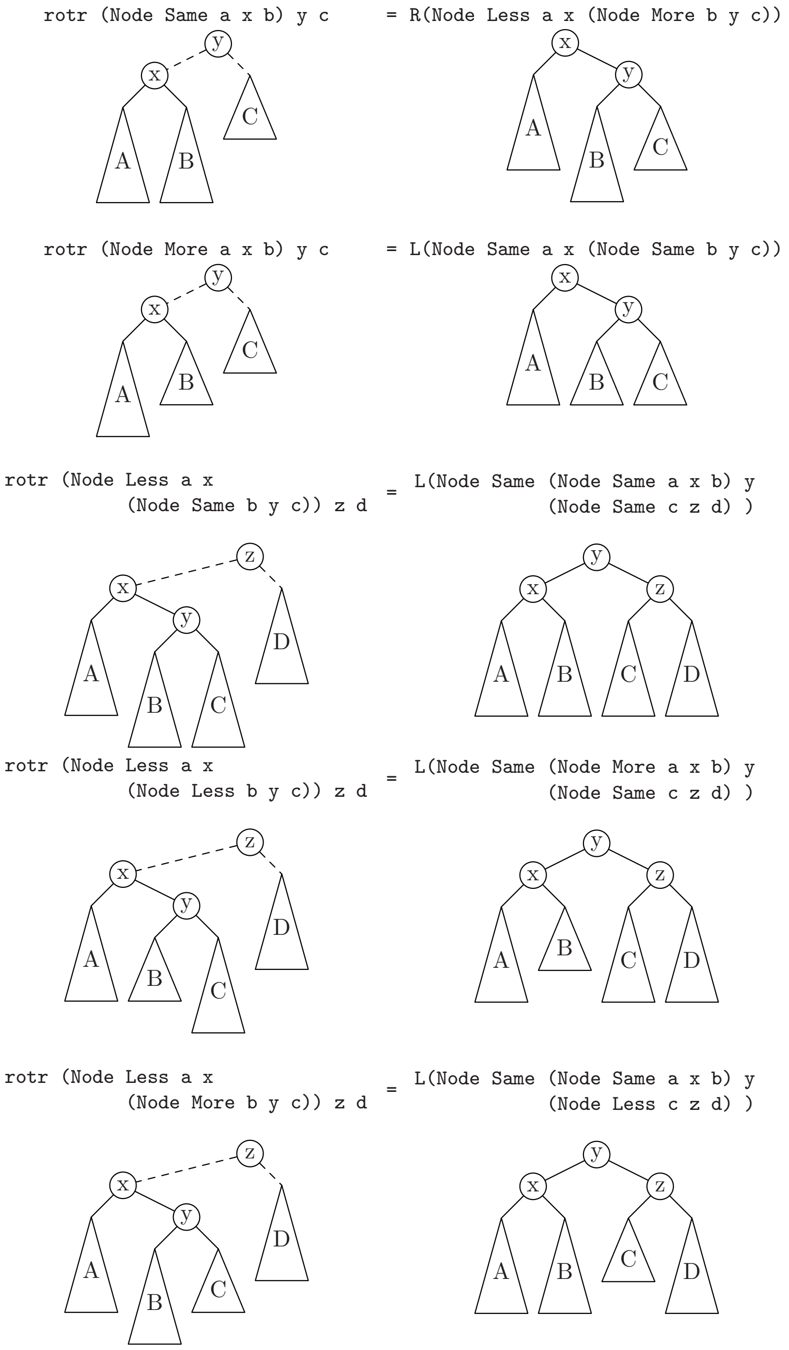
The only wrinkle in the "smart constructor" story is that the height of the resulting tree depends on what rotations were performed. However, the result height ranges over merely two values, so we just return a value of a sum type[3]. Here is the code:

```
rotr :: Avl (2+n)t -> Int -> Avl n -> ( Avl(2+n)t + Avl (3+n)t )
rotr Leaf x a = unreachable
rotr (Node Less a x Leaf) y b = unreachable
-- single rotations
rotr (Node Same a x b) y c = R(Node Less a x (Node More b y c))
rotr (Node More a x b) y c = L(Node Same a x (Node Same b y c))
-- double rotations
rotr (Node Less a x (Node Same b y c)) z d =
   L(Node Same (Node Same a x b) y (Node Same c z d))
rotr (Node Less a x (Node Less b y c)) z d =
   L(Node Same (Node More a x b) y (Node Same c z d))
rotr (Node Less a x (Node More b y c)) z d =
   L(Node Same (Node Same a x b) y (Node Less c z d))
```

Figure 2 depicts the rotation for each substantive case in the definition of `rotr`. The algorithm for `rotl` is perfectly symmetric to that for `rotr`.

```
rotl :: Avl n -> Int -> Avl (2+n)t -> ( Avl (2+n)t + Avl (3+n)t )
rotl a x Leaf = unreachable
rotl a x (Node More Leaf y b) = unreachable
-- single rotations
rotl a u (Node Same b v c) = R(Node More (Node Less a u b) v c)
rotl a u (Node Less b v c) = L(Node Same (Node Same a u b) v c)
-- double rotations
rotl a u (Node More (Node Same x m y) v c) =
   L(Node Same (Node Same a u x) m (Node Same y v c))
rotl a u (Node More (Node Less x m y) v c) =
   L(Node Same (Node More a u x) m (Node Same y v c))
rotl a u (Node More (Node More x m y) v c) =
   L(Node Same (Node Same a u x) m (Node Less y v c))
```

[3] In Ωmega the value constructors `L :: a -> (a+b)` and `R :: b -> (a+b)` are used to construct sums.



```
rotr (Node Same a x b) y c      = R(Node Less a x (Node More b y c))

rotr (Node More a x b) y c      = L(Node Same a x (Node Same b y c))

rotr (Node Less a x                 L(Node Same (Node Same a x b) y
           (Node Same b y c)) z d =                (Node Same c z d) )

rotr (Node Less a x                 L(Node Same (Node More a x b) y
           (Node Less b y c)) z d =                (Node Same c z d) )

rotr (Node Less a x                 L(Node Same (Node Same a x b) y
           (Node More b y c)) z d =                (Node Less c z d) )
```

Fig. 2. Each substantive case in the definition of `rotr`

As these functions are both self-contained and non-recursive, we see that they operate in constant time.

Insertion. When we insert an element into an AVL tree, the height of the tree either remains the same or increases by at most one. We therefore arrive at the following type for insertion:

```
ins :: Int -> Avl n -> (Avl n + Avl (S n))
```

The code for `ins` is an elaborate case analysis. The first decision to make is whether we're at the right spot for insertion. If so, then do the insertion (or not, depending on whether the value already exists in the tree), and then return. If not, make the appropriate recursive call and then rebalance. Most of the work goes into determining how to rebuild a balanced tree by choosing the correct `Balance` value or rebalancing constructor.

```
ins :: Int -> Avl n -> (Avl n + Avl (S n))
ins x Leaf = R(Node Same Leaf x Leaf)
ins x (Node bal a y b)
  | x == y = L(Node bal a y b)
  | x < y  = case ins x a of
               L a -> L(Node bal a y b)
               R a ->
                 case bal of
                   Less -> L(Node Same a y b)
                   Same -> R(Node More a y b)
                   More -> rotr a y b -- rebalance!
  | x > y  = case ins x b of
               L b -> L(Node bal a y b)
               R b ->
                 case bal of
                   Less -> rotl a y b -- rebalance!
                   Same -> R(Node Less a y b)
                   More -> L(Node Same a y b)
```

Figure 3 depicts each case in the `x < y` branch. Now we wrap this function up to work on user-level AVL trees.

```
insert :: Int -> AVL -> AVL
insert x (AVL t) = case ins x t of L t -> AVL t; R t -> AVL t
```

Deletion. Whereas insertion always places an element in the fringe of a tree, deletion may find the targeted element somewhere in the interior. For this reason, deletion is a more complex operation. The strategy for deleting the value x at an interior node is to first replace its value with that of the minimum value z of its right child (or the maximum value of its left child, depending on the policy). Then delete z (which is always at a leaf) from the right child.

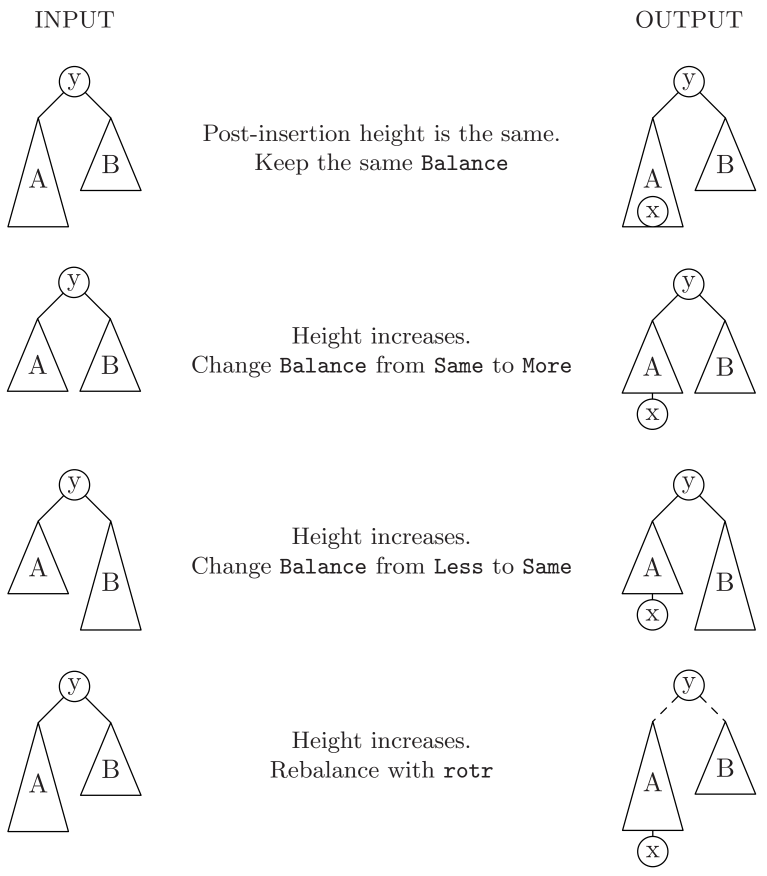


Fig. 3. Rebalancing after insertion into the left child

We will calculate the minimum value in a tree and delete it in a single pass. The operation only works on trees of height ≥ 1 (which are therefore non-empty). The returned tree might have decreased in size by one.

```
delMin :: Avl (S n) -> (Int, (Avl n + Avl (S n)))
delMin Leaf = unreachable
delMin (Node Less Leaf x r) = (x,L r)
delMin (Node Same Leaf x r) = (x,L r)
delMin (Node More Leaf x r) = unreachable
delMin (Node bal (l@(Node _ _ _ _)) x r) =
      case delMin l of
        (y,R l) -> (y,R(Node bal l x r))
        (y,L l) ->
          case bal of
            Same -> (y,R(Node Less l x r))
            More -> (y,L(Node Same l x r))
            Less -> (y,rotl l x r) -- rebalance!
```

Deletion of the minimum element requires rebalancing operations on the way up, just as in insertion.

When we delete an element from an AVL tree, the height of the tree either remains the same or decreases by at most one. We therefore arrive at the following type for deletion:

```
del :: Int -> Avl (S n) -> (Avl n + Avl (S n))
```

The code for `del` is an elaborate case analysis. The first decision to make is whether we're at the right spot for deletion. If so, then do the deletion (or not, depending on whether the value exists in the tree) and return. If not, make the appropriate recursive call and then rebalance. Most of the work goes into determining how to rebuild a balanced tree by choosing the correct `Balance` value or rebalancing constructor.

```
del :: Int -> Avl n -> (Avl n + exists m .(Equal (S m) n,Avl m))
del y Leaf = L Leaf
del y (Node bal l x r)
  | y == x = case r of
              Leaf ->
                case bal of
                  Same -> R(Ex(Eq,l))
                  More -> R(Ex(Eq,l))
                  Less -> unreachable
              Node _ _ _ _ ->
                case (bal,delMin r) of
                  (_,z,R r) -> L(Node bal l z r)
                  (Same,z,L r) -> L(Node More l z r)
                  (Less,z,L r) -> R(Ex(Eq,Node Same l z r))
                  (More,z,L r) ->
                     case rotr l z r of -- rebalance!
                       L t -> R(Ex(Eq,t))
                       R t -> L t
  | y < x = case del y l of
             L l -> L(Node bal l x r)
             R(Ex(Eq,l)) ->
               case bal of
                 Same -> L(Node Less l x r)
                 More -> R(Ex(Eq,Node Same l x r))
                 Less ->
                    case rotl l x r of -- rebalance!
                      L t -> R(Ex(Eq,t))
                      R t -> L t
  | y > x = case del y r of
             L r -> L(Node bal l x r)
             R(Ex(Eq,r)) ->
               case bal of
                 Same -> L(Node More l x r)
                 Less -> R(Ex(Eq,Node Same l x r))
                 More ->
```

```
                        case rotr l x r of -- rebalance!
                          L t -> R(Ex(Eq,t))
                          R t -> L t
```

Now we wrap this function up to work on user-level AVL trees.

```
delete :: Int -> AVL -> AVL
delete x (AVL t) = case del x t of L t -> AVL t; R t -> AVL t
```

Exercise 22. **Red Black Trees.** A red-black tree is a binary search tree with the following additional invariants:

1. Each node is colored either red or black
2. The root is black
3. The leaves are black
4. Each Red node has Black children
5. For all internal nodes, each path from that node to a descendant leaf contains the same number of black nodes.

We can encode these invariants by thinking of each internal node as having two attributes: a color and a black-height. We will use a GADT, we call `SubTree`, with two indexes, one of them a `Nat` (for the black-height) and the other a `Color`.

```
data Color:: *1 where
  Red:: Color
  Black:: Color

data SubTree:: Color ~> Nat ~> *0 where
 Leaf:: SubTree Black Z
 RNode:: SubTree Black n -> Int -> SubTree Black n -> SubTree Red n
 BNode:: SubTree cL m    -> Int -> SubTree cR m    -> SubTree Black (S m)

data RBTree:: *0 where
 Root:: SubTree Black n -> RBTree
```

Note how the black height increases only on black nodes. The type `RBTree` encodes a “full” Red-Black tree, forcing the root to be black, but placing no restriction on the black-height. Write an insertion function for Red-Black trees. A solution to this exercise is found in Appendix A.

5 Ωmega as a Meta Language

It has become common practice when designing a new language to study the relationship between a static semantics (a type system) and a dynamic semantics (a meaning function). This process is often exploratory. The designer has an idea, the approach is analyzed, and hopefully the consequences of the approach are quickly discovered. Automated aid in this process would be a great boon.

The ultimate goal of this exploratory process is a type system, a semantics, and a proof. The proof witnesses the fact that *well-typed programs do not go* *wrong* [16] for the language under consideration. The most common way to perform such a proof is by a subject reduction proof in the style of Wright and Felleisen [39] on a small step semantics, though there other approaches as well [10, 16]. Such proofs require an amazing amount of detail and are most often carried out by hand, and are thus subject to all the foils of human endeavors.

*Ω*mega is our attempt at developing a generic meta-language that could be used for exploring the static and dynamic semantics for new object-languages [19, 26] that could aid in the generation of such proofs. This section describes how *Ω*mega can be used as a meta-language. We show that:

– Much of the work of exploring the nuances of a type system for a new language can be assisted by using mechanized tools – a generic meta-language.
– Such tools need not be much more complicated than your favorite functional language (Haskell), and are thus within the reach of most language researchers.
– The automation helps language designers visualize the consequences of their design choices quickly, and thus helps speed the design process.
– The artifacts created by this exploration are machine checked proofs, and are hence less subject to error than proofs constructed by the more traditional approach.

5.1 Object Languages

In meta-programming systems meta-programs manipulate object-programs. Meta-programs may construct object-programs, combine object-program fragments into larger object-programs, observe the structure and other properties of object-programs, and execute object-programs to obtain their values.

There are several important kinds of meta-programming scenarios: program generators, and program analyses. Each of these scenarios has a number of distinguishing characteristics.

A program generator (a meta-program) solves a particular problem by constructing another program (an object-program) that solves the problem at hand. Usually the generated (object) program is "specialized" for the particular problem and uses less resources than a general purpose, non-generator solution.

A program analysis (a meta-program) observes the structure and environment of an object-program and computes some value as a result. Results can be data- or control-flow graphs, or even another object-program with properties based on the properties of the source object-program. Examples of these kinds of meta-systems are: program transformers, optimizers, and partial evaluation systems.

A language model (a meta-program) gives meaning to, and points out properties of an object-language. Examples of these include type systems, type judgments, denotational and operational semantics, and small-step semantics.

5.2 Representing Object Programs

Meta-programs must represent object-programs as data. Object program representations usually fall into one of three categories. (1) Strings, (2) Algebraic datatypes, or (3) Quasi-quote systems. Other representations (as graphs for example) are possible, but not widespread.

With the string encoding, we represent the code fragment `f(x,y)` simply as `"f(x,y)"`. While constructing and combining fragments represented by strings can be done concisely, deconstructing them is quite verbose, and in essence degenerates into a parsing problem. More seriously, there is no automatically verifiable guarantee that programs thusly constructed are syntactically correct. For example, `"f(,y)"` can have the static type `string`, but this clearly does not imply that this string represents a syntactically correct program.

5.3 Object-Programs as Algebraic Datatypes

With the Algebraic datatype encoding, we can address the syntactic correctness problem. A datatype encoding is essentially the same as what is called abstract syntax or parse trees. The encoding of the fragment `plus(x,y)` in an Ωmega datatype might be:

```
Apply Plus (Tuple [Variable "x" ,Variable "y"])
```

using a datatype declared as follows:

```
data Exp:: *0 where
  Variable:: String  -> Exp     -- x
  Constant:: Int -> Exp         -- 5
  Plus:: Exp                    -- plus
  Less:: Exp                    -- less
  Apply:: Exp -> Exp -> Exp     -- Apply Plus (x,y)
  Tuple:: [Exp] -> Exp          -- (x,y)
```

Using a datatype encoding has an immediate benefit: correct typing for the meta-program ensures correct syntax for all object-programs. Because Ωmega (like most functional languages) supports pattern matching over datatypes, deconstructing programs becomes easier than with the string representation. However, constructing programs is now more verbose because we must use the cumbersome constructors like Variable, Apply, and Tuple.

5.4 Representing Programs Using Quasi-quotes

Quasi-quotation is an attempt to represent object-programs without cumbersome constructor functions. Here the actual representation of object-code is hidden from the user by the means of a quotation mechanism. Object code is constructed by placing "quotation" annotations around normal code fragments. The quasi-quotation approach is the approach used in MetaML, Template Haskell, and the staged fragment of Ωmega.

In the staged fragment of Ωmega (Section 3.11), quasi-quotations are called staging annotations, and include Brackets `[| |]` and Escape `$`. An expression `[| e |]` is a quotation, and it builds the code representation of `e` (a data structure); `$(e)` is an anti-quotation, and splices the code obtained by evaluating `e` into the body of a surrounding bracketed expression (embedding one data structure into another). The quotation and anti-quotation mechanism abstracts the actual data-type representing code.

In a quasi-quoted system, the meta-language may now enforce the type-correctness of the object language as well as the meta-language, and avoid the problems associated with a constructor based approach. The major disadvantages of quasi-quoted systems are

– There is usually only a single object-language, and it must be built into the meta-language.
– The quasi-quote mechanism is great for constructing code, but less useful for taking code apart, especially code with binding constructs.
– The type system of the meta-language must be aware of the type system of the object language. Usually this is accomplished by making the meta-language and the object-language the same language. Heterogeneous quasi-quote systems are rare because of this.

In the remainder of this section, we eschew the quasi-quote mechanism in favor of using GADTs in an effort to address these disadvantages.

5.5 Interpreters in a Typed Meta-language

Often one would like to build an interpreter or evaluation function for an object-language. In a typed meta-language, it is necessary to define a `Value` domain, that is a labeled sum of all the possible result types of evaluating an expression. In the `Exp` type above this would include both integers and booleans (as these are the types of the ranges of the functions `Plus` and `Less`), as well as functions and tuples.

```
data Value :: *0 where
  IntV:: Int -> Value
  BoolV:: Bool -> Value
  FunV:: (Value -> Value) -> Value
  TupleV :: [Value] -> Value
```

The evaluation function is then a case analysis over the structure of terms, recursively evaluating sub-terms into values, and then combining the sub-values into answer values.

```
eval:: (String -> Value) -> Exp -> Value
eval env (Variable s) = env s
eval env (Constant n) = IntV n
eval env Plus = FunV plus
  where plus (TupleV[IntV n ,IntV m]) = IntV(n+m)
eval env Less = FunV plus
  where plus (TupleV[IntV n ,IntV m]) = BoolV(n < m)
eval env (Apply f x) =
  case eval env f of
    FunV g -> g (eval env x)
eval env (Tuple xs) = TupleV(map (eval env) xs)
```

The key observation is – there is considerable overhead in such a function. It must first interpret the structure of the expressions, and it must perform quite a bit of tagging and un-tagging by applying the `Value` constructors (`IntV`, `BoolV`, `FunV`, and `TupleV`), and deconstructing them when appropriate.

5.6 Staging an Interpreter

We may remove the interpretive overhead by using staging. Like the evaluation function, the staged evaluation function is a case analysis over the structure of terms, recursively evaluating sub-terms into code values, and then splicing the smaller code values into larger code values.

```
stagedEval:: (String -> Code Value) -> Exp -> Code Value
stagedEval env (Variable s) = env s
stagedEval env (Constant n) = lift(IntV n)
stagedEval env Plus = [| FunV plus |]
  where plus (TupleV[IntV n ,IntV m]) = IntV(n+m)
stagedEval env Less = [| FunV less |]
  where less (TupleV[IntV n ,IntV m]) = BoolV(n < m)
stagedEval env (Apply f x) =
      [| apply $(stagedEval env f) $(stagedEval env x) |]
  where apply (FunV g) x = g x
stagedEval env (Tuple xs) = [| TupleV $(mapLift (stagedEval env) xs) |]
  where mapLift f [] = lift []
        mapLift f (x:xs) = [| $(f x) : $(mapLift f xs) |]
```

We may observe the result of staging by applying `stagedEval` to an actual `Exp`.

```
exp1 = Apply Plus (Tuple [Variable "x" ,Variable "y"]) -- (+)(x,y)

ans = stagedEval f exp1
  where f "x" = lift(IntV 3)
        f "y" = lift(IntV 4)

ans = [| %apply (%FunV %plus) (%TupleV [IntV 3,IntV 4]) |] : Code Value
```

We have removed the interpretive overhead, but the tagging and untagging overhead remains. This overhead is caused by using a disjoint sum as the range of the evaluator, which is necessary in a typed meta-language. This not the only problem when using algebraic datatypes to encode object-languages in a strongly typed meta-language like Haskell. The algebraic datatype approach to encoding object-languages does not track the type correctness of the object-program. We will fix both these problems by representing object-programs using GADTs rather than Algebraic datatypes.

5.7 Typed Object-Languages Using GADTs

GADTs allow us to build datatypes indexed by another type. We can use the GADT to represent object programs (just as we use algebraic datatype to represent object programs), but we may also use the type index to represent the type of the object-language program being represented. A simple typed object-language example is:

```
data Term:: *0 ~> *0 where
  Const :: Int -> Term Int              -- 5
  Add:: Term ((Int,Int) -> Int)         -- (+)
  LT:: Term ((Int,Int) -> Bool)         -- (<)
  Ap:: Term(a -> b) -> Term a -> Term b  -- (+) (x,y)
  Pair:: Term a -> Term b -> Term(a,b)   -- (x,y)
```

Above we introduced the new type constructor `Term`, which is a representation of a simple object-language of constants, pairs, and numeric operators. `Terms` are a typed object-language representation, i.e. a data structure that represents terms in some object-language. The meta-level type of the representation, i.e. the `a` in (`Term a`), indicates the type of the object-level term. This is made possible by the flexibility of the GADT mechanism. Using typed object-level terms, it is impossible to construct ill-typed term representations, because the meta-language type system enforces this constraint.

```
ex1 :: Term Int
ex1 = Ap Add (Pair (Const 3) (Const 5))

ex2 :: Term (Int,Int)
ex2 = Pair ex1 (Const 1)
```

Attempting to construct an ill-typed object term, like `(Ap (Const 3) (Const 5))`, causes a meta-level (Ωmega) type error. Another advantage of using GADTs rather than ADTs is that it is now possible to construct a tagless[19, 34, 35] interpreter directly:

```
evalTerm :: Term a -> a
evalTerm (Const x) = x
evalTerm Add = \ (x,y) -> x+y
evalTerm LT = \ (x,y) -> x<y
evalTerm (Ap f x) = evalTerm f (evalTerm x)
evalTerm (Pair x y) = (evalTerm x,evalTerm y)
```

In a language without GADTs, as we illustrated in Section 5.7, we would need to employ universal value domain like `Value`. See [18] for a detailed discussion of this phenomena. Such a tagless interpreter has the structure of a large step (or operational) semantics. If the `eval` function is total and well-typed at the meta-level, it implies that the object-level semantics (defined by `eval`) is also well-typed. Every well-typed object level term evaluates to a well-formed value.

Exercise 23. In the object-languages we have seen so far, there are no variables. One way to add variables to a typed object language is to add a variable constructor tagged by a name and a type. A singleton type representing all the possible types of a program term is necessary. For example we may add a `Var` constructor as follows (where the `Rep` is similar to the `Rep` type from Exercise 9).

```
data Term:: *0 ~> *0 where
  Var:: String -> Rep t -> Term t        -- x
  Const :: Int -> Term Int               -- 5
  . . .
```

Write a GADT for `Rep`. Now the evaluation function for `Term` needs an environment that can store many different types. One possibility is use existentially quantified types in the environment as we did in Exercise 21. Something like:

```
type Env = [exists t . (String,Rep t,t)]

eval:: Term t -> Env -> t
```

Write the evaluation function for the `Term` type extended with variables. You will need a function akin to `sameNat` from Exercise 16, except it will have prototype: `sameRep:: Rep a -> Rep b -> Maybe(Equal a b)`

Exercise 24. Another way to add variables to a typed object language is to reflect the name and type of variables in the meta-level types of the terms in which they occur. Consider the GADTs:

```
data VNum:: Tag ~> *0 ~> Row Tag *0 ~> *0 where
  Zv:: VNum l t (RCons l t row)
  Sv:: VNum l t (RCons a b row) -> VNum l t (RCons x y (RCons a b row))
 deriving Nat(u)

data Exp2:: Row Tag *0 ~> *0 ~> *0 where
  Var:: Label v -> VNum v t e -> Exp2 e t
  Less:: Exp2 e Int -> Exp2 e Int -> Exp2 e Bool
  Add:: Exp2 e Int -> Exp2 e Int -> Exp2 e Int
  If:: Exp2 e Bool -> Exp2 e t -> Exp2 e t -> Exp2 e t
```

What are the types of the terms `(Var 'x 0u)`, `(Var 'x 1u)`, and `(Var 'x 2u)`. Now the evaluation function for `Exp2` needs an environment that stores both integers and booleans. Write a datatype declaration for the environment, and then write the evaluation function. One way to approach this is to use existentially quantified types in the environment as we did in Exercises 21 and 23. Better mechanisms exist. Can you think of one?

5.8 Tagless Staged Interpreters

By staging an object-level type indexed GADT we can remove both the interpretive and tagging overhead.

```
stagedEvalTerm :: Term a -> Code a
stagedEvalTerm (Const x) = lift x
stagedEvalTerm Add = [| add |]
  where add (x,y) = x+y
stagedEvalTerm LT = [| less |]
  where less (x,y) = x < y
stagedEvalTerm (Ap f x) = [| $(stagedEvalTerm f) $(stagedEvalTerm x) |]
stagedEvalTerm (Pair x y) = [|($(stagedEvalTerm x),$(stagedEvalTerm y))|]

ex2 = (Pair (Ap Add (Pair (Const 3) (Const 5))) (Const 1))
```

We can stage a program like `ex2` by applying `stagedEvalTerm` to produce some code. For `ex2` we get: `[| (add (3, 5), 1) |]`. Note that both the interpretive overhead, and the tagging overhead, have been completely removed.

Exercise 25. A staged evaluator is a simple compiler. Many compilers have an optimization phase. Consider the term language with variables from Exercise 23.

```
data Term:: *0 ~> *0 where
  Var:: String -> Rep t -> Term t
  Const :: Int -> Term Int               -- 5
  Add:: Term ((Int,Int) -> Int)          -- (+)
  LT:: Term ((Int,Int) -> Bool)          -- (<)
  Ap:: Term(a -> b) -> Term a -> Term b  -- (+) (x,y)
  Pair:: Term a -> Term b -> Term(a,b)   -- (x,y)
```

Can you write a well-typed staged evaluator the performs optimizations like constant folding, and applies laws like $(x + 0) = x$ before generating code?

5.9 A Typed-Object Language with Binding

Object languages with variables and binding structures are harder to represent in a way that reflects the type of the object-language term in the type of its meta-language representation.

This is because if we change the type of the object-level variables, the type of the whole object-level term may also change. The key to this dilemma is to represent the type of the free variables in a term, as well as the type of the term, in the type of its meta-level representation. We do this by indexing terms by two indexes: first, the terms object-level type, and second, a type level structure encoding the environment (i.e. a mapping from variables to their types) in which the term has that type.

If we represent variables by labels (see Section 3.14), we can represent the environment by a row. A `Row` is nothing more than a list-like structure (storing pairs of elements at each “cons” node) at the type level (see Exercise 19).

```
data Row :: a ~> b ~> *1 where
   RNil :: Row x y
   RCons :: x ~> y ~> Row x y ~> Row x y
 deriving Record(r)
```

For example the type: `(RCons 3t Int RNil)` is classified by `(Row Nat *0)`. Note that we have defined a syntactic extension for rows tagged by `r`. Thus `(RCons 3t Int RNil)` will display as `{3t=Int}r`. An environment is just a type classified by `(Row Tag *0)`. We define a new value level type, `Lam`, indexed by environments (represented by `(Row Tag *0)`) and types (represented by *0).

```
data Lam:: Row Tag *0 ~> *0 ~> *0  where
  Var   :: Label s -> Lam (RCons s t env) t
  Shift :: Lam env t -> Lam (RCons s q env) t
  Abs   :: Label a -> Lam (RCons a s env) t -> Lam env (s -> t)
  App   :: Lam env (s -> t) -> Lam env s -> Lam env t
```

The first index to `Lam`, is a Row tracking its variables, and the second index, tracks the object-level type of the term. For example a term with variables `x` and `y` might have type `Lam {‘x:Int, ‘y:Bool; u}r Int`.

The key to this approach is the typing of the constructor functions for variables (`Var`) and lambda expressions (`Abs`). Consider the `Var` constructor function. To construct a variable we simply apply `Var` to a label, and its type reflects this. For example here is the output from a short interactive session with the Ωmega interpreter.

```
prompt> Var ‘name
(Var ‘name) : forall a (b:Row Tag *0).Lam {‘name=a; b}r a

prompt> Var ‘age
(Var ‘age) : forall a (b:Row Tag *0).Lam {‘age=a; b}r a
```

Variables are really De Bruijn-like in their behavior. Variables created with `Var` all have index level 0. The two examples have different names in the same index position, and they would clash if they were both used in the same lambda term. To shift the position of variable to a different index, we use the constructor `Shift:: Lam a b -> Lam {c=d; a}r b` (see Exercise 23 for an alternative mechanism to distinguish variables). To define two variables `x` and `y` for use in the same environment we shift one of them into a different index. We type a few examples at the Ωmega top-level loop to illustrate the this.

```
prompt> Var ‘x
(Var ‘x) : Lam {‘x=a; b}r a

prompt> Shift(Var ‘y)
(Shift (Var ‘y)) : Lam {a=b,‘y=c; d}r c

prompt> Shift (Shift (Var ‘z))
(Shift (Shift (Var ‘z))) : Lam {a=b,c=d,‘z=e; f}r e
```

A `Lam` term represented by `(Var ‘x)` has the tag `‘x` appearing as the first element in the environment row. By applying `Shift` once, the tag `‘x` is pushed into the second element of the row, a second `Shift` pushes it into the third element, etc.

The Abs constructor binds the first tag in the first element of the row, removing the tag and its associated type from the environment, and shifting the others towards the front of the environment.

```
prompt> App (Var ‘a) (Shift (Var ‘b))
(App (Var ‘a) (Shift (Var ‘b))) : Lam {‘a=a -> b,‘b=a; c}r b

prompt> Abs ‘f (Abs ‘x (App (Shift (Var ‘f)) (Var ‘x)))
(Abs ‘f (Abs ‘x (App (Shift (Var ‘f)) (Var ‘x))))
    : Lam a ((b -> c) -> b -> c)
```

Note how terms with free variables have non-trivial environment indexes which mention their free variables. For example the first term's type is indexed by the Row: {‘a=a -> b,‘b=a; c}r indicating that both ‘a and ‘b are free variables in the term. To build an evaluator for an object-level typed term (Section 5.10), we will need a data structure, pairing variables with their values, for each free variable in the term. We can package up a set of these values using a record.

A Record structure is a labeled tuple. We use the labels to name the variables. A Record is a level 0 value. Its type is indexed by the level 1 type Row. We can define this data structures as follows.

```
data Record :: Row Tag *0 ~> *0 where
    RecNil :: Record RNil
    RecCons :: Label a -> b -> Record r -> Record (RCons a b r)
  deriving Record()
```

Note that we have defined a syntactic extension for records tagged by the empty tag. Thus we may use the record syntax (with no tag) to build records.

```
prompt> {‘a=34,‘b="abc"}
{‘a=34,‘b="abc"} : Record {‘a=Int,‘b=[Char]}r
```

5.10 A Tagless Interpreter for a Language with Variables

The typed-object language Lam can be supplied with a typed evaluation function. The key is to supply a record that supplies exactly the values necessary for the free variables in the term being evaluated. The type system ensures that the record and the free variables coincide.

```
evalLam:: Record r -> Lam r t -> t
evalLam (RecCons _ v r) (Var _)   = v
evalLam RecNil          (Var _)   = unreachable
evalLam (RecCons _ _ r) (Shift e) = evalLam r e
evalLam RecNil          (Shift _) = unreachable
evalLam env     (Abs lab body) = \ x -> evalLam (RecCons lab x env) body
evalLam env     (App f x)      = (evalLam env f) (evalLam env x)
```

Exercise 26. Instead of using Var and Shift, fold the ideas from Exercise 24 into the Lam datatype, and then write the evaluation function for this GADT.

5.11 A Staged Interpreter for a Language with Variables

It is even possible to stage such an interpreter. One complication is that the record encoding the environment will not pair variables with values, but instead it will pair variables with code. To enable this we define the staged record.

```
data StaticRecord:: Row Tag *0 ~> *0 where
  StNil :: StaticRecord RNil
  StCons:: Label t -> Code x -> StaticRecord r -> StaticRecord (RCons t x r)

stageLam:: StaticRecord r -> Lam r t -> Code t
stageLam (StCons _ code r) (Var _)        = code
stageLam StNil             (Var _)        = unreachable
stageLam (StCons _ _ r)    (Shift e)      = stageLam r e
stageLam StNil             (Shift _)      = unreachable
stageLam env               (App f x)      =
   [| $(stageLam env f) $(stageLam env x) |]
stageLam env               (Abs lab body) =
   [| \ x -> $(stageLam (StCons lab [|x|] env) body) |]
```

5.12 Small Step Semantics

The datatype declarations for representing well-typed terms in the previous sections bear a striking similarity to the typing judgments for those languages. For example consider:

$$\frac{}{\Gamma, x{:}\tau \vdash x : \tau}\textsc{Var} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma, x{:}\sigma \vdash e : \tau}\textsc{Shift} \qquad \frac{\Gamma, x{:}\tau \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \tau \to \sigma}\textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \sigma \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \sigma}\textsc{Abs}$$

The similarity justifies a slight change in perspective. We have been thinking of `Lam` as representing a piece of abstract syntax, but we may also think of it as representing a typing derivation.

The latter perspective supports an interesting approach to studying the meta-theory of object languages. A typing derivation is a *proof* that a given term has a given type in a given context. So total functions that transform proofs into other proofs can be considered as constructive proofs of results in the meta-theory of our object language. This is the approach taken in Twelf, where the meta-language is a Prolog-style logic language. In Ωmega, we can write our meta-programs in a functional programming style.

In the remainder of this section, we build, in several steps, a proof of type soundness for our little language. Our proof has the following basic structure (due to Wright and Felleisen)[39].

1. All terms are categorized syntactically as either values or non-values.
2. A reduction relation $e \to e'$ comprising a small-step operational semantics is given.

3. Any non-value term that cannot be reduced any further is considered to exhibit a run-time error.
4. **Progress.** Any well-typed term e is either a value or can step to another well-typed term e' (that is, $e \rightarrow e'$).
5. **Preservation.** The reduction relation preserves types: If e has type τ and $e \rightarrow e'$, then e' has type τ.
6. Therefore if a term is well-typed, and we reduce it until no more reduction steps are possible, then the resulting term must be a value (rather than a term exhibiting a run-time error).

To begin, we slightly modify our `Lam` datatype from Section 5.9. We call the datatype `E` (for *E*xpression), and change the constructor names, to avoid confusion between the two. The substantive changes include the addition of a new type index (`Mode` explained in greater detail below), and a shift from usings types of kind `*0`, as indexes indicating the type of a term, to types of kind `ObjType` (also explained in greater detail below). To highlight these changes, we have included the classification of the old type `Lam` for comparison.

We emphasize, as explained earlier, that the datatype `E` can be thought of as both abstract syntax or a typing derivation.

```
--   Lam::          Row Tag *0      ~> *0      ~> *0

data E  :: Mode ~> Row Tag ObjType ~> ObjType ~> *0  where
  Const:: Rel a b -> b -> E Val env a
  Var  :: Label s -> E Val (RCons s t env) t
  Shift:: E m env t -> E m (RCons s q env) t
  Lam  :: Label a -> E m (RCons a s env) t -> E Val env (ArrT s t)
  App  :: E m1 env (ArrT s t) -> E m2 env s -> E Exp env t
```

Values versus computations. The first step to proving type-soundness in *Ω*mega by this method is to distinguish between values and non-values. We accomplish this by the introduction of the new index `Mode`.

```
data Mode:: *1 where
 Exp:: Mode
 Val:: Mode
```

Go back and study how the `Mode` index is used in the types of the constructor functions of `E`. Note how terms in normal form have types where `Val` is the first index, and ones with redexes have types where `Exp` is the first index. Consider the short *Ω*mega session:

```
prompt> Const IntR 3
(Const IntR 3) : E Val a IntT

prompt> Lam ‘x (Var ‘x)
(Lam ‘x (Var ‘x)) : E Val a (ArrT b b)

prompt> App (Lam ‘x (Var ‘x)) (Const IntR 3)
(App (Lam ‘x (Var ‘x)) (Const IntR 3)) : E Exp a IntT
```

Object-types versus meta-types. In E, we no longer use types of kind *0 as object-level types. We do this because we wish to lift some, but not all, meta-level values into constants in the object-language. In this example we wish to lift integer constants, and n-ary functions over integers (the so-called δ-reductions). To accomplish this we define a new kind to represent object-level types.

```
data ObjType:: *1 where
 ArrT:: ObjType ~> ObjType ~> ObjType
 IntT:: ObjType
```

This new kind appears as the third index of E, and also as an index to the Row comprising the environment. The constructor Const lifts only those values classified by types that are related to some ObjType by the witness relation Rel.

```
data Rel:: ObjType ~> *0 ~> *0 where
 IntR:: Rel IntT Int
 IntTo:: Rel b s -> Rel (ArrT IntT b) (Int -> s)
   -- First order functions only as constants
```

The structure of Rel relates only integers and first-order, n-ary functions over integers to the type ObjType. Consider the short Ωmega session:

```
prompt> IntR
IntR : Rel IntT Int

prompt> IntTo IntR
(IntTo IntR) : Rel (ArrT IntT IntT) (Int -> Int)

prompt> IntTo (IntTo IntR)
(IntTo (IntTo IntR)) : Rel (ArrT IntT (ArrT IntT IntT)) (Int -> Int -> Int)
```

Static versus Dynamic test for Mode. Finally, on occasion we will need to observe the structure of an object-level term, and compute whether it is a value in normal form, or a term with a redex. We do this by defining a singleton type reflecting the kind Mode into the value world, and by writing a total function that computes a safe approximation of the mode of any expression. By safe, we mean that no term is ever indexed by Exp if it is a value, though some terms might be indexed by Exp even though they do not contain a redex. Such terms generally have the form (App (Var ‘x) _), i.e. an application with a variable in the function part).

```
data Mode':: Mode ~> *0 where
  Exp':: Mode' Exp
  Val':: Mode' Val

mode :: E m e t -> Mode' m
mode (Lam v body) = Val'
mode (Var v) = Val'
mode (Const r v) = Val'
mode (Shift e) = mode e
mode (App _ _) = Exp'
```

Summary of changes. Thus a well-typed term of type `(E m env t)` is (1) a data structure representing an object-level term, (2) a derivation that the term is well typed with type `t` in environment `env`, and (3) a derivation that the term has mode `m`. Lets review the roles of the 3 kinds of indexes to `E`.

- `Mode.` The mode of the term. Either a `Val`, a term in normal form, or an `Exp`, a term with redex.
- `Row Tag ObjType.` The environment which indicates the position and type of the free variables in the term.
- `ObjType.` The object-level type of the term. Because of the relation `Rel`, we know only first order functions can be lifted from the meta-language to the object language.

There are two kinds of redexes in a term. β-redexes (explicit λ - expressions in the function position of an application) and δ-redexes (higher-order constants in the function position of an application). We give meaning to β-redexes by the use of substitution. Thus we need a well-typed version of substitution over object-level terms represented by `E`.

Substitution lemma. The key lemma behind the preservation part of the type-soundness proof is called the *substitution lemma*. The lemma says that if a term e has type σ under the assumption that some variable x has type τ, then substituting any term e' of type τ for x in e yields $e[e'/x]$ of type σ. In our version of the preservation proof, the lemma exhibits itself as a total well-typed function that performs substitution.

We choose to represent substitutions as data structures. This provides another example of object language syntax because our syntax is similar to explicit substitutions [5]. In this approach a substitution of type `(Sub e1 e2)` is a mapping from one environment (of kind `e1`) to another (of kind `e2`).

```
data Sub:: Row Tag ObjType ~> Row Tag ObjType ~> *0 where
  Id:: Sub r r
  Bind:: Label t -> E m r2 x -> Sub r r2 -> Sub (RCons t x r) r2
  Push:: Sub r1 r2 -> Sub (RCons a b r1) (RCons a b r2)

subst:: E m1 r t -> Sub r s -> exists m2 . E m2 s t
subst t             Id            = Ex t
subst (Const r c) sub             = Ex (Const r c)
subst (Var v)       (Bind u e r) = Ex e
subst (Var v)       (Push sub)   = Ex (Var v)
subst (Shift e)     (Bind _ _ r) = subst e r
subst (Shift e)     (Push sub)   = case subst e sub of {Ex a -> Ex(Shift a)}
subst (App f x)     sub          = case (subst f sub,subst x sub) of
                                      (Ex g,Ex y) -> Ex(App g y)
subst (Lam v x)     sub          = case subst x (Push sub) of
                                      (Ex body) -> Ex(Lam v body)
```

Preservation. In our proof, we perform steps 2 (define the one-step evaluation relation), 4 (prove progress), and 5 (prove type preservation) at once by defining a total single-step operation that operates on well-typed non-value closed terms. Its type is given by

```
onestep :: E m Closed t -> (E Exp Closed t + E Val Closed t).
```

Read logically this type says that every closed term (regardless of whether it is a value or an expression with a redex) can be transformed into another closed term with the same type, or is already a value.

```
type Closed = RNil

onestep :: E m Closed t -> (E Exp Closed t + E Val Closed t)
onestep (Var v)      = unreachable
onestep (Shift e)    = unreachable
onestep (Lam v body) = R (Lam v body)
onestep (Const r v)  = R(Const r v)
onestep (App e1 e2)  =
  case (mode e1,mode e2) of
    (Exp',_) ->
      case onestep e1 of
        L e -> L(App e e2)
        R v -> L(App v e2)
    (Val',Exp') ->
      case onestep e2 of
        L e -> L(App e1 e)
        R v -> L(App e1 v)
    (Val',Val') -> rule e1 e2
```

This function is a non-recursive case analysis. The `Var` and `Shift` cases are unreachable (they cannot be closed terms). The `Lam` and `Const` cases are already values. Observing the mode of the two parts of an application we have three choices. If the function is an expression with a possible redex, we take one step in the function part, and then rebuild the term. If the function part is a value, we must apply one of the β- or δ-rules. Note that the function part is always a closed term with an `(ArrT _ _)` object-level type.

```
rule::  E Val Closed (ArrT a b) ->
        E Val Closed a ->
        (E Exp Closed b + E Val Closed b)
rule (Var _)   _ = unreachable
rule (Shift _) _ = unreachable
rule (App _ _) _ = unreachable
-- The beta-rule
rule (Lam x body) v =
  let (Ex term) = subst body (Bind x v Id)
  in case mode term of
       Exp' -> L term
       Val' -> R term
rule (Const IntR _)       _                 = unreachable
rule (Const (IntTo b) _) (Var _)            = unreachable
rule (Const (IntTo b) _) (Shift _)          = unreachable
rule (Const (IntTo b) _) (App _ _)           = unreachable
rule (Const (IntTo b) f) (Lam x body)        = unreachable
rule (Const (IntTo b) f) (Const (IntTo _) x) = unreachable
-- The delta-rule
rule (Const (IntTo b) f) (Const IntR x)      = R(Const b (f x))
```

There are eleven cases. Nine of which are unreachable from type considerations (i.e. the inputs are not values, are not closed, or the first argument does not have an arrow type). We have structured our function body to make it explicit that we have covered every case. This allows us to prove (by a meta-level argument) that `rule` is total. In other systems (i.e. Twelf, Coq, etc.) this argument can be enforced by the type-system of the meta-language. In these systems all functions are total (or they are not accepted). In Ωmega, we aspire to this level of automated assistance, but as we think of Ωmega as a programming language (not a proof system) we must support both total and partial functions. We hope to separate total and partial functions by using the type system sometime in the near future.

The function `onestep` makes progress. By inspecting the code we see all values are immediately returned, and all non-values actually take one step forward.

5.13 Example: Constructing Typing Derivations at Runtime

At first glance, using GADTs to represent object-languages solves many problems. But, further introspection reveals a subtle problem. We can build typed object-level terms by typing constructed terms into our program using the constructors of the GADT, but how do we build such terms algorithmically? I.e. how do we write a parser, for example, that builds a well-typed object-level term? What would the type of the parser be? The type `(parse:: String -> E m e t)` is clearly not sufficient. Not every string can be parsed. But the type `(parse:: String -> Maybe(E m e t))` is also not sufficient. What mode, environment, and object level type should constrain the meta-level type variables `m`, `e`, and `t`? The type `(parse:: String -> exists m e t . Maybe(E m e t))` is closer to the mark, but this is also too unconstrained. We expect some properties to be true of these type variables. One solution is to build runtime representations that represent the constraints we envision, and runtime tests for these constraints, that we can execute at runtime.

We do this by building a singleton type to reflect the object-level types as meta-level runtime values (Section 3.6 and Exercise 9), and a runtime test for equality of these object-level type indexes (Exercises 16 and 23).

```
data Rep:: ObjType ~> *0 where
 I:: Rep IntT
 Ar:: Rep a -> Rep b -> Rep (ArrT a b)
```

In the function `compare`, because we want our runtime tests to report interesting error messages, the comparison returns a sum type, were the left injection (a failure) is an error message, and the right injection (a success) is an equality proof. Because the partial application of the type constuctor `(+)` to `String` is monadic [4], we use the `do` notation to specify what happens on success. On failure (of either `(comapre x s)` or `(compare y t)`) the error message in the left injection will be propogated.

```
compare:: Rep a -> Rep b -> (String + Equal a b)
compare I I = R Eq
compare (Ar x y) (Ar s t) =
  do { Eq <- compare x s
     ; Eq <- compare y t
     ; R Eq}
compare I (Ar x y) = L "I /= (Ar _ _)"
compare (Ar x y) I = L "(Ar _ _) /= I"
```

We will break our parsing problem into two parts. First, parsing a string into an untyped object-language representation (not shown in this paper, as this is the ordinary parsing problem). Second, transforming this untyped representation into a well-typed GADT representing a typed object-language term (or typing derivation, depending upon your perspective). In this report, we assume that the untyped representation suggests a type for every variable, and that our algorithm checks that this suggestion is correct. The inference problem is much harder, and not shown here. Our untyped representation follows:

```
data Term:: *0 where
  C:: Int -> Term
  Ab:: String -> Rep a -> Term -> Term
  Ap:: Term -> Term -> Term
  V:: String -> Term
```

We will check each term with respect to a given environment which maps every variable to an object-level type. It will also store the string used to name the variable in the untyped representation, and the label used to represent the variable in the typed-representation. Such an environment is indexed by `(Row Tag ObjType)` in the same manner as terms `E` and substitutions `Sub`.

```
data Env:: Row Tag ObjType ~> *0 where
    Enil:: Env RNil
    Econs:: Label t -> (String,Rep x) -> Env e -> Env (RCons t x e)
  deriving Record(e)
```

A key component of our algorithm, to produce a well-typed representation from an untyped representation, is to look up the type of a variable.

[4]
```
return:: a -> (String + a)
return x = R x
bind:: (String + a) -> (a -> (String + b)) -> (String + b)
bind (L message) f = Left message
bind (R x) f = f x
```

```
fail:: String -> (String + a)
fail s = L s

lookup:: String -> Env e -> (String + exists t m .(E m e t,Rep t))
lookup name Enil = fail ("Name not found: "++name)
lookup name {l=(s,t);rs}e | eqStr name s = R(Ex(Var l,t))
lookup name {l=(s,t);rs}e =
  do { Ex(v,t) <- lookup name rs
     ; R(Ex(Shift v,t)) }
```

If successful, both a representation of a type, and a term with that type are returned. Now we need put all this machinery together. The type checker is a program with the following prototype:

```
tc:: Term -> Env e -> (String + exists t m . (E m e t,Rep t))
```

Read logically, for every untyped term, and every environment with types for variables reflected in the row `e`, we can either report a type-checking error, or return a representation of a typed term. In this representation (consisting of a pair of a term and a singleton), its actual type and its mode are existentially quantified, but the actual object-level type is reflected in the "shape" of the runtime singleton object.

```
tc:: Term -> Env e -> (String + exists t m . (E m e t,Rep t))
tc (V s) env = lookup s env
tc (Ap f x) env =
  do { Ex(f',ft) <- tc f env
     ; Ex(x',xt) <- tc x env
     ; case ft of
         (Ar a b) ->
            do { Eq <- compare a xt
               ; R(Ex(App f' x',b)) }
         _ -> fail "Non fun in Ap" }
tc (Ab s t body) env =
  do { let (Hidden l) = newLabel s
     ; Ex(body',et) <- tc body {l=(s,t); env}e
     ; R(Ex(Lam l body',Ar t et)) }
tc (C n) env = R(Ex(Const IntR n,I))
```

The application case is the most interesting. First, recursively type-check the function and argument, returning typed terms `f'` and `x'`, and reflected types `ft` and `xt`. If either of these fails, the monad syntax causes the whole function to fail. Test that the function argument is really a function, and then compare the domain with the type of the argument. Only if this succeeds, and we have a proof that the two types are equal, can the whole case succeed.

5.14 The Bottom Line

The ability to define type-indexed GADTs, and the ability to define new kinds, creates a rich playground for those wishing to explore the design of new languages. These features, along with the use of rank-N polymorphism (which is beyond the scope of this paper) make Ωmega a better meta-language than Haskell. In order to explore the design of a new language one can proceed as follows:

– First, represent the object-language as a type-indexed GADT. The indexes correspond to static properties of the program.
– The indexes can have arbitrary structure, because they are introduced as the type constructors of new kinds.
– The typed constructor functions of the object-language GADT define a static semantics for the object language.
– Meta-programs written in Ωmegamanipulate object-language represented as data, and check and maintain the properties captured in the type indexes by using the meta-language type system. This lets us build and test type systems interactively.
– A dynamic semantics for the language can be defined by (1) writing either a large step semantics in the form of an interpreter or evaluation function, or by (2) writing a small step semantics in terms of substitution over the term language. In either case, the type system of the meta-language guarantees that these meta-level programs maintain object level type-safety.
– Normal operations such as pretty-printing and parsing functions can also be constructed, albeit with a little more cleverness than is ordinarily required.

6 Using Terms as Theorems

We can use a value of type `(Nat' n)` as a proof that `n` is a natural number. In Ωmega, ordinary datatypes can be used as constraints over types. A constraint can be discharged by exhibiting a non-divergent term with that type. The classic datatype used in this fashion is the equality type from Section 3.8. Recall:

```
data Equal :: a ~> a ~> *0 where
  Eq:: Equal x x
```

The `Equal` constraint can be applied to all types of the same kind because it is level polymorphic (see section 3.12). Thus `(Equal 2t 3t)` and `(Equal Int Bool)` are both well formed, but neither is inhabited (i.e. there are no non-divergent values with these types since (`Int` $\neq$ `Bool`) and (`(S(S Z))` $\neq$ `(S(S(S Z)))`)). The normal mode of use is to construct terms with types like (`Equal` x y) where x and `y` are type level function applications. For example consider the type of the function `plusZ` below. Its type: (`Nat' n -> Equal plus n Z n`) when read logically means *for all natural numbers* `n`*,* $n{+}0 = n$. One way to prove this is with a proof by induction over `n`. The following recursive definition of `plusZ` is a term witnessing this property. The `theorem` clause, inside the defintion of `plusZ` is a mechanism that helps organize this proof, and is explained in detail in the sequel.

```
plusZ :: Nat' n -> Equal {plus n Z} n
plusZ Z = Eq
plusZ (S m) = Eq
  where theorem indHyp = plusZ m
```

This function is a proof by induction that for all natural numbers `n` : `{plus n 0t}` = `n`. The definition exhibits a well-typed, total function with this type. The declaration, `where theorem indHyp = plusZ m`, instructs the type checker to use the type of the term `(plusZ m)` as a reasoning rule. Thus we may assume its type: `(Equal {plus b 0t} b)` while discharging `(Equal (S{plus b Z}) (S b))`.

To see that `plusZ` is well typed, the type checker does the following. The expected type is the type given in the function prototype. We compute the type of both the left- and right-hand-side of the equation defining a clause. We compare the expected type with the computed type for both the left- and right-hand-sides. This comparison generates some necessary equalities (for each side) to make the expected and computed types equal. We assume the left-hand-side equalities to prove the right-hand-side equalities. To see this in action, consider the two clauses of the definition of `plusZ`.

1.

expected type	`Nat' n` → `Equal {plus n Z} n`
equation	`plusZ Z` = `Eq`
computed type	`Nat' Z` → `Equal a a`
equalities	`n = Z` ⇒ `(a = n, a= {plus n Z})`

In the first case, the left-hand-side equalities let us assume `n` = `Z`. The right-hand-side equalities require us to establish that `a` = `{plus n Z}` and `a` = `n`. This can be established *iff* `n` = `{plus n Z}`. Using the assumption that `n` = `Z`, we are left with the requirement that `Z` = `{plus Z Z}`, which is easy to prove using the definition of `plus`.

2.

expected type	`Nat' n` → `Equal {plus n Z} n`
equation	`plusZ (S m)` = `Eq`
computed type	`Nat' (S b)` → `Equal a a`
equalities	`n = (S b)` ⇒ `(a = n, a= {plus n Z})`

In the second case, the left-hand-side assumptions are `n` = `(S b)` (where the pattern introduced variable `m` has type `(Nat' b)`). The right-hand-side equalities require us to establish that `a` = `{plus n Z}` and `a` = `n`. Again, this can only be established if `n` = `{plus n Z}`. Using the assumption that `n` = `(S b)`, we are left with the requirement that `(S b)` = `{plus (S b) Z}`. Using the definition of `plus`, this reduces to `(S b)` = `(S{plus b Z})`. To establish this fact, we use the inductive hypothesis. Since the argument `(S m)` is finitely constructed, and the function `plusZ` is total, the term, `(plusZ m)` exhibits a proof that `(Equal {plus b Z} b)`.

Other interesting facts, that are established in the same way, but omitted for brevity, include:

```
plusS :: Nat' n -> Equal {plus n (S m)} (S{plus n m})
plusCommutes :: Nat' n -> Nat' m -> Equal {plus n m} {plus m n}
plusAssoc :: Nat' n -> Equal {plus {plus n b} c} {plus n {plus b c}}
plusNorm :: Nat' x -> Equal {plus x {plus y z}} {plus y {plus x z}}
```

Exercise 27. Write an Ωmega function body for each of the prototypes above. The function bodies for `plusS` and `plusAssoc` are very similar to `plusZ`. The other two require appealing to theorems in addition to an induction hypotheses. In fact, `plusCommutes` requires both `plusZ` and `plusS` in addition to an induction hypothesis. We leave it to you to figure out what theorem is required for `plusNorm`.

6.1 Self Describing Combinatorial Circuits

Our next example is the description of combinatorial circuits. We will use types to ensure that our descriptions describe what they implement. We first describe the `Bit` type.

```
data Bit:: Nat ~> *0 where
  One :: Bit (S Z)
  Zero :: Bit Z
```

Like `Nat'`, `Bit` is a singleton type (see Section 3.6), there is only one value for each type. Note how the type of a bit carries the value of the bit as a natural number as its type index. I.e. `(One :: Bit 1t)` and `(Zero :: Bit 0t)`. We exploit this to define a data structure representing a base-2 number as a sequence of bits. The idea is for a value of type `(Binary Bit w v)` to represent a binary number built from a sequence of `Bits`, with width `w` and value `v`.

```
data Binary:: (Nat ~> *0) ~> Nat ~> Nat ~> *0 where
  Nil :: Binary bit Z Z
  Cons:: bit i -> Binary bit w n -> Binary bit (S w) {plus {plus n n} i}
```

Note that the type of the elements in the sequence has been abstracted to be any type constructor classified by the kind `(Nat ~> *0)`. In our first few examples, we will construct lists of (*Bit* `i`), so we will have values with type (`Binary` *Bit* `len value`) as a result. Later in the text, we will build binary numbers from other representations of bits.

A value with type `(Binary Bit 2t 3t)` is a sequence of `(Bit j)` values. The individual `j`'s are combined to represent a binary number with value `3t`. Binary numbers are stored least significant bit first. Prefixing a new bit shifts the previous bits into the next significant position, so the value of the new number is the value of the new bit plus twice the value of the old bits. Thus the type expression `{plus {plus n n} i}` in the type of `Cons` which prefixes a new bit. For example consider the term: `(Cons Zero (Cons One (Cons Zero (Cons Zero Nil))))` that has type `(Binary Bit 4t 2t)`. I.e. "0100" (where the least significant bit is left-most) has value 2 and width 4.

If we add three one-bit numbers, we always get a two bit result. We can write this function as follows.

```
add3Bits:: (Bit i) -> (Bit j) -> (Bit k) ->
           Binary Bit 2t {plus {plus j k} i}
add3Bits Zero Zero Zero = Cons Zero (Cons Zero Nil)
```

```
add3Bits Zero Zero One  = Cons One  (Cons Zero Nil)
add3Bits Zero One  Zero = Cons One  (Cons Zero Nil)
add3Bits Zero One  One  = Cons Zero (Cons One  Nil)
add3Bits One  Zero Zero = Cons One  (Cons Zero Nil)
add3Bits One  Zero One  = Cons Zero (Cons One  Nil)
add3Bits One  One  Zero = Cons Zero (Cons One  Nil)
add3Bits One  One  One  = Cons One  (Cons One  Nil)
```

This function is an exhaustive case analysis of all 8 possible combination of bits. It is exhaustive and total. Consider type checking one case.

expected type	`Bit i -> Bit j -> Bit k` → `Binary Bit 2t {plus {plus j k} i}`
equation	`add3Bits Zero One One` = `Cons Zero (Cons One Nil)`
computed type	`Bit 0t -> Bit 1t -> Bit 1t` → `Binary Bit 2t {plus {plus{plus {plus 0t 0t} 1t} {plus {plus 0t 0t} 1t} } 0t }`
equalities	`(i = 0t,j = 1t,k = 1t)` ⇒ `{plus {plus j k} i} = {plus {plus{plus {plus 0t 0t} 1t} {plus {plus 0t 0t} 1t} } 0t }`

Under the assumptions, both parts of the equality in the requirements for the right-hand-side reduce to `(Binary Bit t2 2t)`, so the clause is well typed. Iterating `add3Bits`, we can construct a ripple carry adder, whose type states that it is really an addition function!

```
add :: Bit c ->
       Binary Bit n i ->
       Binary Bit n j -> Binary Bit (S n) {plus {plus i j} c}
add c Nil Nil = Cons c Nil
add c (Cons x xs) (Cons y ys) =
  case add3Bits c x y of
    (Cons bit (Cons c2 Nil)) -> Cons bit (add c2 xs ys)
       where theorem plusCommutes, plusAssoc, plusNorm
```

The function `add` is type checked in the same manner as we illustrated with `plusZ` and `add3Bits`. In `add`, the type checker relies on the three theorems `plusCommutes`, `plusAssoc`, `plusNorm` that are the focus of Exercise 27 from the end of Section 6. We repeat their types here for convenience.

```
plusCommutes :: Equal {plus n m}          {plus m n}
plusAssoc    :: Equal {plus {plus n b} c} {plus n {plus b c}}
plusNorm     :: Equal {plus x {plus y z}} {plus y {plus x z}}
```

When used in conjunction, these theorems act as a set of left-to-right rewriting rules, and have a very strong normalizing effect. This effect occurs because the theorems `plusCommutes` and `plusNorm` are only applied if the rewritten term is lexigraphically smaller than the original term. For example, while type checking `add` the type checker uses them to repeatedly rewrite the term:

```
{plus {plus {plus {plus x3 x3} x2} {plus {plus x5 x5} x4}} x1}
        to the term:
{plus x1 {plus x2 {plus x3 {plus x3 {plus x4 {plus x5 x5}}}}}}
```

Exercise 28. Repeat the progression of defining the GADT `Binary` through defining the function `add`, but this time make `Binary` store most-significant bits on the left.

6.2 Symbolically Combining Bits

While we have shown how to use types to describe properties of programs, our adder is not a very effective hardware description. We need a data structure that can represent not only the constant bits, `One` and `Zero`, but also operations on bits. This motivates `BitX` (for eXtended bit).

```
data BitX:: Nat ~> *0 where
  OneX :: BitX (S Z)
  ZeroX :: BitX Z
  And:: BitX i -> BitX j -> BitX {and i j}
  Or:: BitX i -> BitX j -> BitX {or i j}
  Xor:: BitX i -> BitX j -> BitX {xor i j}
```

In order to track the result of *and*ing (*or*ing, *xor*ing) two bits, we need the `and` (`or`, `xor`) functions at the type level. These functions take any two natural numbers as input, but always return `0t` or `1t` as a result.

```
and :: Nat ~> Nat ~> Nat           | or :: Nat ~> Nat ~> Nat
{and Z Z} = Z                      | {or Z Z} = Z
{and Z (S n)} = Z                  | {or Z (S n)} = S Z
{and (S n) Z} = Z                  | {or (S n) Z} = S Z
{and (S n) (S n)} = S Z            | {or (S n) (S n)} = S Z
```

Exercise 29. Write the Ωmega type-level function:
`xor :: Nat ~> Nat ~> Nat`
that implements the exclusive-or function.

We can prove a number of interesting theorems about these functions by exhibiting terms with logical types. As we did with `add3Bits`, these functions are basically an exhaustive analysis of the cases. Here we prove that `and` is associative.

```
andAs :: Bit a -> Bit b -> Bit c ->
         Equal {and {and a b} c} {and a {and b c}}
andAs Zero Zero Zero = Eq
andAs Zero Zero One  = Eq
andAs Zero One  Zero = Eq
andAs Zero One  One  = Eq
andAs One  Zero Zero = Eq
andAs One  Zero One  = Eq
andAs One  One  Zero = Eq
andAs One  One  One  = Eq
```

Note, that this is a theorem about `Bit a`, `Bit b`, and `Bit c`, not about natural numbers `a`, `b`, and `c`. I.e.

```
(Bit a -> Bit b -> Bit c -> Equal {and {and a b} c} {and a {and b c}})
```

is a theorem but

```
(Nat' a -> Nat' b -> Nat' c -> Equal {and {and a b} c} {and a {and b c}}}
```

is not. A number of other useful theorems are proved in a similar manner.

```
andZ1:: Bit a -> Equal {and a Z} Z
andZ2:: Bit a -> Equal {and Z a} Z
andOne2:: Bit a -> Equal {and a (S Z)} a
andOne1:: Bit a -> Equal {and (S Z) a} a
```

Exercise 30. Following the pattern of `AndAs`, write function definitions for the above prototypes.

Every `(BitX i)` can be evaluated into a `(Bit i)` by applying the definitions of the operations `and`, `or` and `xor`. This is the purpose of the function `fromX`. Since the operations are functions at the type level, and we need operations on bits (which live at the value level) we define the functions `and'`, `or'` and `xor'`.

```
fromX :: BitX n -> Bit n
fromX OneX = One
fromX ZeroX = Zero
fromX (Or x y) = or' (fromX x) (fromX y)
fromX (And x y) = and' (fromX x) (fromX y)
fromX (Xor x y) = xor' (fromX x) (fromX y)
fromX (And3 x y z) =
      and' (fromX x) (and' (fromX y) (fromX z))

and' :: Bit i -> Bit j -> Bit {and i j}
and' Zero Zero = Zero
and' Zero One = Zero
and' One Zero = Zero
and' One One = One

or' :: Bit i -> Bit j -> Bit {or i j}
xor' :: Bit i -> Bit j -> Bit {xor i j}
```

Exercise 31. Write *Ω*mega function bodies for the omitted functions `or'` and `xor'`.

Because every `(BitX i)` can be evaluated into a `(Bit i)`, we can lift theorems about `Bit` to theorems about `BitX`. For example, consider the theorem:

```
andAs:: Bit a -> Bit b -> Bit c -> Equal {and {and a b} c} {and a {and b c}}
```

If `a`, `b` and `c` are `Bits`, then `a`, `b` and `c` associate under `and`. This is not the case for arbitrary `a`, `b` and `c`. Recall that the natural number indexes to `Bit` can only be 0 or 1. A similar theorem holds if `a`, `b` and `c` are `BitX`, and this theorem can be computed from the theorem involving `Bit`.

```
andAssoc:: BitX a -> BitX b -> BitX c ->
           Equal {and {and a b} c} {and a {and b c}}
andAssoc a b c = andAs (fromX a) (fromX b) (fromX c)
```

So unlike `andAs`, where we could not lift a theorem about `Bit` to a theorem about `Nat`, every theorem about `Nat` can be lifted to a theorem about `Bit`. With these tools, we can build a ripple carry adder that performs addition by applying the bit operations. For example, to add three one-bit numbers to obtain a two-bit result, we need to construct a logical formula that captures the following table.

```
inputs     sum

i j k      high bit  low bit
------
0 0 0      0         0
0 0 1      0         1
0 1 0      0         1          low bit  = (Xor i (Xor j k))
0 1 1      1         0          high bit = (Or (And i j)
1 0 0      0         1                         (Or (And i k)
1 0 1      1         0                             (And j k)))
1 1 0      1         0
1 1 1      1         1
```

To implement this is Ωmega, we introduce a 2-bit number `Pair` (more significant bit on the left), and the function `addthree`.

```
data Pair:: Nat ~> *0 where
 Pair:: BitX hi -> BitX lo -> Pair {plus {plus hi hi} lo}

addthree :: BitX i -> BitX j -> BitX k -> Pair {plus j {plus k i}}
addthree i j k = Pair (Or (And i j) (Or (And i k) (And j k)))
                                    (Xor i (Xor j k))
  where theorem lemma = logic3 (fromX i) (fromX j) (fromX k)
```

Unlike the function `add3Bits`, we cannot type check `addthree` by exhaustively enumerating all possible inputs because there are an infinite number of possible terms of type `(BitX i)` for each natural number `i`. But we can prove a lemma about `Bit` (which we can prove by exhaustive analysis) and then lift it to a theorem about `BitX`. This is the role of the term `(logic3 (fromX i) (fromX j) (fromX k))` in the `theorem` clause in `addthree`.

```
logic3 :: Bit i -> Bit j -> Bit k ->
          (Equal {plus {plus {or {and i j}
                                 {or {and i k} {and j k}}}
                             {or {and i j}
                                 {or {and i k} {and j k}}}}
                       {xor i {xor j k}}}
                 {plus j {plus k i}})
logic3 Zero Zero Zero = Eq
logic3 Zero Zero One  = Eq
logic3 Zero One  Zero = Eq
logic3 Zero One  One  = Eq
logic3 One  Zero Zero = Eq
logic3 One  Zero One  = Eq
logic3 One  One  Zero = Eq
logic3 One  One  One  = Eq
```

We can now re-implement our ripple carry adder, but this time by symbolically combining the input bits, to compute the output bits as a logical function of the inputs. This function has a similar type, the same structure, and uses the same theorems as the function add.

```
addBits :: BitX c -> Binary BitX n i -> Binary BitX n j ->
           Binary BitX (S n) {plus {plus i j} c}
addBits c Nil Nil = Cons c Nil
addBits c (Cons x xs) (Cons y ys) =
  case addthree c x y of
    (Pair c2 bit) -> Cons bit (addBits c2 xs ys)
       where theorem plusCommutes, plusAssoc, plusNorm
```

To actually compute a circuit we need to have some symbolic inputs. We do this by extending the type BitX with a constructor to represent variables. We can then construct some inputs, and compute the description of an adder. Our function works on inputs of any size.

```
data BitX:: Nat ~> *0 where
  . . .
  X:: Int -> BitX a

xs :: Binary BitX 2t {plus {plus a a} b}
xs = Cons (X 1) (Cons (X 2) Nil)

ys :: Binary BitX 2t {plus {plus a a} b}
ys = Cons (X 3) (Cons (X 4) Nil)
carry = (X 5)

ans = addBits carry xs ys
```

Here xs and ys are two-bit symbolic inputs, and carry is a symbolic input carry. Calling addBits we construct an output which is a (Binary Bit) list with three elements, each of which is a combinatorial function of the input bits, whose value is guaranteed by the types to be the sum of the inputs! Below, we display the output with a pretty printer that displays (X n) as "xn", and indents the display to emphasize its structure.

```
(Cons (Xor x5
          (Xor x1 x3))
(Cons (Xor (Or (And x5 x1)
               (Or (And x5 x3)
                   (And x1 x3)))
           (Xor x2 x4))
(Cons (Or (And (Or (And x5 x1)
                   (Or (And x5 x3)
                       (And x1 x3)))
               x2)
          (Or (And (Or (And x5 x1)
                       (Or (And x5 x3)
                           (And x1 x3)))
                   x4)
             (And x2 x4)))
Nil)))
```

The key property here is that the type of this structure guarantees that it implements an addition function.

Exercise 32. There are many equivalencies between boolean expressions. Any function with the type: (BitX n -> Maybe (BitX n)) can be thought of as a meaning preserving transformation. Given a value typed v:: BitX n, a meaning preserving transformation returns (Just u) or Nothing. If it returns (Just u) then u is semantically equivalent to v. If it returns Nothing we interpret this to mean the transformation did not apply to v. Choose a few boolean laws and implement them as meaning preserving transformations as discussed above.

Exercise 33. Transformations can be combined by placing them in a list, and applying them using transformation combinators. Consider functions with the types below:

```
first:: [BitX n -> Maybe(BitX n)] -> BitX n -> Maybe(BitX n)
all:: [BitX n -> Maybe(BitX n)] -> BitX n -> [BitX n]
```

The combinator first lifts a list of transformations to a single transformation, applying the first applicable transformation in the list. The combinator all finds all applicable transformations and returns a list of all possible results, including the untransformed term as well. Define these two functions in Ωmega.

The combinator retry continually re-applies a meaning preserving transformation until the term reaches a fixed-point. What is the type of retry? Write an Ωmega function body for retry. What other combinators can you think of?

6.3 A Caveat

The addition of the variable BitX constructor X was necessary if we want to use our functions to build hardware descriptions. Without it, we can only build constant combinatorial circuits! Unfortunately, it breaks the soundness of our descriptions. The lack of soundness flows from the fact that our function `fromX` is no longer total. How do we turn a variable into a `Bit`? Thus, we can no longer lift facts about the functions `and`, `or`, and `xor` and the type `Bit` to facts about the type `BitX`. To overcome this limitation we would need to track the variables in the type of `BitX` objects. For example we may write `(BitX Bit env width value)` as the type of a binary number whose free variables are described by `env`. Now, we must recast our theorems in terms of `(BitX Bit env width value)` and well formed environments `env`. This is sufficient, because a well formed environment means every variable will eventually be replaced by a bit, and in this new formulation the lifting of theorems hold.

Exercise 34. Using the patterns discussed in Section 5.9 for languages with binding structures, re-do the progression from the GADT `BitX` to the function `addBits`, but this time track the variables in the types of `BitX`. Recast the theorems about `BitX` so that they hold for all environments.

7 Conclusion

We hope that the programs and exercises described in this paper give you, the reader, an appreciation for the power of types in describing the properties of programs. Additional resources and papers can be found on the authors web page `http://cs.pdx.edu/~sheard` where you can also obtain the *Ω*mega system for download.

7.1 Relation to Other Systems

In order to make *Ω*mega accessible to as broad an audience as possible, it is built around a framework which appears to the user to be a pure but strict version of Haskell. *Ω*mega was designed, first and foremost, to be a programming language. Our goal was to design a language where program specifications, program properties, program analyses, proofs about programs, and programs themselves, are all represented using a single unifying notion of term. Thus programmers communicate many different things using the same language.

Our second goal was to make *Ω*mega a logic, in which our reasoning would be sound. This is the basis of our decision to make *Ω*mega strict. We made this design decision because the use of GADTs as proof objects requires that bottom not be an inhabitant of certain types. Strictness is part of our eventual strategy to accomplish that goal. This goal is not yet achieved.

There are many systems where soundness was the principal goal, and has been achieved. All of the examples, except for the staged examples, could be done in these languages as well. Such systems were principally designed to be logical frameworks or theorem provers. These include Inductive Families [9, 12], theorem provers (Coq [37], Isabelle [20]), logical frameworks (Twelf [22], LEGO [14]), and proof assistants (ALF [17], Agda [8]). Recently, there has been much interest in systems that use dependent types to build "practical" systems that are part language, part reasoning system. These systems include Augustsson's Cayenne language [2, 3], McBride's Epigram [15], Stump's Rogue-Sigma-Pi [33, 38], Xi and Pfenning's Dependent ML [11, 42], and Xi's Applied Type Systems [7, 41]. In fact, we owe a large debt to all these systems for inspiration.

We realize that just *a little* loss in soundness makes all our reasoning claims vacuous, but we are working to fill these gaps. Our goal is to do this in a different manner than the systems listed above, which require all functions to be total in order to ensure soundness. We wish to use types to separate terminating functions from non-terminating functions, and make logical claims only about the terminating fragment of the language. This seems almost a necessary condition for a system that claims to be a programming language. In any case, these issues have little effect on our use of *Ω*mega to program generic programs, since logical soundness is not an issue in this domain.

Acknowledgments

This paper was originally prepared for the Central-European Functional Programming School held in Cluj, Romania, between 25-30 June, 2007. The text, code, and examples come from many sources. Here is quick (though, not by any means exhaustive) list.

1. Section 2 comes from the paper *Generic Programming in Ωmega* [31]. This paper has many additional examples, not covered in this report.
2. Section 3 also comes from the same paper, as well as the *Ωmega Users' Guide* [29], both of which have their roots in the paper *Putting Curry-Howard to Work* [28].
3. The shaped tree example with paths (Section 3.1) comes from discussions with James L. Caldwell from the University of Wyoming.
4. The exercise about Red-Black trees comes from some examples posted by Hong-Wei Xi for the ATS system.
   `http://www.cs.bu.edu/~hwxi/ATS/ATS.html`
5. The Leibniz equality GADT has its roots in two papers [4, 13] which introduce similar types with slight variants.
6. The material on using *Ω*mega as a meta-language comes from an unpublished paper called *Playing with Types* [27], though many of the examples used here are original to this report.
7. The combinatorial circuit examples come from a paper *Types and Hardware Description Languages* [30] prepared for the *Hardware design and Functional Languages* workshop held March 24-25 2007, in Braga, Portugal.
8. The answers to selected exercises was created by Ki-Yung Ahn.
9. This material is based upon work supported by the National Science Foundation under Grant No. 0613969.

References

1. Antoy, S.: Definitional trees. In: Kirchner, H., Levi, G. (eds.) ALP 1992. LNCS, vol. 632, pp. 143–157. Springer, Heidelberg (1992)
2. Augustsson, L.: Cayenne — a language with dependent types. ACM SIGPLAN Notices 34(1), 239–250 (1999)
3. Augustsson, L.: Equality proofs in Cayenne, July 11 (2000), `http://www.cs.chalmers.se/~augustss/cayenne/eqproof.ps`
4. Baars, A.I., Swierstra, S.D.: Typing dynamic typing. In: Jones, S.P. (ed.) Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, pp. 157–166. ACM Press, New York (2002)
5. Benaissa, Z.-E.-A., Briaud, D., Lescanne, P., Rouyer-Degli, J.: lambda-nu, A calculus of explicit substitutions which preserves strong normalisation. J. Funct. Program 6(5), 699–722 (1996)
6. Cardelli, L., Wegner, P.: On understanding types, data abstraction and polymorphism. ACM Computing Surveys 17(4), 471–522 (1985)
7. Chen, C., Xi, H.: Combining programming with theorem proving. In: ICFP 2005 (2005), `http://www.cs.bu.edu/~hwxi/`
8. Coquand, C.: Agda is a system for incrementally developing proofs and programs. Web page describing AGDA, `http://www.cs.chalmers.se/~catarina/agda/`
9. Coquand, T., Dybjer, P.: Inductive definitions and type theory: an introduction. In: Thiagarajan, P.S. (ed.) FSTTCS 1994. LNCS, vol. 880, pp. 60–76. Springer, Heidelberg (1994)
10. Damas, L.: Type assignment in programming languages. PhD thesis, Edinburgh University CST-33-85 (1985)
11. Davies, R.: A refinement-type checker for Standard ML. In: Johnson, M. (ed.) AMAST 1997. LNCS, vol. 1349. Springer, Heidelberg (1997)
12. Dybjer, P., Setzer, A.: A finite axiomatization of inductive-recursive definitions. In: Girard, J.-Y. (ed.) TLCA 1999. LNCS, vol. 1581, pp. 129–146. Springer, Heidelberg (1999)
13. Hinze, R., Cheney, J.: A lightweight implementation of generics and dynamics. In: Chakravarty, M. (ed.) Proceedings of the ACM SIGPLAN 2002 Haskell Workshop ACM SIGPLAN, pp. 90–104 (October 2002)
14. Luo, Z., Pollack, R.: LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept., University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, Updated version (May 1992)
15. McBride, C.: Epigram: Practical programming with dependent types. In: Notes from the 5th International Summer School on Advanced Functional Programming (August 2004), `http://www.dur.ac.uk/CARG/epigram/epigram-afpnotes.pdf`
16. Milner, R.: A theory of type polymorphism in programming languages. Journal of Computer and System Science 17(3), 348–375 (1978)
17. Nordstrom, B.: The ALF proof editor (March 20, 1996), `ftp://ftp.cs.chalmers.se/pub/users/ilya/FMC/alfintro.ps.gz`
18. Pasalic, E., Linger, N.: Meta-programming with typed object-language representations. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 136–167. Springer, Heidelberg (2004)
19. Pasalic, E., Taha, W., Sheard, T.: Tagless staged interpreters for typed languages. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP 2002), Pittasburgh, PA, October 4–6, pp. 218–229. ACM Press, New York (2002)

20. Paulson, L.C.: Isabelle: The next 700 theorem provers. In: Odifreddi, P. (ed.) Logic and Computer Science, pp. 361–386. Academic Press, London (1990)
21. Jones, S.P.: Special issue: Haskell 98 language and libraries. Journal of Functional Programming 3 (January 2003)
22. Pfenning, F., Schürmann, C.: System description: Twelf — A meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
23. Shao, Z., Saha, B., Trifonov, V., Papaspyrou, N.: A type system for certified binaries. ACM SIGPLAN Notices 37(1), 217–232 (2002)
24. Sheard, T.: Using MetaML: A staged programming language. In: Swierstra, S.D., Oliveira, J.N. (eds.) AFP 1998. LNCS, vol. 1608, pp. 207–239. Springer, Heidelberg (1999)
25. Sheard, T., Peyton-Jones, S.: Template meta-programming for Haskell. In: Proc. of the workshop on Haskell, pp. 1–16. ACM Press, New York (2002)
26. Sheard, T.: Accomplishments and research challenges in meta-programming. In: Taha, W. (ed.) SAIG 2001. LNCS, vol. 2196, pp. 2–44. Springer, Heidelberg (2001)
27. Sheard, T.: Playing with types. Technical report, Portland State University (2005), `http://www.cs.pdx.edu/~sheard`
28. Sheard, T.: Putting Curry-Howard to work. In: Proceedings of the ACM SIGPLAN 2005 Haskell Workshop, pp. 74–85 (2005)
29. Sheard, T.: Omega users' gude. Technical report, Portland Stage University (2005-2007), `http://web.cecs.pdx.edu/~sheard/Omega/index.html`
30. Sheard, T.: Types and hardware description languages. In: Proceedings of the Hardware design and Functional Languages workshop (March 2007), `http://web.cecs.pdx.edu/~sheard/`
31. Sheard, T.: Generic programming in omega. In: Notes from the Spring School on Datatype-Generic Programming. LNCS (to appear, 2006)
32. Stone, C.A., Harper, R.: Deciding type equivalence in a language with singleton kinds. In: Conference Record of POPL 2000: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, January 19–21, pp. 214–227 (2000)
33. Stump, A.: Imperative LF meta-programming. In: Logical Frameworks and Meta-Languages workshop (July 2004), `http://cs-www.cs.yale.edu/homes/carsten/lfm04/`
34. Taha, W.: Tag elimination - or - type specialisation is a type-indexed effect. In: APPSEM Workshop on Subtyping & Dependent Types in Programming. Ponte de Lima Portugal (July 2000), `http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html`
35. Taha, W., Makholm, H., Hughes, J.: Tag elimination and jones-optimality. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, p. 257. Springer, Heidelberg (2001)
36. Taha, W., Sheard, T.: MetaML: Multi-stage programming with explicit annotations. Theoretical Computer Science 248(1-2) (2000)
37. The Coq Development Team. The Coq Proof Assistant Reference Manual, Version 7.4. INRIA (2003), `http://pauillac.inria.fr/coq/doc/main.html`
38. Westbrook, E., Stump, A., Wehrman, I.: A language-based approach to functionally correct inperative programming. Technical report, Washington University in St. Louis (2005), `http://cl.cse.wustl.edu/`

39. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1), 38–94 (1994)
40. Xi, H.: Dependent Types in Practical Programming. PhD thesis, Carnegie Mellon University (1997)
41. Xi, H.: Applied type systems (extended abstract). In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 394–408. Springer, Heidelberg (2004)
42. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. ACM SIGPLAN Notices 33(5), 249–257 (1998)
43. Xi, H., Pfenning, F.: Dependent types in practical programming. In: ACM (ed.) POPL 1999. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, ACM SIGPLAN Notices, San Antonio, TX, January 20–22, 1999, pp. 214–227. ACM Press, New York (1999)

A Red-Black Tree Insertion

```
------------------------------------------------------------------------
-- Introduce a new kind to represent colors

kind Color  = Red | Black

------------------------------------------------------------------------
-- Top-level type that hides both
-- color of the node and tree height

data RBTree:: *0 where
 Root:: SubTree Black n -> RBTree

------------------------------------------------------------------------
-- GADT that captures invariants

data SubTree:: Color ~> Nat ~> *0 where
 Leaf:: SubTree Black Z
 RNode:: SubTree Black n ->
         Int ->
         SubTree Black n ->
         SubTree Red n
 BNode:: SubTree cL m ->
         Int ->
         SubTree cR m ->
         SubTree Black (S m)

------------------------------------------------------------------------
-- A Ctxt records where we've been as we descend
-- down into a tree as we search for a value

data Dir = LeftD | RightD
data Ctxt:: Color ~> Nat ~> *0 where
  Nil:: Ctxt Black n
  RCons:: Int -> Dir ->
          SubTree Black n ->
          Ctxt Red n ->
          Ctxt Black n
  BCons:: Int -> Dir ->
          SubTree c1 n ->
          Ctxt Black (S n) ->
          Ctxt c n

-------------------------------------------------------------------------
-- Turn a Red tree into a black tree. Always
-- possible, since Black nodes do not restrict
-- the color of their sub-trees.

blacken :: SubTree Red n -> SubTree Black (S n)
blacken (RNode l e r) = (BNode l e r)

-------------------------------------------------------------------------
-- A singleton type representing Color at
-- the value level.

data CRep :: Color ~> *0 where
  Red   :: CRep Red
  Black :: CRep Black

color :: SubTree c n -> CRep c
color Leaf = Black
color (RNode _ _ _) = Red
color (BNode _ _ _) = Black

-------------------------------------------------------------------------
-- fill a context with a subtree to regain the original
-- RBTree, works if the colors and black depth match up

fill :: Ctxt c n -> SubTree c n -> RBTree
fill Nil t = Root t
fill (RCons e LeftD  uncle c) tree = fill c (RNode uncle e tree)
fill (RCons e RightD uncle c) tree = fill c (RNode tree  e uncle)
fill (BCons e LeftD  uncle c) tree = fill c (BNode uncle e tree)
fill (BCons e RightD uncle c) tree = fill c (BNode tree  e uncle)

insert :: Int -> RBTree -> RBTree
insert e (Root t) = insert_ e t Nil

-------------------------------------------------------------------------
-- as we walk down the tree, keep track of everywhere
-- we've been in the Ctxt input.
```

```
insert_ :: Int -> SubTree c n -> Ctxt c n -> RBTree
insert_ e (RNode l e' r) ctxt
       | e < e'        = insert_ e l (RCons e' RightD r ctxt)
       | True          = insert_ e r (RCons e' LeftD  l ctxt)
insert_ e (BNode l e' r) ctxt
       | e < e'        = insert_ e l (BCons e' RightD r ctxt)
       | True          = insert_ e r (BCons e' LeftD  l ctxt)
-- once we get to the bottom we "insert" the node as a Red node.
-- since this may break invariant, we may need do some patch work
insert_ e Leaf ctxt = repair (RNode Leaf e Leaf) ctxt

-------------------------------------------------------------------------
-- Repair a tree if its out of balance. The Ctxt holds
-- crucial information about colors of parent and
-- grand-parent nodes.

repair :: SubTree Red n -> Ctxt c n -> RBTree
repair t (Nil)                 = Root (blacken t)
repair t (BCons e LeftD  sib c) = fill c (BNode sib e t)
repair t (BCons e RightD sib c) = fill c (BNode t e sib)
-- these are the tricky cases
repair t (RCons e dir sib (BCons e' dir' uncle ctxt)) =
  case color uncle of
    Red   -> repair (recolor dir e sib dir' e' (blacken uncle) t) ctxt
    Black -> fill ctxt (rotate dir e sib dir' e' uncle t)
repair t (RCons e dir sib (RCons e' dir' uncle ctxt)) = unreachable

recolor :: Dir -> Int -> SubTree Black n ->
           Dir -> Int -> SubTree Black (S n) ->
           SubTree Red n -> SubTree Red (S n)
recolor LeftD  pE sib RightD gE uncle t = RNode (BNode sib pE t) gE uncle
recolor RightD pE sib RightD gE uncle t = RNode (BNode t pE sib) gE uncle
recolor LeftD  pE sib LeftD  gE uncle t = RNode uncle gE (BNode sib pE t)
recolor RightD pE sib LeftD  gE uncle t = RNode uncle gE (BNode t pE sib)

rotate :: Dir -> Int -> SubTree Black n ->
          Dir -> Int -> SubTree Black n ->
          SubTree Red n -> SubTree Black (S n)
rotate RightD pE sib RightD gE uncle (RNode x e y) =
   BNode (RNode x e y) pE (RNode sib gE uncle)
rotate LeftD  pE sib RightD gE uncle (RNode x e y) =
   BNode (RNode sib pE x) e (RNode y gE uncle)
rotate LeftD  pE sib LeftD  gE uncle (RNode x e y) =
   BNode (RNode uncle gE sib) pE (RNode x e y)
rotate RightD pE sib LeftD  gE uncle (RNode x e y) =
   BNode (RNode uncle gE x) e (RNode y pE sib)
```

B Inductively Sequential Functions

We restrict the form of function definitions at the type level and higher to be inductively sequential [1]. If a type function is not inductively sequential then the type checker rejects that type function.

Inductively sequential type functions ensures a sound and complete narrowing strategy for answering type-checking time questions. The class of inductively sequential functions is a large one, in fact every Haskell function has an inductively sequential definition. The inductively sequential restriction affects the form of the equations, and not the functions that can be expressed. Informally, a function definition is inductively sequential if all its clauses are non-overlapping. For example the definition of `zip1` is not inductively sequential, but the equivalent program `zip2` is.

```
zip1 (x:xs) (y:ys) = (x,y): (zip1 xs ys)
zip1 xs ys = []

zip2 (x:xs) (y:ys) = (x,y): (zip2 xs ys)
zip2 (x:xs) []     = []
zip2 []     ys     = []
```

The definition for `zip1` is not inductively sequential, since its two clauses overlap. In general any non-inductively sequential definition can be turned into an inductively sequential definition by duplicating some of its clauses, instantiating variable patterns with constructor based patterns. This will make the new clauses non-overlapping. We do not think this burden is too much of a burden to pay, since it is applied only to functions at the type level, and it supports sound and complete narrowing strategies. In addition to the inductively sequential form required for type functions, Ωmega assumes that each type function is a total terminating function. This assumption is not currently enforced, and it is up to the programmer to ensure that this is the case.

C Answers to Selected Exercises

```
--------------
-- Exercise  1
--------------
data Seq :: *0 ~> Nat ~> *0 where
  Snil  :: Seq a Z
  Scons :: a -> Seq a n -> Seq a (S n)

length :: Seq a n -> Int
length Snil         = 0
length (Scons _ xs) = 1 + length xs

-- we can can also use (Nat' n) (see 3.7)
-- to ensure that the size of the result is n

length' :: Seq a n -> Nat' n
legnth' Snil         = Z
length' (Scons _ xs) = S (length' xs)
```

```
--------------
-- Exercise  3
--------------
data Color :: *1 where
  Red   :: Color
  Black :: Color

data RBT :: Color ~> *0 where
  LeafB :: RBT Black
  NodeR :: RBT Black -> RBT Black -> RBT Red
  NodeB :: RBT cL    -> RBT cR    -> RBT Black


--------------
-- Exercise  4
--------------
plus :: Nat ~> Nat ~> Nat
{plus Z m} = m
{plus (S n) m} = S {plus n m}

mult :: Nat ~> Nat ~> Nat
{mult Z m} = Z
{mult (S n) m} = {plus {mult n m} m}


--------------
-- Exercise  5
--------------
data Boolean :: *1 where
  T :: Boolean
  F :: Boolean

odd :: Nat ~> Boolean
{odd Z}     = F
{odd (S Z)} = T
{odd (S (S n))} = {odd n}

--------------
-- Exercise  6
--------------

or :: Boolean ~> Boolean ~> Boolean
{or T b} = T
{or F b} = b

-- The function (not :: Bool -> Bool) is predefined
-- so we use different name
not' :: Boolean ~> Boolean
{not' T} = F
{not' F} = T
```

```
--------------
-- Exercise  7
--------------
data Shape :: *1 where
  Tp :: Shape
  Nd :: Shape
  Fk :: Shape ~> Shape ~> Shape

data Path :: Shape ~> *0 ~> *0 where
  None  :: Path Tp a
  Here  :: b -> Path Nd b
  Left  :: Path x a -> Path (Fk x y) a
  Right :: Path y a -> Path (Fk x y) a

data Tree :: Shape ~> *0 ~> *0 where
  Tip :: Tree Tp a
  Node :: a -> Tree Nd a
  Fork :: Tree x a -> Tree y a -> Tree (Fk x y) a

extract :: Path sh a -> Tree sh a -> a
extract None      Tip           = error "(extract None Tip) has nothing"
extract (Here _)  (Node v)      = v
extract (Left p)  (Fork lt rt) = extract p lt
extract (Right p) (Fork lt rt) = extract p rt

--------------
-- Exercise  8
--------------
data ListShape :: *1 where
  LSnil  :: ListShape
  LScons :: ListShape ~> ListShape

data List :: ListShape ~> *0 ~> *0 where
  Lnil  :: List LSnil a
  Lcons :: a -> List sh a -> List (LScons sh) a

data ListPath :: ListShape ~> *0 ~> *0 where
  ListNone :: ListPath LSnil a
  ListHere :: b -> ListPath (LScons sh) b
  ListNext :: ListPath sh a -> ListPath (LScons sh) a

find :: (a -> a -> Bool) -> a -> List sh a -> Maybe(ListPath sh a)
find eq n Lnil
   = Nothing
find eq n (Lcons x xs)
   = if eq n x
        then Just (ListHere n)
        else case find eq n xs of
               Nothing -> Nothing
               Just p  -> Just (ListNext p)
```

```
--------------
-- Exercise  9
--------------
data Rep :: *0 ~> *0 where
  Int  :: Rep Int
  Bool :: Rep Bool
  Prod :: Rep a -> Rep b -> Rep (a,b)
  List :: Rep a -> Rep [a]

showR :: Rep a -> a -> String
showR Int        n  = show n
showR Bool       True  = "True"
showR Bool       False  = "False"
showR (Prod x y) (a,b) = "("++showR x a++","++showR y b++")"
showR (List t)   xs = "["++ help xs ++ "]"
  where help [x] = showR t x
        help []  = ""
        help (x:xs) = showR t x++","++help xs

--------------
-- Exercise 10
--------------
data Plus :: Nat ~> Nat ~> Nat ~> *0 where
  PlusZ :: Plus Z m m
  PlusS :: Plus n m z -> Plus (S n) m (S z)

plus2v3v5v :: Plus 2t 3t 5t
plus2v3v5v = PlusS (PlusS PlusZ)

plus2v1v3v :: Plus 2t 1t 3t
plus2v1v3v = PlusS (PlusS PlusZ)

plus2v6v8v :: Plus 2t 6t 8t
plus2v6v8v = PlusS (PlusS PlusZ)


--------------
-- Exercise 11
--------------
data LE :: Nat ~> Nat ~> *0 where
  LeZ :: LE Z n
  LeS :: LE n m -> LE (S n) (S m)

sumandLessThanOrEqualToSum :: Plus a b c -> LE a c
sumandLessThanOrEqualToSum PlusZ     = LeZ
sumandLessThanOrEqualToSum (PlusS p) = LeS (sumandLessThanOrEqualToSum p)

-- Can we define a function with type (Plus a b c -> LE b c)?
-- not exactly, but we can write one with a similar type.
```

```
sumandLTorEQ2sum' :: Nat' c -> Plus a b c -> LE b c
sumandLTorEQ2sum' n     PlusZ     = same n
sumandLTorEQ2sum' Z     (PlusS _) = unreachable
sumandLTorEQ2sum' (S n) (PlusS p) = predLE (sumandLTorEQ2sum' n p)

-- see Exercise 13 for the definitions of same and predLE.

--------------
-- Exercise 12
--------------
even :: Nat ~> Boolean
{even Z}     = T
{even (S Z)} = F
{even (S (S n))} = {even n}

data EvenRel :: Nat ~> Boolean ~> *0 where
  Er0  :: EvenRel 0t T
  Er1  :: EvenRel 1t F
  ErSS :: EvenRel n b -> EvenRel (S (S n)) b


--------------
-- Exercise 13
--------------
same :: Nat' n -> LE n n
same Z     = LeZ
same (S n) = LeS (same n)

predLE :: LE m n -> LE m (S n)
predLE LeZ     = LeZ
predLE (LeS p) = LeS (predLE p)


--------------
-- Exercise 14
--------------
trans :: LE a b -> LE b c -> LE a c
trans LeZ      _        = LeZ
trans (LeS _)  LeZ      = unreachable
trans (LeS p1) (LeS p2) = LeS (trans p1 p2)


--------------
-- Exercise 15
--------------
f15 :: Nat' b -> Plus a b c -> LE b c
f15 n     PlusZ     = same n
f15 Z     (PlusS _) = LeZ
f15 (S n) (PlusS p) = predLE (f15 (S n) p)
```

```
--------------
-- Exercise 16
--------------
sameNat' :: Nat' a -> Nat' b -> Maybe (Equal a b)
sameNat' Z     Z     = Just Eq
sameNat' Z     (S _) = Nothing
sameNat' (S _) Z     = Nothing
sameNat' (S n) (S m) = case sameNat' n m of
                         Nothing -> Nothing
                         Just Eq -> Just Eq

--------------
-- Exercise 17
--------------
filter :: (a->Bool) -> Seq a n -> exists m . (Nat' m, Seq a m)
filter p Snil         = Ex (Z, Snil)
filter p (Scons x xs) =
  case filter p xs of
    Ex (n, xs') -> if p x then Ex (S n, Scons x xs')
                          else Ex (n, xs')

filter' :: (a->Bool) -> Seq a n -> exists m . (LE m n, Nat' m, Seq a m)
filter' p Snil         = Ex (LeZ, Z, Snil)
filter' p (Scons x xs) =
  case filter' p xs of
    Ex (le, n, xs') -> if p x then Ex (LeS le, S n, Scons x xs')
                              else Ex (predLE le, n, xs')

--------------
-- Exercise 18
--------------
pow :: Int -> Code Int -> Code Int
pow 0 _ = [| 1 |]
pow n x = [| $(x) * $(pow (n - 1) x) |]

--------------
-- Exercise 19
--------------
-- Row is already defined so we use MyRow

data MyRow :: a ~> c ~> *1 where
  Rnil :: MyRow e f
  Rcons :: e ~> f ~> MyRow e f ~> MyRow e f
 deriving Record(mr)

-- We derive syntax 'mr' because the
-- predefined Row uses syntax 'r' already.
```

```
--------------
-- Exercise 20
--------------
data Nsum :: *0 ~> *0 where
  SumZ :: Nsum Int
  SumS :: Nsum x -> Nsum (Int -> x)
 deriving Nat(i)

-- 0i : Nsum Int
-- 1i : Nsum (Int -> Int)
-- 2i : Nsum (Int -> Int -> Int)

add :: Nsum i -> i
add = add' 0

add' :: Int -> Nsum i -> i
add' x 0i     = x
add' x (1+n)i = \k -> add' (x+k) n


--------------
-- Exercise 21
--------------
data Expr :: *0 where
  VarExpr  :: Label t -> Expr
  PlusExpr :: Expr -> Expr -> Expr

valueOf :: Expr -> [exists t .(Label t,Int)] -> Int
valueOf (VarExpr v)    env = lookup v env
valueOf (PlusExpr x y) env = valueOf x env + valueOf y env

lookup :: Label v -> [exists t .(Label t,Int)] -> Int
lookup v ((Ex(u,n)):xs) =
 case labelEq v u of
   Just Eq -> n
   Nothing -> lookup v xs

pair1:: exists t .(Label t,Int)
pair1 = Ex(`a,5)

pair2:: exists t .(Label t,Int)
pair2 = Ex(`x,22)

pair3:: exists t .(Label t,Int)
pair3 = Ex(`z,2)

table :: [exists t .(Label t,Int)]
table = [pair1,pair2,pair3]

xValue = valueOf (VarExpr `x) table
```

```
--------------
-- Exercise 22
--------------

-- see Appendix A


--------------
-- Exercise 23
--------------

{-  already defined in Exercise 9
data Rep :: *0 ~> *0 where
  Int  :: Rep Int
  Bool :: Rep Bool
  Prod :: Rep a -> Rep b -> Rep (a,b)
  List :: Rep a -> Rep [a]
-}
equalRep :: Rep a -> Rep b -> Maybe (Equal a b)
equalRep Int        Int        = Just Eq
equalRep Bool       Bool       = Just Eq
equalRep (Prod a b) (Prod c d) =
  case equalRep a c of
    Nothing -> Nothing
    Just Eq -> case equalRep b d of
                 Nothing -> Nothing
                 Just Eq -> Just Eq
-- alternatively we could use Monad syntax
equalRep (Prod a b) (Prod c d) =
  do { Eq <- equalRep a c
     ; Eq <- equalRep b d
     ; return Eq}
 where monad maybeM
equalRep _ _ = Nothing

maybeM = Monad (Just) bind fail
  where bind (Just x) f = f x
        fail s = Nothing

data Term :: *0 ~> *0 where
  Var   :: String -> Rep t -> Term t          -- x
  Const :: Int -> Term Int                    -- 5
  Add   :: Term ((Int,Int) -> Int)            -- (+)
  LT    :: Term ((Int,Int) -> Bool)           -- (<)
  Ap    :: Term(a -> b) -> Term a -> Term b  -- (+) (x,y)
  Pair  :: Term a -> Term b -> Term(a,b)      -- (x,y)

type Env = [ exists t . (String, Rep t, t) ]

lookupWithRepr :: Env -> Rep t -> String -> t
lookupWithRepr [] r1 x1 = error "variable not found"
lookupWithRepr (Ex(x,r,v):ts) r1 x1
  = if eqStr x x1
       then case equalRep r r1 of
              Just Eq -> v
              Nothing -> lookupWithRepr ts r1 x1
       else lookupWithRepr ts r1 x1

uncurry f (x,y) = f x y

eval :: Term t -> Env -> t
eval (Var x r)  env = lookupWithRepr env r x
eval (Const i)  _   = i
eval Add        _   = uncurry (+)
eval LT         _   = uncurry (<)
eval (Ap f p)   env = (eval f env) (eval p env)
eval (Pair a b) env = (eval a env, eval b env)

--------------
-- Exercise 25
--------------

opt:: Term a -> Term a
opt (Ap Add (Pair (Const n) (Const m)))
  -- constant folding
  = Const(n+m)
opt (Ap Add (Pair (Const 0) x))
  -- law: (0 + x)=x
  = x
opt (Ap Add (Pair x (Const 0)))
  -- law: (x + 0)=x
  = x
opt (Ap x y) = Ap (opt x) (opt y)
opt (Pair x y) = Pair (opt x) (opt y)
opt x = x

-- can you make opt work for (x + (3 + -3)) or (1 + (2 + 4))

stagedEvalTerm :: Term a -> Code a
stagedEvalTerm (Const x) = lift x
stagedEvalTerm Add = [| add |]
  where add (x,y) = x+y
stagedEvalTerm LT = [| less |]
  where less (x,y) = x < y
stagedEvalTerm (Ap f x) = [| $(stagedEvalTerm f) $(stagedEvalTerm x) |]
stagedEvalTerm (Pair x y) = [|($(stagedEvalTerm x),$(stagedEvalTerm y))|]


optStagedEvalTerm x = stagedEvalTerm(opt x)
```

A Tutorial on Object-Oriented Functional Programming

Horia F. Pop

Department of Computer Science
Babes-Bolyai University
Cluj-Napoca, Romania
`hfpop@cs.ubbcluj.ro`

Abstract. This paper forms the notes of a two-hours lecture introducing Object-Oriented Functional Programming with Lisp as a support language. We start by remembering the key concepts of functional programming, imperative programming and object-oriented programming. We continue with a discussion of object-orientedness in Lisp, including Lisp packages, Lisp data structures and CLOS – Common Lisp Object System. We then remind the alternate approach, of functional paradigm in C++. A suggestion for a lab session follows.

1 Introduction

This paper follows from a two-hours lecture on Object-Orientedness in Functional Programming with Lisp as a support Language. Why the choice for Object-Oriented Functional Programming? There are a few essential reasons, especially from the point of view of young students programming background. While the declarative programming paradigm is inherently closer to the human natural thinking process, it is a tradition to form the programming education using the imperative and then the object-oriented programming paradigm.

While teaching both programming paradigms we should underline as well the similarities, and concentrate on a set of common concerns and themes rather than a list of distinctions from other paradigms. For example, it is important to note that almost all rules of good programming that should be part of any lecture of fundamentals of imperative programming, are valid as well with functional programming. Once students understand this, the learning gap between the paradigms starts to close. On the other side, while students may feel that the rules of good programming may appear artificial in the the imperative paradigm, they notice that the same rules are quite natural in the declarative paradigm.

This paper assumes that the reader is effective in the imperative and object-oriented programming paradigm, and familiar with the Lisp programming language. We aim at demonstrating how the students may use their object-oriented programming knowledge in the framework of declarative programming.

2 Programming Paradigms

Functional programming

Functional and imperative programming are two main paradigms taught during undergraduate studies. But, while the imperative programming style emphasizes changes in state, the functional programming style has a few important and distinct properties:

– It treats computation as the evaluation of mathematical functions;
– It avoids state and mutable data;
– It emphasizes the application of functions.

Key concepts, important in functional programming, include higher-order and first-class functions, closures, recursion and lambda calculus. The latter forms the foundation for most models of functional programming.

Among the most used functional languages we recall here APL, Erlang, Haskell, Lisp, ML, Scheme.

Imperative programming

Imperative programming describes computation as statements that change a program state. As such, imperative programs lead to a sequence of commands for the computer to perform. On the contrary, functional programs are NOT a sequence of statements and have NO global states. As well, logical programs are thought of as defining WHAT is to be computed, rather than HOW the computation is to take place.

Object-oriented programming

The object-oriented programming paradigm uses objects to design applications and computer programs. Object-oriented programming uses several techniques from previously established paradigms and it was not commonly used in mainstream software application development until 1990. Some of the main properties of object-oriented programming are inheritance, modularity, polymorphism, encapsulation. Through its properties, object-oriented programming addresses the problem of quality in software by strongly emphasizing modularity.

Inheritance. Inheritance is a way to form new classes (instances of which are called objects) using classes that have already been defined. The new classes (derived classes) inherit attributes and behavior of the pre-existing classes, which are referred to as base classes. Inheritance is intended to help reuse existing code with little or no modification.

A few applications of inheritance follow.

Specialization: Create specializations of existing classes or objects. This feature is called subtyping. The new class or object has data or behavior aspects that are not part of the inherited class.

Overriding: A class or object may replace the implementation of an aspect that it has inherited. This feature is called overriding. The question on which version of the behavior does code from the inherited class see may be raised here: is the one that is part of its own class, or the overriding behavior?

Code re-use: One of earliest motivations: allow a new class to re-use code which already existed in another class. This feature is called implementation inheritance.

Modularity. A module is a software entity that groups a set of subprograms and data structures; that can be compiled separately; and that provides a separation between interface and implementation.

The possibility of creating modules is one of the important features of object-oriented programming.

Polymorphism. Polymorphism allows a single definition to be used with different types of data. A few examples follow:

– a polymorphic function definition can replace several type-specific ones;
– a single polymorphic operator can act in expressions of various types;
– ad-hoc polymorphism: range of types is finite and combinations must be specified individually prior to use;
– parametric polymorphism: all code is written without mention of any specific type.

Encapsulation. Encapsulation is also called information hiding. Through encapsulation the programmer hides design decisions in a computer program that are most likely to change. As such, other parts of program are protected from change if design decision is changed. Encapsulation reduces software development risk by shifting the code's dependency on an uncertain implementation (design decision) onto a well-defined interface.

3 Using Packages in Lisp

Lisp packages introduce a mechanism similar to namespaces, as they allow the definition of Lisp entities local to namespaces [1,2]. A package is a collection of Lisp symbols. The following examples will illustrate the main features of the package mechanism:

– Defining and using packages;
– Defining Lisp entities local to a package and using them in a different package;
– Importing symbols from another package, to be used without a package prefix;
– Exporting symbols, i.e. making them accessible to users of the package.

We start by creating two packages, `one` and `two`.

```
> (make-package :one)
#<Package "ONE">

> (make-package :two)
#<Package "TWO">
```

We open package `one` and define the function `foo` internal to `one`.

```
> (in-package one)
#<Package "ONE">

> (defun foo ()
    "This is one-foo")
FOO
```

Similarly, we open package `two` and define a different function `foo`, internal to `two`.

```
> (in-package two)
#<Package "TWO">

> (defun foo ()
    "This is two-foo")
FOO
```

In package `two`, we run the function `foo`. Of course, it is the function internal to the package `two`.

```
> (in-package two)
#<Package "TWO">

> (foo)
"This is two-foo"
```

In package `one`, we run the function `foo`. Of course, this is the function defined in package `one`. But, if needed, we may run as well the function `foo` defined in package `two`. See the package prefix notation `two::`.

```
> (in-package one)
#<Package "ONE">

> (foo)
"This is one-foo"

> (two::foo)
"This is two-foo"
```

But we do not need to carry over the prefix notation. The `import` function allows us to import in the current package a symbol internal to another package, and to use this symbol directly, i.e. with no package prefix. The `unintern` function reverses the effect of `import`.

```
> (in-package one)
#<Package "ONE">

> (defun baz ()
    "This is one-baz")
BAZ

> (in-package two)
#<Package "TWO">

> (import 'one::baz)
T

> (baz)
"This is one-baz"

> (unintern 'baz)
T
```

One of the problems of `import` is that you only import the stated symbol. However, this may not be what you actually need. A typical example is the Lisp class, whose definition leads to the definition of class attributes and methods as separate, associated symbols. When importing the class, you actually mean to import the class attributes and methods as well, but this is not what you get. In order to explicitly indicate what symbols are to be exported from a package, the `export` function is to be used.

```
> (in-package one)
#<Package "ONE">

> (setf x 1)
1

> (export '(x y z))
T
```

All the symbols declared as exported with `export` will be made available in another package when using the `use-package` function.

```
> (in-package two)
#<Package "TWO">

> (use-package 'one)
T

> x
1
```

Now let us assume that both packages `one` and `two` export the same symbols, `x`, `y` and `z`.

```
> (in-package one)
#<Package "ONE">

> (export '(x y z))
T

> (in-package two)
#<Package "TWO">

> (export '(x y z))
T
```

We may need to use both packages in package `three`. Using `use-package` as before creates a problem: both packages `one` and `two` export the same list of symbols, so we need a precedence mechanism in order to be able to access these symbols. Lisp maintains a *shadowing symbols list*, which is a list of symbols that override any symbol that would become visible as a result of `use-package`. There are two functions available: `shadow`, which shadows an internal symbol, and `shadowing-import`, which shadows a symbol from another package.

```
> (in-package three)
#<Package "THREE">

> (shadow 'x)
T
> (shadowing-import 'one:y)
T
> (shadowing-import 'two:z)
T

> (use-package 'one)
T
> (use-package 'two)
T
```

Now, after the two `use-package` calls, the visible symbols are the local `x`, `y` from `one`, and `z` from `two`.

4 Data Structures in Lisp

We are going to illustrate here the Common Lisp and CLOS features for creating data structures [1,2,3].

The `defstruct` macro allows the user to create and use aggregate data types with named elements, like structures or records.

For example, assume you deal with space ships in a two-dimensional plane. You need to represent a space ship by a Lisp object of some kind:

– A ship
– Its position (x and y coordinates)
– Its speed (along the x and y axes)
– Its mass

One possible solution is to use a list and CAR and CDR type of functions for access:

```
(setf ship1 (list x-position y-position x-speed y-speed mass))
```

In order to access, for example, the `y-speed` element of a ship we may use `(cadddr ship1)`.

A more elegant solution is to use the `defstruct` macro.

```
  (defstruct ship
    x-position
    y-position
    x-speed
    y-speed
    mass
  )
```

When calling `defstruct`, a few functions are created:

– Access functions, for example `(ship-x-position ship)`, which returns the `x-position` of the ship
– The symbol, `ship`, which holds the name of a data type; `(typep x 'ship)` is true if `x` is a `ship`
– The function, `(ship-p arg)`, which returns true if `arg` is a `ship`
– The constructor, `(make-ship)`, which creates a data structure with five components, suitable for use with the access functions
  • (setq ship2 (make-ship))
  • (setq ship2 (make-ship :mass *default-ship-mass* :x-position 0 :y-position 0))
– the copier function `(copy-ship ship)` creates a new `ship` object that is a copy of the given object
– the setter function: `(setf (ship-x-position ship2) 100)`

The value of a variable of type `ship` may be printed as `#S(ship x-position 0 y-position 0 x-speed nil y-speed nil mass 170000.0)`

This approach deserves a few comments. One of the major issues with students learning imperative and object-oriented programming is the difference between what you *can* do and what you *may* do. A data structure is created starting from a record, seen as an association of variables sharing the same purpose. While students are explicitly learned, for example, to access attributes of abstract data types strictly through defined functions, they still tend to use direct access, allowing the user to break the logic of the abstract data type.

The `defstruct` construct does much more than to simply define a record-type data structure. Access functions and setter functions are created for each attribute, highlighting the functional aspect of the access to the data structure. Functions are made available to test whether a symbol is associated an instance of the data structure, and to copy instances of a data structure.

The format of `defstruct` is:

```
(defstruct
  (name option-1 ... option-m)
  doc-string
  slot-description-1
  slot-description-2 ...
  slot-description-n
)
```

The format of each slot follows:

```
(slot-name
    default-init
  slot-option-name-1 slot-option-value-1 ...
  slot-option-name-k slot-option-value-k
)
```

Note that the fields of a data structure are called ‘slots’ in Lisp. Usually, no options or slot options are needed. If the options are missing, the parentheses around the name are not required.

There are two types of slot options, namely `:type` and `:read-only`. As well, an initial slot value may be given. For example, the `ship` data structure may be defined as follows:

```
(defstruct ship
  (x-position 0.0 :type short-float)
  (y-position 0.0 :type short-float)
  (x-speed 0.0 :type short-float)
  (y-speed 0.0 :type short-float)
  (mass *default-ship-mass* :type short-float :read-only t)
)
```

Among the possible `defstruct` options, here are the most widely used:

:conc-name prefix – specifies the prefix to be used for names of access functions
:constructor symbol – specifies the name of the constructor function
:copier symbol – specifies the name of the copier function
:predicate symol – specifies the name of the type predicate
:include structure – used to build a new structure as an extension of an existing structure

A few examples follow:

Example 1

We create the `door` data structure, having three attributes: `knob-color`, `width`, and `material`. We then initialize the symbol `my-dooor` to be a `door` with a red knob color and a width of 5.0.

```
(defstruct door knob-color width material)
(setq my-door (make-door :knob-color 'red :width 5.0))
```

Then we get and set the values of different attributes of `my-door`, using the getter and setter functions made available by the `defstruct` construct.

```
(door-width my-door) => 5.0
(setf (door-width my-door) 43.7)
(door-width my-door) => 43.7
(door-knob-color my-door) => red
```

Example 2

We create the `person` data structure, with the attributes `name`, `age`, and `sex`. We then define the `astronaut` data structure to extend the `person` data structure, and the new attributes `helmet-size` and `favorite-beverage`, this latter one initialized to `tang`. We set, as well, the prefix of all the `astronaut` functions to `astro-`.

```
(defstruct person name age sex)

(defstruct
    (astronaut (:include person) (:conc-name astro-))
  helmet-size
  (favorite-beverage 'tang))
```

We initialize `x` to be an astronaut named buzz, aged 45, male, with a helmet size of 17.5.

```
(setq x (make-astronaut
  :name 'buzz :age 45 :sex male :helmet-size 17.5))
```

Since the `astronaut` data structure is an extension of `person`, both functions `person-name` and `astro-name` will return buzz. As well, we check the favorite beverage, which is, of course, `tang`, since it has not been modified in any way.

```
(person-name x) => buzz
(astro-name x) => buzz
(astro-favorite-beverage x) => tang
```

Example 3

We define a `point` as a data structure with three attributes: the three coordinates of a point in space.

```
(defstruct point x y z)
```

We define two functions to operate with points: `distance-from-origin` and `reflect-in-y-axis`. Their meaning is self-explanatory.

```
(defun distance-from-origin (point)
  (let*
    ((x (point-x point))
     (y (point-y point))
     (z (point-z point))
    )
    (sqrt (+ (* x x) (* y y) (* z z)))
  )
)
(defun reflect-in-y-axis (point)
  (setf (point-y point) (- (point-y point)))
)
```

We set `my-point` to be a `point` with the coordinates `(3, 4, 12)`. We then verify that `my-point` is, indeed, a `point`.

```
> (setf my-point (make-point :x 3 :y 4 :z 12))
#S(POINT X 3 Y 4 Z 12)
> (type-of my-point)
POINT
```

We demonstrate the use of the two functions on our `my-point`.

```
> (distance-from-origin my-point)
13.0
> (reflect-in-y-axis my-point)
-4
```

We make sure that `reflect-in-y-axis` has indeed changed the value of `my-point`, and set `a-similar-point` to be a structure with attributes of the same values.

```
> my-point
#S(POINT X 3 Y -4 Z 12)
> (setf a-similar-point #s(point :x 3 :y -4 :z 12))
#S(POINT X 3 Y -4 Z 12)
```

The function `equal`, when used with compound objects (structures, instances, etc), uses `eq` to determine arguments equality. As such, `equal` will return `NIL` in our case.

```
> (equal my-point a-similar-point)
NIL
```

On the other side, `equalp` returns true if the arguments are `equal`, or if they have components that are of the same type as each other and if those components are `equalp`. As such, `equalp` will return `T` in our case.

```
> (equalp my-point a-similar-point)
T
```

5 Common Lisp Object System

Common Lisp Object System [1,2,3] is a set of operators for implementing symbolic object-oriented programming, available as a mix of declarative (`defclass`) and functional programming (`defmethod`). Similar functionality can been achieved by structures and functions in Lisp.

CLOS allows to define differently the same methods for different objects. Each object has got different slots and properties. An object can inherit its properties from its superclass. CLOS allows multiple inheritance with the precedence list. The objects properties are manipulated by methods.

Classes

A class is an object that determines the structure and behavior of a set of other objects, called its instances. A class can inherit structure and behavior from other classes. We distinguish here subclasses and superclasses. Classes are represented by objects that are themselves instances of classes. The class of the class of an object is called the metaclass of that object. The class precedence list is a total ordering on the set of the given class and its superclasses. This list is relevant for determining the methods execution order.

The macro `defclass` is used to define a new named class. A few items are required:

– The name of the new class; proper name for newly defined classes;
– A list of direct superclasses of new class;
– A set of slot specifiers, each including the slot name and slot options;
– A set of class options.

```
(defclass class-name ({superclass}*)
  ((slot-name [slot-option])
   (slot-name [slot-option])
    . . .
   (slot-name [slot-option]))
  [class-option]
)
```

The slot options and class options may be used for different purposes, such as:

- To supply a default initial value form for a slot;
- To name methods for generic functions that are automatically generated for reading or writing slots;
- To specify whether a slot is shared by instances of the class or whether each instance of the class has its own slot;
- To supply initial arguments and argument defaults, used in instance creation;
- To indicate that the metaclass is to be other than default;
- To indicate the expected type for value stored in slot;
- To indicate documentation string for slot.

The slots of an object are determined by the class of the object. Each slot has a name and can hold a value. The name is a symbol syntactically valid for use as a variable name. The most important slot options follow:

:initarg – defines the key to pass an initial value to make-instance;
:initform – gives the default initial value form for a slot;
:allocation :instance – declares a slot to be local, i.e. visible to exactly one instance);
:allocation :class – declares a slot to be shared, i.e. visible to more than one instance of a given class and its subclasses);
:reader and **:accessor** – specify the names of the access functions for the slot, generated by `defclass`.

There is, as well, the primitive function `slot-value`, used alone as slot reader, or together with `setf` as slot accessor.

A few class functions are available:

- The function `class-name` takes a class object and returns its symbolic name.
- The function `find-class` takes a symbol and returns the class the symbol names.
- The function `class-of` returns the class of which the argument is a direct instancer.

The following examples illustrate the functions `class-name` and `find-class`. These illustrative examples are inspired from [3].

```
> (defclass point () (x y z))
#<STANDARD-CLASS POINT 275B78DC>

> (find-class 'point)
#<STANDARD-CLASS POINT 275B78DC>

> (class-name (find-class 'point))
POINT
```

Following, let us illustrate the functions `class-of`, `type-of` and `typep` in this context.

```
> (setf my-point (make-instance 'point))
#<POINT 205FA53C>

> (type-of my-point)
POINT

> (class-of my-point)
#<STANDARD-CLASS POINT 275B78DC>

> (typep my-point (class-of my-point))
T
```

The function `type-of` returns the symbolic class name of the class instance, while the function `class-of` returns the class object. As well, the predicate function `typep` uses the class object for class comparison.

```
> (class-of (class-of my-point))
#<STANDARD-CLASS STANDARD-CLASS 20306534>
```

The last example shows that classes are instances of other classes. The class `standard-class` of which `point` is an instance, is called the metaclass of `my-point`.

Objects creation and initialization

The generic function `make-instance` creates and returns a new instance of a class:

```
(make-instance class-name {arg-name init-value}*)
```

– The first argument is class or name of class;
– The remaining arguments are the initialization argument list, i.e. a list of alternating initialization argument names (defined with the `:initarg` slot option) and initial values.

As an example, we define the class `point`, and create the instance `my-point`:

```
> (defclass point () (x y z))
#<STANDARD-CLASS POINT 2060C12C>

> (setf my-point (make-instance 'point))
#<POINT 205FA53C>

> (type-of my-point)
POINT
```

Write the function `set-point-values` to assign values for the three coordinates of a point:

```
> (defun set-point-values (point x y z)
    (setf (slot-value point 'x) x
          (slot-value point 'y) y
          (slot-value point 'z) z
    )
  )
SET-POINT-VALUES

> (set-point-values my-point 3 4 12)
12
```

Write the function `distance-from-origin` to determine and return the Euclidean distance from a point to the origin of coordinates:

```
> (defun distance-from-origin (point)
    (with-slots (x y z) point
      (sqrt (+ (* x x) (* y y) (* z z)))
    )
  )
DISTANCE-FROM-ORIGIN

> (distance-from-origin my-point)
13.0
```

As a remark, we have used in this last fragment, macro `with-slots` to ease the repeated call to `slot-value`. This construct is found with most of the imperative object-oriented languages as well.

As another example, let us consider the class `point`, with accessor functions `point-x` and `point-y` for `x` and `y`, reader function `point-z` for `z`. The key `:x` for the initial value of slot `x` is set, the initial value of `y` is set to 3.14159, and the slot `z` is declared as a class slot.

```
> (defclass point ()
    ((x :accessor point-x :initarg :x)
     (y :accessor point-y :initform 3.14159)
     (z :reader point-z :allocation :class)))
#<STANDARD-CLASS DAFT-POINT 21DF867C>
```

Since `z` is a class slot, we may assign it a value by using `make-instance` as suggested by the first statement below. The second statement creates the class instance `my-point`, with 19 as the initial value for slot `x`.

```
> (setf (slot-value (make-instance 'point) 'z) 42)
42
> (setf my-point (make-instance 'point :x 19))
#<DAFT-POINT 205F264C>
```

Now we verify that `z` is a class slot and `y` is a local slot. As such, the value `999` assigned below to `(point-y temp)` is not passed to `my-point`, while the value `0` assigned below to `(slot-value temp 'z)` is indeed passed to `my-point`.

```
> (list
  (point-x my-point)
  (point-y my-point)
  (point-z my-point)
)
(19 3.14159 42)
> (let ((temp (make-instance 'point)))
    (setf (point-y temp) 999 (slot-value temp 'z) 0))
0
> (list
  (point-x my-point)
  (point-y my-point)
  (point-z my-point)
)
(19 3.14159 0)
```

5.1 Inheritance

A class can inherit methods, slots, and some `defclass` options from its superclasses. Any method applicable to all instances of a class is also applicable to all instances of any subclass of that class.

There are two cases of slots inheritance, as follows:

– Only one class among C and its superclasses defines a slot with a given slot name. The slot characteristics are determined by the slot specifier of the defining class.
– More classes can define a slot with a given name. Only one slot with the given name is accessible in an instance of C, and the characteristics of that slot are a combination of the several slot specifiers.

All the slot specifiers for a given slot name are ordered from the most specific to the least specific. The slot allocation is controlled by the most specific slot specifier. The default initial value form for a slot is the value of the `:initform` slot option in the most specific slot specifier that contains a value. The contents of a slot will always be of type `(and T1 ... Tn)`, where `Ti` are the slot types of ancestors.

The set of initialization arguments that initialize a given slot is the union of initialization arguments declared in `:initarg` slots of the ancestors.

The slot documentation string is the value of `:documentation` slot option in most specific slot specifier that contains a documentation string.

As an example, we define `Class1` with slots `Slot1` and `Slot2`. We then define the class `Class2`, as a subclass of `Class1`. The slots `Slot1` and `Slot2` are redefined, and a new slot, `Slot3` is added.

```
(defclass Class1 ()
  ((Slot1 :initform 5.4 :type number)
   (Slot2 :allocation :class)
  )
)

(defclass Class2 (Class1)
  ((Slot1 :initform 5 :type integer)
   (Slot2 :allocation :instance)
   (Slot3 :accessor Class2-Slot3)
  )
)
```

5.2 Changing a Class

In order to change the definition of a particular class, you simply evaluate a new `defclass` form. This takes the place of the old definition, and the existing class object is updated. All instances of the class, and recursively, its subclasses, are updated to reflect the new definition. This feature is actually useful during application development. In order to change the class of a particular instance, use the following form:

```
(change-class obj newclass :arg val ...)
```

5.3 Defining Methods

The traditional way

We may write functions that take as parameters class instances. The key issue is to be able to discriminate among different classes and, actually, to run different functions in this manner. We will need to use the `typecase` macro:

```
(typecase keyform
  (type form ... form)
  (type form ... form)
  ;; [ etc etc etc ]
  (t form ... form)
)
```

The `keyform` is evaluated to produce a test-key. The clauses are evaluated in the given order. If the test-key is of the `type` mentioned as the first element of the clause, then all the `forms` of the clause are evaluated in the given order and the result of the last evaluation is the returned result.

As an example, consider the above definition of classes `Class1` and `Class2`.

```
> (defun my-assign (obj n1 n2)
    (typecase obj
      (Class1 (setf (slot-value obj 'Slot1) (* n1 n2)))
      (Class2 (setf (slot-value obj 'Slot1) (+ n1 n2)))
    )
  )
MY-ASSIGN
```

The method `my-assign` will assign to `Slot1` of an instance of `Class1` the product of the two given numbers, and to `Slot1` of an instance of `Class2` the sum of the two given numbers.

The `defmethod` macro

The `defmethod` associates a body of code with the function name but that body may only be executed if the types of arguments match the pattern declared by the lambda list.

```
    (defmethod name list body)
```

The elements of the list may either be `variable` (denoting method arguments) or `(variable class-specializer)` (denoting the class of the argument).

In the following example, the method `Square`, defined for a `number X`, returns the square of that number. But the method `Square`, defined for a `string S` returns a string holding two successive copies of `S`.

```
      (defmethod Square ((X number)) (* X X))

    (defmethod Square ((S string)) (format nil "~A~A" S S))
```

As a particular case of specializer, we may use the `eql` specializer. The specializing class name is thus replaced by a list whose first element is `eql` and whose second value is any lisp form. The form is evaluated at the same time as `defmethod`, and the corresponding argument must be `eql` to the result of the evaluation. We should remark that an `eql` method is more specific than one specializing on classes.

As an example, we write two methods `isprime`: one to run with the argument `1`, and the other, to run with any `number` argument:

```
    (defmethod isprime
      ((x (eql 1)))
      (format t "Number 1 is neither prime nor compound")
    )
    #<STANDARD-METHOD ISPRIME NIL ((EQL 1)) 2060E57C>

    > (defmethod isprime
      ((x number))
      ...
      ;; find whether x is prime and return T or NIL
      ...
    )
```

```
#<STANDARD-METHOD ISPRIME NIL (NUMBER) 2061EEF4>

> (isprime 1)
Number 1 is neither prime nor compound
NIL
```

5.4 Before/After Methods

CLOS distinguishes among more types of methods:

- primary methods (no qualifier): only the single most specific method is executed;
- `:around` methods: the most specific method is executed;
- `:before` methods: all the methods are executed in least-specific to most-specific order;
- `:after` methods: all the methods are executed in most-specific to least-specific order.

The purpose of before/after methods is to allow for side effects functions to be run before or after the main method fires. As opposed to the main method inheritance, based on which only one method fires, in the case of before/after methods all applicable methods fire.

All `:around` methods run before any other methods run. A less specific `:around` method runs before a more specific primary method.

The execution of `before` and `after` methods is done based on the following succession of steps:

1. Preliminaries:
   (a) determine the applicable methods;
   (b) partition them into separate lists according to their qualifier;
   (c) if no applicable primary method is found, then signal error;
   (d) sort each list into order of specificity;
2. Execute the most specific `:around` method and return the result;
3. If an `:around` method invokes `call-next-method`, execute the next most specific `:around` method;
4. If no `:around` methods are found at step (2) or no further `:around` methods are found at step (3):
   (a) run all the `:before` methods, ignoring return values and forbidding calls to `call-next-method` or `next-method-p`;
   (b) execute the most specific primary method and return whatever that returns;
   (c) if a primary method invokes `call-next-method`, execute the next most specific primary method;
   (d) if a primary method invokes `call-next-method` but no further primary methods are found, then signal error;
   (e) run all the `:after` methods, in reverse order, ignoring their return values and forbidding calls to `call-next-method` or `next-method-p`.

In the following example we define a few classes: Graphical-Object, Circle and Rectangle, subclasses of Graphical-Object, and Square, a subclass of Rectangle.

```
(defclass Graphical-Object ()
  ((Color :type symbol :initform 'red)
   (Position :type symbol :initform 'here)
  )
)
(defclass Circle (Graphical-Object)
  ((Radius :accessor Radius :type number :initform 0)
  )
)
(defclass Rectangle (Graphical-Object)
  ((Width  :accessor Width  :type number :initform 0)
   (Height :accessor Height :type number :initform 0)
  )
)
(defclass Square (Rectangle))
```

We use again the previous definitions of Square for numbers and strings.

```
(defmethod Square ((X number)) (* X X))
(defmethod Square ((S string)) (format nil "~A~A" S S))
```

We define the method Area to be used with circles, rectangles and squares. The last definition below produces an error message, and is used only when called with an argument of a type other than Circle, Rectangle, or Square.

```
(defmethod Area ((C Circle))
  (* pi (Square (Radius C))
)
(defmethod Area ((R Rectangle))
  (* (Width R) (Height R))
)
(defmethod Area ((Sq Square))
  (Square (Width Sq))
)
(defmethod Area (Arg)
  (error "Area only defined for Circles, Rectangles, Squares")
)
```

We define the method Height for a Square, to return its width, and actually enforce the square property.

```
(defmethod Height ((Sq Square))
  (Width Sq)
)
```

We define the method `Color` for a `Graphical-Object`. The first definition is for the reader, and the second is for the accessor.

```
(defmethod Color ((Obj Graphical-Object))
  (slot-value Obj 'Color)
)
(defmethod (setf Color)
  (New-Color (Obj Graphical-Object))
  (setf (slot-value Obj 'Color) New-Color)
)
```

We define an `:after` method for the accessor method `Radius` of a circle: as soon as the radius of a circle will have been set, its area will have been computed as well.

```
(defmethod (setf Radius) :after ((New-Radius number) (C Circle))
  (setf (Area C) (* pi (Square New-Radius)))
)
```

We finally define a `:before` method for the reader method `Area` of a rectangle: right before computing the area of a rectangle, verify that the `Height` slot is bound. If the slot is not bound, then bind it to the width of the rectangle. Only after this assignment, the `Area` method is called.

```
(defmethod Area :before ((R Rectangle))
  (unless
    (slot-boundp R 'Height)
    (setf (Height R) (Width R))
  )
)
```

6 The Functional Paradigm in C++

The purpose of this tutorial so far has been allow a reader familiar with the object-oriented programming and functional programming paradigms to get an introduction to object-oriented functional programming. That is, the ability to use the object-oriented paradigm in a functional context.

The reciprocal may as well be approached. One may be interested to study the ability to use concepts and features of the functional paradigm in an imperative, or object-oriented context.

A package allowing functional programming support for C++ is `FC++`, available at the address `http://sourceforge.net/projects/fcpp` [4,5].

The package provides complete support for polymorphism. FC++ polymorphic higher-order functions can take other polymorphic functions as arguments and return polymorphic functions as results. The user can define own higher-order polymorphic functions.

The package contains, among other features:

– Infinite (”lazy”) lists;
– Useful higher-order functions (like map, compose, etc.);
– Reference-counting facility to be used to replace C++ pointers;
– Many common logical and arithmetic operators in a form that can be used with higher-order functions.

7 Lab Subject

The suggested subject for a lab on object-oriented functional programming is to solve Sudoku and Einstein houses problems using Constraints Satisfaction Programming (CSP) [6].

The purpose is to write:

– a CSP class to model a CSP problem with n variables and domains of values, with constraints and Backtracking function;
– a Sudoku descendant class to model a Sudoku board and use the CSP functionality in the parent class to fill the board;
– an Einstein descendant class to model the Einstein houses problem and use the CSP functionality in the parent class to get a solution.

Again, the point of the lab subject is to implement the Backtracking function together with the required helping functions, in one class, and to allow particular modeling of the method in descending classes, in order to solve the Sudoku and Einstein problems. An understanding of CSP is a bonus, but this is not actually required to approach the lab subject.

CSP formalization

A Constraints Satisfaction Programming problem is defined as follows [6]:

– A set of variables, x_1, x_2, ..., x_n are defined;
– A set of finite domains D_1, D_2, ..., D_n are provided for each variable;
– A set of constraints are defined; they are logical predicates on sets of variables, that constrain the legal combinations of values the participating variables may take.

A solution to a CSP problem is a set of values v_1, v_2, ..., v_n such that $v_i \in D_i$ and all the constraints are satisfied.

The Sudoku problem

The Sudoku problem is defined as follows:

– A board with nine lines and nine columns is given;
– The board is divided in nine squares sized 3x3 each;
– Each line, column and square must contain all the numbers 1, 2, ..., 9, exactly once each.

A partially filled board is given. You should fill the board. For references, see the web address `http://www.websudoku.com`.

The Einstein houses problem

The Einstein houses problem is defined as follows:

- There are three houses, three nationals, three animals, three cigarette brands;
- The Englishman lives in the first house on the left;
- In the house immediately on the right of that housing the wolf, they smoke Lucky Strike;
- The Spaniard smokes Kent;
- The Russian has a horse.

You are required to determine the configuration of the houses and to answer to the following two questions:

- Who smokes LM?
- Who has the dog?

8 Concluding Remarks

The Common Lisp Object System is a complex and powerful implementation of the object-oriented paradigm in Lisp. The purpose of this paper has been to illustrate that the declarative paradigm, actually closer to the human thinking process, may, as well, be viewed from a different perspective, the young students may be more familiar with: the use of abstract data types and object oriented programming.

For the homework a topic on Constraints Satisfaction Programming (CSP) has been chosen because it is closer to many paradigms, between algorithms and problem solving, on one hand, and logic programming and artificial intelligence on another hand. On the other hand, the backtracking algorithm is actually studied by first year undergraduate students, or, in many cases, even in high school.

The reader is invited to work on the lab homework, using his/her experience on object-oriented programming, but thinking in this new paradigm, of functional programming.

References

1. Pitman, K.M. (ed.): The Common Lisp Hyperspec (1996), `http://www.lispworks.com/reference/HyperSpec/Front/index.htm`
2. Steele Jr., G.L.: Common Lisp the Language, 2nd edn. (1990), `http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/clm.html`
3. Levine, N.: Fundamentals of CLOS. International Lisp Conference, New York City (2003)
4. McNamara, B., Smaragdakis, Y.: Functional Programming in C++. In: The 2000 International Conference on Functional Programming, Montreal, Canada, September 18–20 (2000)
5. McNamara, B., Smaragdakis, Y.: FC++: Functional Programming in C++, `http://sourceforge.net/projects/fcpp`, `http://sourceforge.net/projects/fcpp`
6. Bartak, R.: On-line Guide to Constraint Programming, Charles University, Prague, `http://ktiml.mff.cuni.cz/bartak/constraints`

Use Cases for Refactoring in Erlang*

Tamás Kozsik, Zoltán Csörnyei, Zoltán Horváth, Roland Király,
Róbert Kitlei, László Lövei, Tamás Nagy, Melinda Tóth, and Anikó Víg

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
{kto,csz,hz,kiralyroland,kitlei,lovei,lestat,toth_m,viganiko}@inf.elte.hu

Abstract. Tool support for refactoring provides guarantees for the preservation of the program semantics during program transformation. This paper explains how RefactorErl, a refactoring tool for the Erlang language helps the programmer raise the quality of Erlang code or make the code suitable for further changes and improvements. Many examples illustrate the seven transformations currently implemented in RefactorErl. The paper also discusses the problems the refactor tool has to face.

1 Introduction

The concept "refactoring" refers to program transformations that preserve the meaning of programs. Such transformations are often applied during software development in order to raise the quality of the code or to make the code suitable for further changes and improvements. The most well-known transformations are renaming of entities (variables, operations, modules etc.), changing the arguments of a subprogram and extracting a piece of code into a subprogram. There are refactorings that are easy to perform: if only a small fragment, for example a single compilation unit, of the program needs to be updated, the programmer can apply the changes manually without too much effort. Most refactorings, typically those that affect many modules, or modify the structure of the code significantly, are much harder to perform. Tool support for refactoring increases programmer productivity by taking over time consuming, tedious and error-prone work and by providing guarantees for the sound and consistent application of the transformations.

There are many refactoring tools available today. Many of them are integrated into popular software development environments for mainstream object-oriented languages – for functional programming the choice is not that ample. However, there are promising research results for the Erlang programming language. This paper presents some refactorings that are useful for Erlang programmers, explains the possibilities and limitations for these transformations, and describes a tool, RefactorErl, that can help the programmers in refactoring.

* Supported by GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK, Ericsson Hungary, ELTE CNL and OMAA-ÖAU 66öu2.

Erlang [1] is an eager, impure, dynamically typed functional programming language developed by Ericsson. It was designed for building concurrent and distributed fault-tolerant systems with soft real-time characteristics, like telecommunication systems. The Erlang language consists of simple functional constructs extended with message passing to manage concurrency. Erlang has a module system with export/import lists, exception handling, reflective programming facilities, preprocessing mechanism to support macros and file inclusion, and a comprehensive standard library.

Refactoring means changing the program code without changing what the code does. It is not about functional changes or fixing bugs, but about making the code better – in some respect. For instance, refactoring can take place before the introduction of a new feature. In such a situation refactoring might be necessary to make the code more suitable for the reception of the new feature. In other cases refactoring is used to meet coding conventions, to improve style and readability, or even efficiency. A refactoring tool should be able to decide whether a certain refactoring transformation is legal, that is it can be carried out in a semantics preserving way. Furthermore, the tool must also be able to perform transformations accurately, namely modifying exactly those parts of the code that should indeed be modified – not forgetting about anything and not modifying anything accidentally. For this reason the refactoring tool will analyse not only the structure of the refactored program (based on the syntactic rules of the underlying programming language), but it will also collect and use semantical information about the program.

Some features of the Erlang language are advantageous for refactoring: side effects are restricted to message passing and built-in functions, variables are assigned a value only once in their lifetime, and code is organised into modules with explicit interface definitions and static export and import lists. There are, however, some features that are disadvantageous for refactoring, e.g. the possibility to run dynamically constructed code, and the lack of programmer defined types. In order to use a refactoring tool properly, its user should be aware of the problems imposed by the limitations of static program analysis.

The rest of the paper is organized as follows. Sect. 2 presents a distributed Erlang application; this presentation also provides a brief overview of Erlang. For more information on this programming language see e.g. [1,2,7]. In Sect. 3 seven refactoring transformations are illustrated – the distributed application serves as the code body to be refactored. Sect. 4 explains in more details how RefactorErl can be used. The rules and conditions of the seven refactoring transformations are made more clear through the discussion in Sect. 5. Related work is described in Sect. 6, and finally Sect. 7 concludes the paper.

2 Erlang by Example

As a brief overview of Erlang, consider the code of a chat server in Fig. 1. This code will be used as an example for the introduction of the refactoring transformations presented in this paper.

```
-module(srv).
-export([start/1, stop/0]).
-export([connect/2, disconnect/1, send/2]).
-define(CLIENT, cli).

start(Max) -> register(chatsrv, spawn(fun() -> init(Max) end)).
stop() -> chatsrv ! stop.

connect(Srv, Nick) ->
   {chatsrv, Srv} ! {connect, self(), Nick},
   receive
      ok -> ok;
      deny -> deny
   after 1000 -> timeout
   end.

disconnect(Srv) -> {chatsrv, Srv} ! {disconnect, self()}.
send(Srv, Text) -> {chatsrv, Srv} ! {text, self(), Text}.

init(Max) -> loop([], 0, Max).

loop(Users, Nr, Max) ->
   receive
      stop -> ok;
      {connect, Pid, Nick} ->
         if  length(Users) >= Max -> Pid ! deny,
                                     loop(Users, Nr, Max);
            true                  -> link(Pid),
                                     Pid ! ok,
                                     MoreUsers = [{Nick, Pid} | Users],
                                     loop(MoreUsers, Nr, Max)
         end;
      {text, From, Text} ->
         [{Nick,_}] = lists:filter(fun({_,Pid}) -> Pid==From end,Users),
         send(Nick, Text, Nr + 1, Users),
         loop(Users, Nr + 1, Max);
      {disconnect, Pid} ->
         Remaining = lists:filter(fun({_,P}) -> P/=Pid end, Users),
         loop(Remaining, Nr, Max)
   end.

send(_, _, _, []) -> ok;
send(Sender, Text, Nr, [{Nick,Pid} | Rest]) ->
   Msg = "[" ++ integer_to_list(Nr) ++ "]" ++ Sender ++ ": " ++ Text,
   (?CLIENT):send(Pid,Msg), send(Sender, Text, Nr, Rest).
```

Fig. 1. Chat server

The code of the chat server is in the `srv` module. The module exports two functions, the unary `start` and the nullary `stop` for starting and stopping the server process (the number after the slash sign is the arity of the function). Three more functions are exported for the chat clients: `connect/2`, `disconnect/1` and `send/2`. All the other functions of the module are private to the module.

The fourth line is a macro definition: `cli` – the name of the module containing the code of the chat clients – is bound to the `CLIENT` macro. `CLIENT` will be used in the last line of the module, when the chat server sends a message to each of the chat clients using the `send/2` function defined in, and exported by, the client module. After macro expansion, the call will be in the following form: `cli:send(Pid,Msg)`.

The `start/1` function spawns a new process and registers it under the name `chatsrv`. This name – which is, in the Erlang terminology, an *atom* – can be used for sending messages to the process. The new process will evaluate the function passed as the argument to `spawn`. The nullary anonymous function used here makes it possible to pass the computation `init(Max)` lazily to the new process. Anonymous functions are called *explicit* `fun`*-expressions* in Erlang, and λ-expressions in many other functional languages. The higher-order `spawn/1` function is a built-in function, or BIF for short. Many BIFs have side effects; for example, both `spawn` and `register` modify the global program state – the process structure and the process registry, respectively.

The other four interface functions communicate with the server process using the registry. They send messages to the server process. The left argument of the message send operator `!` is the `chatsrv` atom in the case of `stop/1`, and the tuple `{chatsrv, Srv}` in the case of the functions used by the chat clients. The difference is because the chat application will be a distributed application, the server and the clients will typically run on different machines. When a client communicates to the server, it should identify the machine (the *Erlang node*) the server is running on. This is achieved by passing the `Srv` argument to the `connect/2`, `disconnect/1` and `send/2` functions. On the other hand, `stop/1` is supposed to be called on the Erlang node where the server process is running, so a local process name suffices for message sending.

After sending a message to the server, `connect/2` waits for a response. The response is supposed to come from the server process, and it can be either `ok` or `deny`. It is also possible that no response arrives in one second, and in this case `connect/2` times out. The server process knows whom to respond because the client sends its process identifier in the `connect` message. A process can find out its own identifier by calling the `self/0` BIF. This function has no side effects, but it violates referential transparency by returning a different value when called by different processes.

The main logic of the server process is coded in `loop/3`. This tail-recursive function implements the reactive behaviour of an event loop. Since tail-recursive functions are by definition compiled into loops in Erlang, this is the proper idiom for coding long or infinite loops. The event loop is initiated by the `init/1` function, and it terminates when the server receives the `stop` message from the `stop/0` function. The server maintains a list of the connected clients. For each client, a nickname and the identifier of the client process is stored; the latter is capable of identifying a process in a distributed environment. The number of connected clients is bounded by `Max`, which is passed to `loop/3` through `start/1` and `init/1`. As usual in functional programming, state (in this case the local state of the server process) is modelled by passing information explicitly as arguments and return values of functions – in this case those of `loop/3`. Functions receiving and returning state information act as state transformers. For instance, the list of clients is updated when a `disconnect` message is received. The updated list is computed by the higher-order function `filter` from the standard library module `lists`, which filters out those elements of a list that satisfy a given predicate. The predicate is passed to `filter` as an explicit fun-expression, an unnamed local function of the enclosing (clause of the) `loop/3` function. According to the scoping rules, `Pid` is a local variable of the single clause of `loop/3`, and in the meantime it is a non-local variable of the nested function definition.

The last comments are devoted to patterns. Patterns occur in formal argument lists, on the left-hand side of the pattern matching operator `=`, in branching statements like `case` and `receive` (see `loop/3` for examples) and in list comprehension expressions. Patterns can also be nested. The pattern matching expression that binds the nickname of a chat client to variable `Nick` when the client sends a text (the first expression in the sequence for processing `text` messages) is worth a look. The pattern involved is a list pattern, the only element of the list is a tuple pattern, and the tuple should have two fields. The first field is bound to `Nick`, while the second field is bound to no variable. Assuming that the received `text` message is indeed from a connected chat client, and this client process is not connected multiple times to the server, the pattern matching will not fail, viz. it will not result in a run-time error, and it will bind variable `Nick`.

3 Refactoring by Example

Before diving into the details of RefactorErl, it is worth to think over how refactoring is integrated in the programming process and how a refactoring tool is used by software developers in general. One could identify (at least) three scenarios for refactoring.

During coding. In this scenario the programmer typically (but, of course, not exclusively) performs minor refactoring transformations, often affecting a single module. To support this case, it is convenient to have the refactoring tool integrated in the used software development environment. For example, RefactorErl is integrated into Emacs, a development environment preferred by many Erlang developers, and a plug-in for Eclipse is also being implemented. In this scenario transformations carried out with the refactoring tool are usually interleaved with manual editing of the code, and the programmer expects short response times (of few seconds) from the tool. Furthermore, an undo facility is also very helpful in this case: both manual editing and refactoring transformations should be possible to undo. Such a facility is missing from the first public version of RefactorErl, but it is available in the second one.

Post-programming polishing. In this scenario programmers tend to perform many major transformations in sequence. Larger response times (few minutes, or even hours) are acceptable here, especially if the transformations affect millions of lines of code. To support this case, it might be useful to provide a scripting interface to the tool.

Before the introduction of a new feature. When an operational software component is to be extended with a novel feature, the code often needs to be refactored beforehand: to be adapted to the reception of the novel feature. Similarly to the previous scenario, in this case many major transformations are applied without intervening manual editing of the code, and larger response times are affordable.

In all three scenarios the programmers place confidence in the refactoring tool, i.e. that refactoring transformations respect the meaning of the refactored program and do not have unexpected effects – without such a confidence the tool will never gain industrial acceptance. However, the requirement that refactorings should preserve semantics is rather vague. First of all, although the possibility to refactor illegal code would be beneficial for the first scenario, it is less painful to confine ourselves to legal programs, and not require refactorings to behave well on illegal ones. RefactorErl assumes that refactored programs respect the static legality rules of Erlang (i.e. they are accepted by the compiler). Moreover, in case of certain transformations it is also assumed that the refactored program is free of dynamic errors (see Sect. 5.3). Unfortunately, in Sect. 5.1 it turns out that certain features of Erlang make it practically impossible to capture the semantics of programs by static analysis. A completely safe, conservative tool would not be sufficiently serviceable on real-world code: it would refuse to transform programs which contain certain widely used constructs that are not statically analyzable. An effective refactoring tool for Erlang should establish the right balance between safety and serviceability. To find this balance, one should experiment with real industrial code, not only with artificial examples illustrating extreme uses of the language.

In the following seven different transformations will be presented and illustrated on the chat server example. Some of the transformations are related to variables, others are related to functions or expressions. The refactorings are described in the order of their application on the `srv` module. All these transformations are implemented in, and possible to execute with, the first public version of the RefactorErl tool (the implementation of these transformations in the second version of the tool is ongoing work). In the future further refactorings will be added into RefactorErl. The currently available transformations, listed below, already cover a wide selection of topics.

Abstraction: Extract function, Merge expression duplicates.
Substitution: Eliminate variable.
Renaming: Rename variable, Rename function.
Reordering: Reorder function arguments, Tuple function arguments.

3.1 Extract Function

The seven transformations above will be used to improve the code presented in Fig. 1. The first idea is that a chat server should treat clients that terminate abruptly, without sending a `disconnect` message, similarly to those that disconnect properly. This is possible if the server process traps exit messages. To avoid code repetition, refactoring is used to make the relevant piece of code reusable. The applied transformation, *Extract Function*, encapsulates a sequence of expressions in a newly created function definition. The free variables of the selected sequence of expressions become the arguments of the function.

Fig. 2 shows how the refactoring changes the `srv` module. The sequence of two expressions managing the `disconnect` case of the `receive` in `loop/3` is extracted into function `continue_without_client`. Note that `loop/3` still remains tail-recursive.

Now it is time to add code manually to trap exit messages. Before starting the main loop, `init/1` now calls the `process_flag` BIF. The `receive` construct inside `loop/3` should also be extended with an additional branch. The necessary modifications are shown in Fig. 3.

Inline Function, the inverse of the *Extract Function* refactoring, is also a useful transformation; it will be available in the next public release of RefactorErl.

```
loop(Users, Nr, Max) ->
   receive
      ...
      {disconnect, Pid} ->
         continue_without_client(Pid,Users,Nr,Max)
   end.

continue_without_client(Pid,Users,Nr,Max) ->
   Remaining = lists:filter(fun ({_, P}) -> P /= Pid end, Users),
   loop(Remaining, Nr, Max).
```

Fig. 2. Having extracted `continue_without_client`

```
init(Max) -> process_flag(trap_exit,true), loop([],0,Max).

loop(Users, Nr, Max) ->
   receive
      ...
      {disconnect, Pid} -> continue_without_client(Pid,Users,Nr,Max);
      {'EXIT', From, _} -> continue_without_client(From,Users,Nr,Max)
   end.
```

Fig. 3. Adding code to handle clients that quit abruptly

```
loop(Users, Nr, Max) ->
   NextNr = Nr + 1,
   receive
      ...
      {text, From, Text} ->
         [{Nick,_}] = lists:filter(fun({_,Pid}) -> Pid==From end,Users),
         send(Nick, Text, NextNr, Users),
         loop(Users, NextNr, Max);
      ...
   end.
```

Fig. 4. Merging occurrences of `Nr+1`

3.2 Merge Expression Duplicates

The next transformation is somewhat similar to *Extract Function*. The idea is again the reduction of redundancy in the code. Within a function definition, multiple occurrences of the same expression should be localized, and a fresh variable should be introduced which will hold the pre-computed value of the expression. All the occurrences of the expression should than be replaced with a reference to the variable. This technique is illustrated in Fig. 4, where the occurrences of `Nr+1` in `loop/3` are replaced with the fresh variable `NextNr`.

The term "multiple occurrences of the same expression" needs some more clarification. It is not the lexical equality which is implied by "same", the requirement is more strict. A variable name occurring in the expression occurrences at a given lexical position should mean the same variable (cf. scoping rules), and these variable occurrences must have the same unique binding occurrence – put it simple, they must refer to the same value in every expression occurrence. This condition clearly holds for the merged two occurrences of `Nr+1`, since the unique binding occurrence of `Nr` is the one in the formal argument list of `loop/3`.

Another question is where to insert the binding for the fresh variable. RefactorErl chooses the earliest possible location, the first position in the enclosing function clause where all the variables of the merged expression occurrences are already bound. In the case of `NextNr`, this is the first expression in the clause body.

3.3 Eliminate Variable

The inverse of *Merge Expression Duplicates* is the elimination of a variable by replacing its applied occurrences by the bound expression. This transformation will be used to get rid of variable `MoreUsers` in the definition of `loop/3`. There is a single applied occurrence of this variable, right after its binding. Fig. 5 shows how the transformation affects the chat server.

```
loop(Users, Nr, Max) ->
   NextNr = Nr + 1,
   receive
      stop -> ok;
      {connect, Pid, Nick} ->
         if  length(Users) >= Max -> Pid ! deny,
                                     loop(Users, Nr, Max);
            true                  -> link(Pid),
                                     Pid ! ok,
                                     loop([{Nick, Pid} | Users], Nr, Max)
         end;
      ...
   end.
```

Fig. 5. Eliminating variable `MoreUsers`

3.4 Rename Variable

This transformation needs to locate each occurrence of a variable and change the used variable name at each occurrence. The difficulty lies in the scoping and visibility rules, which make it possible, for instance, to shadow variables. This means that the scope of a variable can be nested within the scope of another variable with the same name. Consider Fig. 6, which shows how this refactoring changes the name of the local variable `Pid` of `loop/3`, occurring for example in the code that handles `connect` messages. The new name of the variable is `Client`.

Within the `receive` statement in `loop/3` there are two affected branches: those responsible for `connect` and `disconnect` messages. These two branches provide two independent binding occurrences of the same variable (originally `Pid`, now `Client`). There are five applied occurrences of this variable in these two branches. The local `Pid` variable of the explicit `fun`-expression nested in `loop/3` (in the branch handling `text` messages) is left unchanged by the refactoring.

The refactoring should also be careful with the new name of the variable: this new name should not conflict with existing variables. `Client` was a fresh variable name in `loop/3`, so it caused no problems. But, for example, `Users` would have not been possible to use as the new name of the renamed variable, because there had been already a variable with the same name, and occurrences of former `Pid` would have become occurrences of this already existing variable. Another impossible renaming is to change the name `P` in the formal argument list of the anonymous function nested in `continue_without_client/4` to `Pid`. That would cause the existing reference to the non-local `Pid` become a reference to the local `Pid` in the anonymous function.

3.5 Rename Function

Functions can also be renamed, but this refactoring poses completely different questions than *Rename Variable*. Named functions can only be defined on the

```
loop(Users, Nr, Max) ->
   NextNr = Nr + 1,
   receive
      stop -> ok;
      {connect, Client, Nick} ->
         if  length(Users) >= Max -> Client ! deny,
                                     loop(Users, Nr, Max);
            true                  -> link(Client),
                                     Client ! ok,
                                     loop([{Nick,Client}|Users], Nr, Max)
         end;
      {text, From, Text} ->
         [{Nick,_}] = lists:filter(fun({_,Pid}) -> Pid==From end,Users),
         send(Nick, Text, NextNr, Users),
         loop(Users, NextNr, Max);
      {disconnect, Client}
         -> continue_without_client(Client,Users,Nr,Max);
      {'EXIT', From, _}
         -> continue_without_client(From,Users,Nr,Max)
   end.

continue_without_client(Pid,Users,Nr,Max) ->
   Remaining = lists:filter(fun ({_, P}) -> P /= Pid end, Users),
   loop(Remaining, Nr, Max).
```

Fig. 6. Renaming one of the `Pid` variables to `Client`

top level; nested in a module, but not in each other. Variables, on the other hand, are always local to function clauses. The main challenge for *Rename Function* is to locate all the call sites for a given function. Fig. 7 illustrates a very simple situation. The `send/4` function is renamed to `send_to_all/4` using the refactoring. The new name reflects the fact that this function is used to send messages to all connected chat clients.

There are three functions around with the name `send` in module `srv`. The one that is being renamed, the one in the client module (made available through the `CLIENT` macro) and `send/2` defined as an interface function of `srv`. Note that the latter two should not be modified. However, the calls to the first one needs to be adapted. There are two such calls altogether; one recursive call in the same function, and another call in `loop/3`. The new name for this function might give the idea to the programmer that "sending something to *all* clients" can be programmed with the appropriate higher-order list iteration function, `foreach`. Since no transformation is available for this very purpose, it has to be hand-coded.

```
send_to_all(Nick, Text, Nr, Users) ->
   Msg = "[" ++ integer_to_list(Nr) ++ "]" ++ Nick ++ ": " ++ Text,
   lists:foreach(fun (Pid) -> (?CLIENT):send(Pid,Msg) end, Users).
```

```
send(Srv, Text) -> {chatsrv, Srv} ! {text, self(), Text}.
...
loop(Users, Nr, Max) ->
   NextNr = Nr + 1,
   receive
      ...
      {text, From, Text} ->
         [{Nick,_}] = lists:filter(fun({_,Pid}) -> Pid==From end,Users),
         send_to_all(Nick, Text, NextNr, Users),
         loop(Users, NextNr, Max);
      ...
   end.

send_to_all(_, _, _, []) -> ok;
send_to_all(Sender, Text, Nr, [{Nick,Pid} | Rest]) ->
   Msg = "[" ++ integer_to_list(Nr) ++ "]" ++ Sender ++ ": " ++ Text,
   (?CLIENT):send(Pid,Msg), send_to_all(Sender, Text, Nr, Rest).
```

Fig. 7. Renaming the `send/4` function to `send_to_all/4`

Besides increased readability and compactness, and the gain in safety due to avoiding explicit recursion, another advantage of this function definition compared to the previous ones is that it does not compute `Msg` for every element of the list, but only once, before the iteration.

Rename Function – and all other refactorings that change the specification of a named function – becomes more laborsome if the changed function is exported from its module, because in such a situation the calls to the changed function may occur in every module of the software system. Luckily, this was not the case for `send/4`.

3.6 Reorder Function Arguments

The next refactoring is useful when the order of the arguments of a function should be altered. Suppose that one wants to add more structure to the state of the chat server process. To achieve this, the refactorings *Reorder Function Arguments* and *Tuple Function Arguments* will be applied on the functions that maintain the state through their parameters and results. First reordering will be applied, and so the two last arguments of `loop/3` will be swapped. The result can be seen on Fig. 8. The goal here is to make the `Users` list and the upper bound to its length, `Max`, neighbouring parameters of `loop/3`. The next section reveals why this transformation is useful. *Reorder Function Arguments* changes the order of the formal parameters in the definition of `loop/3`, and also the order of the actual parameters whenever `loop/3` is called: the initiating call in `init/3`, the direct recursive calls in `loop/3` itself, and the indirect recursive call in `continue_without_client`.

```
init(Max) -> process_flag(trap_exit,true), loop([],Max,0).

loop(Users, Max, Nr) ->
   NextNr = Nr + 1,
   receive
      stop -> ok;
      {connect, Client, Nick} ->
         if  length(Users) >= Max -> Client ! deny,
                                     loop(Users, Max, Nr);
            true                  -> link(Client),
                                     Client ! ok,
                                     loop([{Nick,Client}|Users], Max, Nr)
         end;
      {text, From, Text} ->
         [{Nick,_}] = lists:filter(fun({_,Pid}) -> Pid==From end,Users),
         send_to_all(Nick, Text, NextNr, Users),
         loop(Users, Max, NextNr);
      {disconnect, Client}
         -> continue_without_client(Client,Users,Max,Nr);
      {'EXIT', From, _}
         -> continue_without_client(From,Users,Max,Nr)
   end.

continue_without_client(Pid,Users,Nr,Max) ->
   Remaining = lists:filter(fun ({_, P}) -> P /= Pid end, Users),
   loop(Remaining, Max, Nr).
```

Fig. 8. Reordering the arguments of `loop/3`

The last two parameters of `continue_without_client` could also be swapped by another application of *Reorder Function Arguments.*

3.7 Tuple Function Arguments

The refactoring work aiming at changing the structure of the state in the chat server continues. The next step is to form a single unit from the list of chat clients and the bound to the length of this list. This single unit will be a tuple of two fields. The result of the application of *Tuple Function Arguments* is shown in Fig. 9.

A similar transformation could be applied on `continue_without_client`, the other state transition function of the chat server process.

In the future another refactoring will be added to RefactorErl, one which can be used to turn tuples into records. Records provide a nice way in Erlang to group related data. The advantage of records to tuples is that record field selection and record update expressions need not mention all the record fields explicitly. This results in shorter code, and – which is even more important – in improved maintainability, since the addition of further record fields requires fewer modifications in the code.

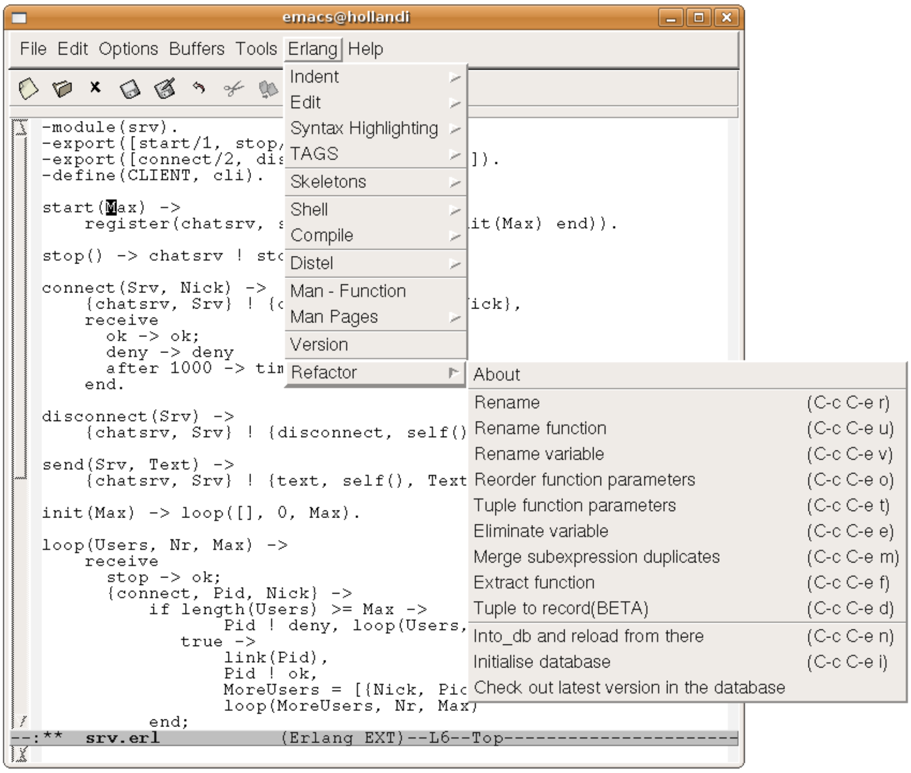
```
init(Max) -> process_flag(trap_exit,true), loop({[],Max},0).

loop({Users, Max}, Nr) ->
   NextNr = Nr + 1,
   receive
      stop -> ok;
      {connect, Client, Nick} ->
         if  length(Users) >= Max -> Client ! deny,
                                     loop({Users, Max}, Nr);
            true                  -> link(Client),
                                     Client ! ok,
                                     loop({[{Nick,Client}|Users],Max},Nr)
         end;
      {text, From, Text} ->
         [{Nick,_}] = lists:filter(fun({_,Pid}) -> Pid==From end,Users),
         send_to_all(Nick, Text, NextNr, Users),
         loop({Users, Max}, NextNr);
      {disconnect, Client}
         -> continue_without_client(Client,Users,Max,Nr);
      {'EXIT', From, _}
         -> continue_without_client(From,Users,Max,Nr)
   end.

continue_without_client(Pid,Users,Nr,Max) ->
   Remaining = lists:filter(fun ({_, P}) -> P /= Pid end, Users),
   loop({Remaining, Max}, Nr).
```

Fig. 9. Turning the first two arguments of `loop/3` into a tuple

4 RefactorErl

All the afore-mentioned refactorings are built into the RefactorErl tool. This refactoring tool is available on many different platforms [11]. It is integrated into the Emacs editor [6], which is very popular among Erlang developers. To install RefactorErl, one needs not only Emacs, but also Erlang (the tool is written in Erlang and runs in an Erlang node), Distel [3] (used to bind Erlang to Emacs) and the database management system MySQL [19]. The DBMS is necessary because the tool represents Erlang programs in a relational database, and behind the scenes the transformations manipulate the program being refactored with SQL statements. The installer for Windows takes care of all the necessary software, but for other platforms the installer requires that those are already installed.

In Emacs the refactoring tool is accessible in Erlang mode: a Refactor submenu appears in the Erlang menu located in the menu bar (see Fig. 10). The programmer can apply a refactoring transformation on the edited program code by selecting the appropriate menu item from the Refactoring submenu. Not all menu items refer to transformations: there are items to synchronize the database



```
-module(srv).
-export([start/1, stop/
-export([connect/2, dis                 ]).
-define(CLIENT, cli).

start(Max) ->
    register(chatsrv, s                 it(Max) end)).

stop() -> chatsrv ! stc

connect(Srv, Nick) ->
    {chatsrv, Srv} ! {c                 ick},
    receive
      ok -> ok;
      deny -> deny
      after 1000 -> tin
    end.

disconnect(Srv) ->
    {chatsrv, Srv} ! {disconnect, self()

send(Srv, Text) ->
    {chatsrv, Srv} ! {text, self(), Text

init(Max) -> loop([], 0, Max).

loop(Users, Nr, Max) ->
    receive
      stop -> ok;
      {connect, Pid, Nick} ->
          if length(Users) >= Max ->
                 Pid ! deny, loop(Users,
             true ->
                 link(Pid),
                 Pid ! ok,
                 MoreUsers = [{Nick, Pic
                 loop(MoreUsers, Nr, Max)
          end;
--:**  srv.erl            (Erlang EXT)--L6--Top--------------------
```

emacs@hollandi
File Edit Options Buffers Tools Erlang Help
Indent
Edit
Syntax Highlighting
TAGS
Skeletons
Shell
Compile
Distel
Man - Function
Man Pages
Version
Refactor
About
Rename (C-c C-e r)
Rename function (C-c C-e u)
Rename variable (C-c C-e v)
Reorder function parameters (C-c C-e o)
Tuple function parameters (C-c C-e t)
Eliminate variable (C-c C-e e)
Merge subexpression duplicates (C-c C-e m)
Extract function (C-c C-e f)
Tuple to record(BETA) (C-c C-e d)
Into_db and reload from there (C-c C-e n)
Initialise database (C-c C-e i)
Check out latest version in the database

Fig. 10. The user interface of RefactorErl in Emacs

representation of the refactored program with the source files. Before applying any transformation, those source files that are intended to be refactored should be loaded into the database. It is assumed that the sources satisfy the syntactic and static semantic rules of the language – only legal programs should be refactored, otherwise the result of a refactoring transformation is undefined. When loaded into the database, source files are parsed and different kinds of semantic analysis are performed. The abstract syntax tree and all the collected semantic information are stored in the database. This improves the performance of the refactoring tool when several refactoring transformations are performed one after the other: transformations are applied on the database representation, not on the source files, so the transformations need not repeatedly parse and analyse the source code. However, if the source files are edited manually between two refactorings, they have to be saved and reloaded into the database, or – alternatively – the manual changes can be undone by checking out the sources from the database. Currently the refactoring tool cannot undo transformations, the support for undo is future work.

Most refactoring transformations (all of those implemented in RefactorErl) require parameters. For example, when renaming an entity, a new name for the entity should be provided; when reordering the arguments of a function, the new order has to be specified; even variable elimination needs extra information, the identification of the variable to be eliminated. The user of the refactoring tool can pass information to the tool through the facilities of the integrating editor, Emacs. To identify a piece of code which is to be refactored, the user points the cursor on it or selects it. An entity to rename, a variable to eliminate, a function the arguments of which are to be reordered can be picked out simply by moving the cursor on an occurrence of the identifier of the entity. On the other hand, an expression to extract into a function or to a local variable should be picked out by selecting it. Any additional information is provided by the user in a separate *editor buffer*. When renaming a variable, for instance, the user first positions the cursor on a variable, and chooses the Erlang/Refactor/Rename variable menu item; then the tool prompts for the new name of the variable in a separate editor buffer. When the new name is provided, the tool decides whether the requested transformation is allowed to perform, and in the positive case performs the refactoring. The user is notified on the success or failure of the requested transformation (and, in the second case, on the cause of the failure) in a separate buffer again.

5 Transformation Rules and Conditions

This section will show the power and the limitations of a refactoring tool. Such a tool should be able to detect if an intended transformation would change the way the refactored program works, and in such cases it should refuse the execution of the transformation. However, this requirement turns out to be too restrictive in practice. In many programing languages, including Erlang, there are widely used language constructs and/or program libraries (for example reflective programming facilities) that make complete semantic analysis impossible to perform at the time of refactoring.

The semantic concepts necessary to understand and analyse the refactorings of Sect. 3 will be studied here. The side conditions (*when* a transformation is applicable) and the rules (*how* the transformation works) will be illustrated on examples which are more tricky than the `srv` module. The examples will also show situations when trade-offs should be made between preservation of program semantics and effectiveness.

5.1 Function Definitions and Function Applications

Similarly to other functional languages, function definitions in Erlang are made up of one or more function clauses (e.g. the definition of `srv:send/4` on Fig. 1 has two), which are matched against the actual arguments of a call in textual order. There are two kinds of functions: named and unnamed ones. A named function is defined in a function declaration, on the global level inside a module. Unnamed functions are local; they are defined in explicit fun-expressions, nested in other function definitions.

Refactorings working on functions (e.g. *Rename Function*, *Reorder Function Arguments* and *Tuple Function Arguments* in RefactorErl) have to find all the expressions where a given function is called. For instance, if the order of the formal arguments is changed in the function definition, then the order of the actual arguments also have to be changed, whenever the function is called. There are many ways to call a function – this will be investigated in the current section. Some forms of calls are easy to handle by static analysis, others are more subtle. Currently RefactorErl has not reached its full potential yet. It can manage "ordinary" function calls and implicit fun-expressions. Its approach to explicit fun-expressions is rather conservative: transformations on unnamed functions are simply refused. Finally, RefactorErl is powerless against the problems arising from reflective programming techniques. At the moment all it can do is to issue a warning if the refactored program uses reflection mechanisms. The discussion below, therefore, describes how the capabilities of RefactorErl can be improved in the future.

Ordinary Function Calls. Most of the function calls refer to named functions, and provide sufficient information to determine the exact signature of the called function. The signature of a (named) function consists of the name of the containing module, the declared name of the function and the arity, like in `srv:send/4`. Many function calls provide the qualified name of the called function, and the length of the actual argument list defines the arity. The call below (occurring in the `srv` module) refers to the standard library function `lists:filter/2`.

```
lists:filter(fun ({_, P}) -> P /= Pid end, Users)
```

It is allowed to use an unqualified function name in a call, if the function is declared in the same module, or it is imported with the `import` module attribute. This expression occurring in the `srv` module calls `srv:loop/3`.

```
loop([],0,Max)
```

According to the function visibility rules of Erlang, a named function can be referred in the declaring module, but it can be referred from other modules only if it is exported (with an `export` module attribute, see lines two and three in the `srv` module, in Fig. 1).

The signature uniquely identifies a function in an Erlang program. The overloading rules do not permit the declaration of a function if its name and arity clashes with the name and arity of another function declared in, or imported by, the same module. Furthermore, it is not allowed to import functions with the same name and arity from different modules. The only exception from the above rule is that it is possible to declare a function with the same name and arity as an "auto-imported built-in function". When calling an auto-imported BIF, no module name has to be supplied. Such a BIF – `spawn/1` – is also used in `srv`.

```
spawn( fun () -> init(Max) end )
```

However, if a function having the same unqualified name and arity as a BIF is called, it is obligatory to use the qualified name. Therefore, in all the cases mentioned so far it is easy to find out which function is called in a call expression. A refactoring tool can handle these cases fairly easily – so does RefactorErl.

Implicit Fun-Expressions. In higher-order functional languages functions are values that can be stored in variables, passed as arguments to other functions, or returned from function calls. Function expressions make it possible to create values representing functions. Expressions evaluating to functions (e.g. variables storing functions) can be used to call functions. The higher-order `lists:filter/2` function can be defined in the following way.

```
filter(Predicate,List) when is_function(Predicate,1)
    -> [ E || E <- List, Predicate(E) ].
```

The right-hand side of the definition is a “list comprehension”; it contains a “generator” (`E <- List`) and a “filter” (`Predicate`E). The first formal argument of `filter/2` is used as a function in the filter of the list comprehension. Furthermore, the guard of the only clause of `filter/2` tests whether `Predicate` is indeed a function with one argument (`is_function/2` is an auto-imported BIF).

When calling `lists:filter/2`, a function expression can be passed to it as the first actual argument. This happens, for example, twice in the `srv` module, when `filter` is parameterized with explicit fun-expressions (which are, as mentioned earlier, definitions of unnamed functions). Another way to create function values in Erlang is to use “implicit fun-expressions”. To select the numbers from a heterogeneous list `SomeList`, one can write the following expression.

```
lists:filter(fun erlang:is_number/1, SomeList)
```

An implicit fun-expression is created using the `fun` keyword and the signature of a named function. Similarly to ordinary function calls, the module name can be omitted when referring to imported functions and to ones declared in the same module (therefore `erlang:` can be omitted from the above fun-expression, since `is_number/1` is an auto-imported BIF). The reason why such fun-expressions are called implicit is that they are merely abbreviations of explicit fun-expressions. The call above is just syntactic sugar for the following.

```
lists:filter(fun (X) -> erlang:is_number(X) end, SomeList)
```

The semantic equivalence of an implicit fun-expression and the corresponding explicit one has an immediate consequence. When an implicit fun-expression is passed as an argument to a higher-order function defined in another module (say M_1), the function named in the implicit fun-expression should be visible in the module (say M_2) that contains the call, and it need not be visible for M_1; i.e. it can be a not exported function of M_2.

Implicit fun-expressions can be translated into explicit fun-expressions by the refactoring tool, and the intended transformations can be applied on the ordinary function call expression occurring in the body of the explicit fun-expression. RefactorErl handles implicit fun-expressions in this way.

Explicit Fun-Expressions. Refactoring unnamed functions requires a different approach. For obvious reasons, the transformation *Rename Function* makes no sense for unnamed functions, but *Reorder Function Arguments* and *Tuple Function Arguments* can be applied on them. It is important to note, however, that RefactorErl does not support these transformations on unnamed functions yet. Therefore the following discussion is rather hypothetical. Consider again the call when an explicit fun-expression is passed to `lists:filter/2`.

```
lists:filter(fun ({_, P}) -> P /= Pid end, Users)
```

Reorder Function Arguments is not applicable on this unary function, but *Tuple Function Arguments* certainly is. Although this is a strange application of the refactoring, because it produces a tuple of a single element, it can be meaningful in some situations. The refactored unnamed function would look like this.

```
fun ({{_, P}}) -> P /= Pid end
```

Observe the double curly braces in the argument pattern: they tell us that the argument is a tuple having a single element, and this element is a tuple having two elements, the second of which is bound to variable `P`.

There are basically two ways to ensure that the semantics of the above call to `lists:filter/2` is not changed by this transformation. Either the definition of `filter` has to be changed, or the actual argument to `filter` should be modified. The first solution might yield the following definition for `filter`.

```
filter(Predicate,List) when is_function(Predicate,1)
    -> [ E || E <- List, Predicate({E}) ].
```

The problem with this solution is that changing the definition of a standard library function such as `lists:filter/2` is probably undesirable. But even if this higher-order function were a user defined function, changing its definition would influence many other calls to this function, calls when functions different from the one being transformed are passed as actual arguments. It is unlikely that the programmer, who applied the refactoring, would like the transformation change all those other calls as well.

Another problem is caused by dynamic typing. In Erlang it is possible to define functions which are polymorphic in a strange way. If you leave out the guard from the definition of `lists:filter/2`, the expression `lists:filter(3,[])` evaluates to `[]`; the first argument need not be a function, if the second argument is an empty list. Functions might have formal arguments that can accept actual arguments of different types. Variables can also be polymorphic. Therefore it is usually not possible to tell whether a variable or an actual argument is a function. Even if the refactoring tool locates all the calls to the strangely polymorphic `filter/2` function, it is not able to tell when it is called with a function, so it is unable to tell which actual arguments to transform. Later on a workaround will be shown – it is possible to decide the type of an expression at run-time, hence the insertion of run-time checks into the code might help discover higher-order applications of `filter/2`.

There is one more issue not covered yet. The *Tuple Function Arguments* transformation should take care of the second argument of `is_function` in the guard. In this very case it need not change, but in general tupling function arguments may change the arity of a function. Moreover, in the above example it is easy to realize that `is_function` is called on the transformed function. However, figuring out that `is_function` is applied to an expression which yields the unnamed function that is being refactored is quite involved in the general case. It might even occur that the value of the expression on which `is_function` is applied depends conditionally on the transformed function.

The problem is even more universal. Consider the case when an explicit fun-expression is assigned to a variable at one of the possibly many bindings of that variable. Other bindings might assign other functions to that variable, and – since variables might be polymorphic – non-function values can also be assigned to it. The variable might be used several times, sometimes as a function (e.g. applied on actual arguments, passed to BIFs such as `is_function` etc.), but it is usually not known statically under what circumstances the variable is bound to the transformed unnamed function. The refactoring tool should either refuse the execution of the refactoring when it is not possible to determine the effect of the transformation statically, or it should insert run-time tests into the code in such situations. The first solution is too restrictive, the latter derogates both readability and efficiency of the code.

Due to the problems described so far, compensation at the applications of a transformed unnamed function is hard to implement. The second compensation technique seems more feasible: compensation right where the transformed explicit fun-expression is introduced. In the presented example this means that the original definition of `lists:filter/2` is kept, and the actual argument to `lists:filter/2` is modified; an adapter function is inserted between `filter` and the refactored unnamed function definition.

```
lists:filter(fun (X) -> fun ({{_, P}}) -> P /= Pid end ({X}) end, Users)
```

Admittedly, in most cases it does not seem very practical to transform an unnamed function definition, and simultaneously wrap it in another function expression, generated by the refactoring tool. However, it might be sensible in situations when the code is being prepared for further transformations.

One might conclude that in theory it is possible to apply transformations on unnamed functions and still preserve semantics using the compensation techniques shown above. It seems, however, that in practice it is not really worth the trouble. This is why RefactorErl currently refuses refactoring the arguments of explicit fun-expressions, and leaves this issue for future development.

Reflective Programming. In addition to the complications introduced by unnamed functions, named functions also raise many challenges. It is possible in Erlang to compute the signature and the actual argument list of a named function at run-time, look up the function dynamically and then call it with the computed arguments. This reflective programming technique is supported both by standard library functions and by language syntax. The most general and most flexible method is the use of the apply/3 BIF. The first two arguments of this function should evaluate to atoms: the name of a module and the name of a function in that module. The third argument should evaluate to a list: the elements of this list will serve as the actual arguments of the call. The following function definition is from the module gen_server (implementing a generic server) used by [10].

```
doStart(Name, Module, Args, ParentPid) ->
    {ok, State} = apply(Module,init,[Args]),
    register(Name,self()),
    ParentPid!started,
    loop(State, Module).
```

This definition is very similar to gen_server:init_it/6 from the Erlang/OTP standard library version R9B-1, but doStart/4 is simpler and shorter, and so it is more apt for illustrating the issue. The use of apply/3 here is an instance of reflective programming, because the name of the module containing the function to be called by apply is computed dynamically. Note that the name of the function and the argument list is provided statically: the name of the function is init and the argument list is a list of a single element, Args. A call to doStart/4 should provide the name of a module that contains a function init/1. Of course it is possible to make the name of the function and the argument list dynamic as well, like in the following call.

```
apply(list_to_atom(factor(97)), list_to_atom(factor(1071509)), factor(6))
```

If factor/1 is a prime-factorization function returning the list of prime factors of a given number, then the above call will apply a:egg/2 on the actual arguments 2 and 3. (The list_to_atom function is a BIF that takes a string – represented as a list of ISO Latin-1 codes in Erlang – and creates an atom from it.) A refactoring tool has no chance to find out by static analysis that a:egg/2 is executed here. The only way to preserve program behaviour when refactoring a:egg/2, for example when swapping its arguments is to insert dynamic checks. The above call to apply/3 could be replaced with the following expression, assuming that M, F, A, A1 and A2 are fresh variables.

```
begin
M=list_to_atom(factor(97)), F=list_to_atom(factor(1071509)), A=factor(6),
case {M,F,A} is
   {a,egg,[A1,A2]} -> a:egg(A2,A1);
   _               -> apply(M,F,A)
end
end
```

This compensation technique is possible to optimize: often some of the three factors (module name, function name and arity) that determine a function signature are statically known. In the fragment from the gen_server, the function name and arity was fixed (i.e. init/1), only the module name was unknown. When a function init/1 of some module m is refactored (say renamed to initialize), the `apply(Module,init,[Args])` call needs compensation (as shown below), otherwise no compensation is necessary for this specific call expression.

```
case Module is
   m -> m:initialize([Args]);
   _ -> apply(Module,init,[Args])
end
```

The amount of compensation code can be further decreased if the refactoring tool adds compensation code to existing compensation code in a clever way. Assume that after renaming `m:init/1`, the programmer decides to rename `n:init/1` to `initialize` as well. Instead of adding compensation to the second branch of the above `case` expression, the tool can contract the two pieces of compensation code into one.

```
case Module is
   m -> m:initialize([Args]);
   n -> n:initialize([Args]);
   _ -> apply(Module,init,[Args])
end
```

This optimization could be supported by implementing the compensation code with macros. The refactoring tool could generate macro definitions and macro calls, and later let them evolve. Macros generated by RefactorErl should have tool-specific (but not obscure) names, so that the relevant macros could be easily identified – without sacrificing the readability of the code.

The example with the `gen_server` reintroduces the question mentioned already in respect to `lists:filter/2`: what to do with standard libraries? Shall the tool refactor them? In a pragmatic approach, the answer is definitely *no*. The tool should be aware of which modules in a system are considered "read-only", and should contain built-in knowledge about these modules if necessary. For example, if the `gen_server` is considered as part of the (unmodifiable) standard library, its `doStart/4` function will never be extended with compensation code. Instead, compensation code should go into the call expression, whenever this function is called from application code (i.e. not from within the standard library). By looking at the definition of `doStart/4`, it is easy to discover that its second argument should be an atom, the name of a module, and `doStart/4` calls a function `init/1` from this module with an actual argument that comes from the third argument of `doStart/4`. Therefore, if *Tuple Function Arguments* is applied on the argument of a function `m:init/1`, each call `doStart`$(\alpha, \beta, \gamma, \delta)$ should be turned into the following expression (assuming that α, β, γ and δ are some expressions, the value of β is not known statically, and `V` is a fresh variable).

```
case β is
   m -> doStart(α,m,{γ},δ);
   V -> doStart(α,V,γ,δ)
end
```

Thanks to `V`, the above expression does not evaluate β more than once, which is important if β might have side effects.

Note that the second (`Module`) argument of `doStart/4` is used not only in the `apply/3` call, but also in the call to `loop/2`. For this reason the compensation code should not change the value of the second actual argument in calls to `doStart/4`. Assume again that the application code contains the `doStart`$(\alpha, \beta, \gamma, \delta)$ call. Given a refactoring for moving a function definition from one module to another, the refactoring tool should refuse to apply it on `m:init/1`, because in this case the task of the compensation would be to change `m` to the new module name in the call to `doStart/4`, and this would invalidate the call `loop(State,Module)` inside `doStart/4`.

To achieve full standard library coverage, the refactoring tool should completely understand the library interface, the role of the different functions and their formal arguments. As the examples above suggest, collecting the necessary information may require extensive, lengthy analysis as well as human help – a good idea is to build this knowledge into the refactoring tool in advance. At the least, support for the built-in functions, like `apply/3`, should be provided.

Not only `apply/3`, but many further BIFs result in calling a function reflectively. Such BIFs are `spawn/3`, its many variants and `hibernate/3`. (Note that `spawn/1`, appearing e.g. in the `srv` module in Sect. 2, expects a fun-expression as argument, so it is not reflective.) The BIF `apply/2` is also quite interesting. Officially it is a higher-order function, which, similarly to `spawn/1`, takes a fun-expression as its first argument (the second argument serves as the actual argument list for the function in the first argument). However, `apply/2` can also be called by providing a tuple as first argument: the tuple should contain a module name and the name of a function in that module. This second way of calling `apply/2` is again an instance of reflective programming. Although this feature is deprecated, old code can contain such calls. As an example, assume that the two arguments of `a:egg/2` are swapped with the *Reorder Function Arguments* transformation. Assume, moreover, that the application code (code outside of the standard library) contains a call `apply`(α, β), where neither α nor the length of the list yielded by β is known statically. The call to `apply/2` should be replaced with the following expression, where `A` and `B` are fresh variables.

```
begin
    A = α, B = β,
    case {A, length(B)} of
        {{a,egg},2} -> a:egg((fun([X,Y])->[Y,X] end)(B));
        _           -> apply(A,B)
    end
end
```

Apart from the afore-mentioned BIFs (and any standard library functions that use these BIFs) some *language constructs* of Erlang are also suitable for calling functions reflectively. For example, the call `apply(Module,init,[Arg])` can also be written as `Module:init(Arg)`. In general, if α evaluates to the name of a module (an atom), β evaluates to the name of a function (again an atom), the named module contains an n-ary function with the given name, and $\gamma_1 \ldots \gamma_n$ are some expressions, then $\alpha : \beta(\gamma_1, \ldots, \gamma_n)$ is a valid reflective function call. This technique is preferred to using `apply/3` whenever the arity of the function to be called is known statically (cf. `gen_server:init_it/6` in Erlang/OTP versions R9B-1 and R12B-0). Obviously, the compensation techniques applicable to uses of `apply/3` are suitable for calls of the form $\alpha : \beta(\gamma_1, \ldots, \gamma_n)$ as well.

Similarly to the behaviour of the `apply/2` built-in function, there is a language construct in Erlang that permits calling functions in semantically different ways: an expression in the form $\beta(\gamma_1, \ldots, \gamma_n)$, where β, γ_1, ... γ_n are some expressions. The call is legal if β evaluates to a function (like `Predicate` in the definition of `lists:filter/2` on page 266), and also when it evaluates to a tuple of a module name and a function name. The second (deprecated but still occurrent) possibility is again an instance of reflective function calls. For example, the call `apply(Module,init,[Args])` in `doStart/4` might be written as `{Module,init}(Args)`. The difference between the usage of `apply/2` and expressions of the form $\beta(\gamma_1, \ldots, \gamma_n)$ is that the latter is preferable when the number of arguments is known statically. Again, compensations for the latter sort of calls are similar to those used for the former.

As a final remark on reflection, note that in Erlang, apart from looking up and calling functions reflectively, it is also possible to construct and execute code at run-time. This possibility is supported by the standard libraries `erl_scan`, `erl_parse` and `erl_eval`. The following `eval` function, taken from [24], can evaluate Erlang source code provided in a string. The argument `S` should contain an Erlang expression sequence. The evaluation takes place with respect to a variable environment passed as the second argument to the function.

```
eval(S,Environ) ->
    {ok,Scanned,_} = erl_scan:string(S),
    {ok,Parsed} = erl_parse:parse_exprs(Scanned),
    erl_eval:exprs(Parsed,Environ).
```

This use of `erl_eval` (The Erlang Meta Interpreter) is clearly out of scope of static program analysis, and can hardly be supported by a refactor tool.

To conclude the topic of function-related refactorings, the following statements can be made. RefactorErl handles both statically bound calls to named functions and implicit fun-expressions properly. Function-oriented transformations on explicit fun-expressions seem impractical and are at present refused by the tool. Finally, the semantics of programs using reflective techniques might not be preserved by the implementation of the transformations in the current version of RefactorErl, so the tool emits a warning if such a case is detected. Since many programs utilize some sorts of reflection mechanism, completely refusing the refactorization of such programs would be worthless in practice. As a trade-off between safety and effectiveness, the tool is willing to apply transformations even if it cannot take the responsibility for preserving the semantics of function calls that result from reflective programming. In the future more support for reflection will be added to RefactorErl based on the compensation techniques discussed in this section.

5.2 Variable Definitions and Variable Applications

A variable in Erlang binds a value to a name in an immutable way: once the variable is bound, it will evaluate to that value during its whole lifetime. However, due to branching constructs, the same variable might be bound at more than one place (it is said that the variable might have more than one *binding occurrence*), which means that there may be more than one definitions that determine the value bound to the variable. At runtime, when the variable is created, one of these definitions will supply the value for the variable. Recall how a variable `Pid` of `srv:loop/3` in Fig. 1 was identified and renamed in Sect. 3.4. This variable has two binding occurrences, in the second and the in fourth branches of a `receive` statement, respectively.

Many refactorings are concerned with variables in some way. Before applying such a refactoring, the refactor tool should understand the "binding structure" of the program being transformed: figure out which variable occurrences refer to the same variable, which expressions define (bind) the variable and which expressions apply (evaluate) the variable. The side conditions and the transformation rules of refactorings such as *Rename Variable*, *Eliminate Variable*, *Merge Expression Duplicates* and *Extract Function* depend on the answers to the above questions. As an example, consider the transformation *Eliminate Variable*. One of its side conditions is that the variable to be eliminated, say `X`, has only one binding occurrence, viz. in a match expression of the form "`X =` ϵ". The transformation removes this definition, and replaces all the other (so-called *applied*) occurrences of `X` with ϵ (cf. Sect. 3.3).

Incidentally, the above side condition for *Eliminate Variable* could be and will be relaxed in the future to allow the elimination of a variable with more than one definitions, on condition for each applied occurrence of the variable there is a unique definition which provides the value for that applied occurrence. In Fig. 11 one can see a function which inserts an element into an unbalanced binary search-tree. A tree is represented with a tuple: an empty tuple for an empty tree, and a triple of a left subtree, a root element and a right subtree for a non-empty one. The `New` variable has two distinct definitions in the first two branches of the `if`-statement. It would be possible to eliminate this variable by replacing the

```
insert( Tree={Left,Root,Right}, Value ) ->
    if
       Root < Value -> New = insert(Right,Value),
                       {Left,Root,New};
       Root > Value -> New = insert(Left,Value),
                       {New,Root,Right};
       true         -> Tree
    end;
insert( {}, Value ) -> {{},Value,{}}.
```

Fig. 11. Insertion into an unbalanced binary search-tree

two applied occurrences with the corresponding definition, although RefactorErl currently refuses to do so.

In this section the rules that a refactor tool should keep in mind when identifying variables and distinguishing between binding and applied variable occurrences are investigated. Compared to the rules for function calls the rules for variables seem quite complicated, but still, variables cause less trouble for a refactor tool. In contrast to the problems raised by the higher-order nature of Erlang and the reflection facilities, the binding structure is possible to completely reveal by static analysis.

Patterns. Variables in Erlang are introduced by variable bindings: when a pattern is matched against an expression, the unbound variables of the pattern become bound. This happens, for instance, when a function is called and the value of actual arguments are bound to the formal arguments. Other constructs containing patterns are match expressions (like `NextNr = Nr + 1`), `case`-, `receive`- and `try`-expressions, and generators of list comprehensions. Patterns can be simple (like `NextNr` above) or more complex: for example, nested patterns can be used to bind the components of a compound expression to different variables, like the fourth argument in the second clause of `send/4` in Fig. 1.

Patterns are similar to "terms" (terms are the values that Erlang programs operate on): patterns can be formed from numbers, atoms, strings, lists of patterns and tuples of patterns. However, patterns may also contain variables (terms may not), either bound or unbound. The following function decides whether a given list contains a given value. The only binding occurrence of the variable `Value` is in the formal argument list. The pattern of the first branch of the `case`-expression also refers to this variable, but since `Value` is already bound, this is an applied occurrence of the variable. The third occurrence of `Value` is not in a pattern, so it is clearly an applied occurrence.

```
contains( List, Value ) ->
    case List of
        [Value|_] -> true;
        [_|Tail]  -> contains(Tail,Value);
        _         -> false
    end.
```

The refactor tool should be capable to find out whether a variable occurrence in a pattern is a binding or an application of a variable. The decision cannot be made by looking at merely the pattern expression, because the syntax is the same for binding and applied occurrences. The answer depends on the context of the pattern expression, i.e. the surrounding program text.

Another difference between patterns and terms is that patterns may contain the match operator as well. If π_1 and π_2 are patterns, then $\pi_1 = \pi_2$ is also a pattern. When matching this latter pattern against an expression ϵ, both π_1 and π_2 are matched against ϵ. Note that this construct is a generalization of as-patterns of Haskell [12]. The first formal argument of the first clause of `insert/2` in Fig. 11 is basically an as-pattern that binds four variables: `Tree` is bound to a tuple, while `Left`, `Root` and `Right` are bound to the components of the tuple.

There is still one more addition to patterns compared to terms: patterns may contain arithmetic expressions that can be evaluated in compilation time. The compiler replaces such expressions with their computed value. These expressions do not contain variables, and therefore they are irrelevant to the current discussion.

Pattern Matching. The matching of a pattern against an expression can succeed (and in such a case the unbound variables of the pattern become bound) or fail. The latter happens when the structure of the pattern differs from that of the expression, or when there exists a subpattern in the pattern which is neither an unbound variable, nor equal to the corresponding subexpression in the expression.

It is also possible that the same unbound variable occurs more than once in a pattern: in that case the same value should be bound to it at every occurrence, otherwise the matching fails. For example, if `X` is an unbound variable, the pattern `{X,X}` matches the expression `{1,1}`, but it does not match `{1,2}`. One could interpret this mechanism in an alternative way, as follows. When a pattern is matched against an expression, the subpatterns of the pattern are matched against the subexpressions of the expression in some order. Whenever a subpattern which is an unbound variable is matched, the variable becomes bound; next time when the variable, being another subpattern, is matched, it is already bound, so its bound value should merely be compared to the value of the corresponding subexpression. The problem with this alternative interpretation is that it is not possible to tell which occurrence of the variable in the pattern is the binding occurrence, because the order in which the subpatterns of a pattern are processed is not specified (it might be compiler-dependent). Therefore the first interpretation, namely that all occurrences of an unbound variable in a pattern are considered binding occurrences, is more rational. RefactorErl handles such a variable in a special way: although the variable has more than one binding occurrences inside the pattern, the pattern as a whole is considered a single binding place for the variable. This can be illustrated on the function definition on the left-hand side of Fig. 12. The variable `Z` is bound by the pattern of the first branch of the `case`-expression, and `Z` has two binding occurrences in this pattern. The only applied occurrence of `Z` is the one in the body of the same `case`-branch. Now it is possible to apply the *Merge Expression Duplicates* transformation on the applied occurrence of `Z` (that is on the expression made up of a single variable application). This expression appears only once in the code, so it is a rather

```
gcd(R) ->
   case R of
      {Z,Z}            -> Z;
      {X,Y} when X > Y -> gcd( {X-Y,Y} );
      {X,Y}            -> gcd( {X,Y-X} )
   end.
```

```
gcd(Z,Z) -> Z;
gcd(X,Y) ->
   if
      X > Y -> gcd(X-Y,Y);
      true  -> gcd(X,Y-X)
   end.
```

Fig. 12. Computing the greatest common divisor of two positive integers

degenerated, but still legal case for *Merge Expression Duplicates*; a new variable has to be introduced, which is bound to the selected expression (observe that the selected expression itself is again degenerated, since it is only a variable application), and the single occurrence of the selected expression is replaced by this new variable. Assuming that the name of the new variable is `N`, the first branch of the `case`-expression becomes "`{Z,Z} -> N = Z, N`". RefactorErl further optimizes this result, so the transformation eventually yields "`{Z,Z} -> N = Z`". The point here is that the side condition of *Merge Expression Duplicates* – somewhat similarly to that of *Eliminate Variable* – requires that each bound variable of the selected expression should have a unique binding (this requirement can be, and will be completely removed in the future). Although `Z` has two binding occurrences, RefactorErl considers the containing pattern as a unique binding place for the variable, and hence the side condition of *Merge Expression Duplicates* is satisfied.

Pattern matching in general is not just about matching a pattern against an expression. For example, in a `case`-expression, like the one in `gcd/1` in Fig. 12, the received expression is matched against a sequence of alternative patterns. The alternative patterns are tried in textual order, and if the matching of one of the patterns succeeds (and, if guards are present, the guards evaluate to `true`), the whole pattern matching also succeeds, and the corresponding branch of the `case`-construct is evaluated. If none of the alternative (possibly guarded) patterns matches the expression, the whole pattern matching fails, and a run-time error is raised. The pattern matching mechanism is further complicated by function definitions. The formal argument list of a function with arity greater than one contains more than one patterns: a clause of such a function matches a call only if each of the patterns matches the corresponding actual argument (and, if guards are present in the clause head, they evaluate to `true`). The simultaneous matching of the patterns that make up the formal argument list can be modelled by matching a tuple of the formal arguments (a single pattern) to the tuple of the actual arguments (a single expression). As a consequence, an (unbound) variable might occur more than once in a formal argument list, like in the function definition shown on the right-hand side of Fig. 12. In such a case it is rational to treat the whole formal argument list as a single binding place for the variable (a binding place with more than one binding occurrences). Similarly to `gcd/1`, RefactorErl can perform *Merge Expression Duplicates* on the applied occurrence of `Z` in the first clause of `gcd/2`.

Contexts. The variable binding rules of the Erlang specification [2] are given in terms of *input and output contexts* for every language construct (a reformulation of these rules, used by RefactorErl, is given in [17]). A context contains the value for the bound variables. The meaning of an expression depends on its input context. The evaluation of an expression yields a result (a term, which is the value of the expression, or a run-time error), and produces an output context (furthermore, expressions might have side effects – this issue will be discussed in Sect. 5.3).

The lifetime of a variable begins when the variable is created in a pattern by a binding. In most cases, the lifetime of a variable ends at the end of its scope[1]. The language constructs that introduce scopes are function clauses, list comprehensions, bit string comprehensions and generator expressions of (list and bit string) comprehensions. These constructs can be nested, and so can scopes be as well. Every variable is local to such a construct, i.e. no variable can be defined at the module level. Since function declarations are always at the module level, and comprehensions are always nested (directly or indirectly) in function clauses, the outermost scopes are introduced by the clauses of function declarations. The good news that immediately follows from this is that the unit of code which the analysis of the binding structure has to consider is a single clause of a named function.

Programmers often use the same name for formal arguments in the different clauses of the same function. For example, the second argument in both clauses of `insert/2` in Fig. 11 is called `Value`. The usage of the same variable name indicates some logical relation between those formal arguments, but in fact the variables introduced in different function clauses are completely independent from each other – in the case of `insert/2` and the alike some formal arguments just happen to have the same name. If `Value` is renamed in the first clause of `insert/2` (by using the *Rename Variable* transformation of RefactorErl), the second clause of this function will remain unmodified. The refactoring preserves the semantics of the program – but not the style. Of course, one could introduce another, different refactoring, called maybe *Rename All Related Variables*, that would rename both `Value` variables at the same time. Such a refactoring could be even more aggressive in finding "related" variables; for example, it might discover that the name `Max` in the chat server of Fig. 1 identifies related variables occurring in `start/1`, `init/1` and `loop/3`. Such (semantics and) style preserving transformations require appropriate heuristics. Does the name `Pid` identify two related or unrelated variables inside `srv:loop/3`? Heuristics-based transformations can be extremely useful and powerful when refactoring programs that obey to certain coding conventions. Records, for example, were introduced in Erlang as syntactic sugar to facilitate the usage of tuples and to improve the readability and maintainability of programs. Records are typically used for communicating chunks of related data. The consistent use of records instead of the underlying tuples assumes obeying to certain coding conventions, and this fact could be exploited by record-related refactorings [16].

As mentioned earlier, due to the lack of syntactic differences, it is not possible to decide whether a variable occurrence in a pattern is a binding one or an applied one by merely looking at the pattern expression: it is necessary to investigate the input context of the pattern expression as well. If the input context contains

[1] The scope of a variable created directly in a filter of a (list or bit string) comprehension extends through the rest of the qualifier list and the head of the comprehension. However, the lifetime of such a variable ends after the generation of each element of the list or bit string produced by the comprehension, and not after the whole comprehension is evaluated.

a binding for a variable, then the occurrences of the variable in the pattern are applied occurrences, otherwise they are all considered binding occurrences. The concept of contexts is suitable to understand the binding structure of a legal Erlang function declaration, but it is not sufficient for deciding whether a function declaration is legal. Consider the following illegal function declaration.

```
illegal_function(X) when X /= 0 ->
   if
      X < 0 -> Y = -X, Z = gcd(Y,1970);
      true  -> Z = gcd(X,2007)
   end,
   Y = Z.    % compilation error
```

The input context of the match-expression `Y = Z` contains a binding for `X` and `Z`, but what about `Y`? It is not allowed to use `Y` in the match-expression, because whether it is bound or unbound depends on run-time information, namely on the sign of the actual argument (`Y` is said to be "unsafe" in the `if`-expression). If `Y` were used in a non-pattern expression, for example on the right-hand side of the match-expression, the function definition would be obviously incorrect, because a variable should not be applied if there is a possibility that it is unbound. The problem with `illegal_function` is less obvious. This function definition could be meaningful: if `X` is positive, the match-expression could bind the value of `Z` to `Y`, and if `X` is negative, the match-expression could try to match the value of `Y` to that of `Z` (and fail if `X` is not equal to -1970). However, the legality rules of Erlang require that the binding status of a pattern variable is known statically (which is indeed useful for generating efficient code), so `illegal_function` is refused by the compiler.

It is true that refactoring transformations might assume that the program to transform is legal – one possibility is to invoke the compiler front-end before starting refactoring to check the legality of the code. However, refactorings that move around expressions (especially *Eliminate Variable* and *Merge Expression Duplicates*) have to ensure that the structurally modified code (1) respects the legality rules described above and (2) produces the same result as the original code. For these reasons the refactoring tool applies static analyses that, similarly to those performed by the compiler, go far beyond input and output contexts. Special care should be taken, for example, when expressions containing variable bindings are moved. Consider the following, not particularly smart definition of `make_rational`, which produces a record representing a rational number as a fraction of an integer and a positive integer. Note that, for the sake of the example, two alternative implementations are provided for the third branch of the `case`-expression, the latter being commented out.

```
gcd(A,0) -> A;
gcd(A,B) -> gcd(B, A rem B).
make_rational(X,Y) ->
    GCD = gcd( abs(X), (AY = abs(Y)) ),
    case Y of
        0  -> throw(division_by_zero);
        AY -> #rational{numerator = X/GCD, denominator = Y/GCD};
        _  -> #rational{numerator = -X/GCD, denominator = AY/GCD}
            % #rational{numerator = -X/GCD, denominator = -Y/GCD}
    end.
```

The definition of the variable GCD is an expression that binds the variable AY as a side effect. The side conditions should prevent the application of *Eliminate Variable* on GCD, because the transformation would yield an illegal program: the third branch of the case-expression would become the following.

```
_ -> #rational{ numerator = -X/gcd(abs(X), (AY = abs(Y))),
                denominator = AY/gcd(abs(X), (AY = abs(Y))) }
```

This contains an unbound applied occurrence of AY, which is illegal. Similarly, given the second implementation of the third branch of the case-expression, the side conditions should again prevent the elimination of GCD, because the refactoring would yield a (legal) binding structure that is completely different from that of the original definition. The resulting code would bind AY in the pattern of the second branch of the case-expression to the value of Y.

```
AY -> #rational{ numerator = X/gcd(abs(X), (AY = abs(Y))),
                 denominator = Y/gcd(abs(X), (AY = abs(Y))) }
```

In the body of this branch AY = abs(Y) would compare the value of AY to abs(Y), causing a "badmatch" run-time error if Y is negative. Furthermore, the third branch of the case-expression would become legal but unreachable.

The static semantic rules of Erlang enable the compiler to guarantee that every occurrence of a variable can be statically classified as either a binding or an applied occurrence; furthermore, a variable is never applied if it is unbound, and never bound if it is already bound. The reason for talking about the role of the compiler here will be apparent from the following example. Assume that X is an unbound variable, and α and β are expressions. The (not very practical) expression (X=α)+(X=β) contains both a binding and an applied occurrence of X. Although the rules of the language do not specify whether the former or the latter occurrence binds the variable (the order of evaluating the arguments of + is specified as compiler-dependent), each compiler knows statically which is the binding occurrence. The problem here is that the refactor tool should be based on the rules of the language, and not on the peculiarities of a compiler, so the right approach is to consider both occurrences of X a "possibly binding occurrence". But is this really a problem? What is happening at run-time with respect to X? If α and β are equal, then X becomes bound to their value, and the expression evaluates to $\alpha+\beta$ – otherwise after evaluating α and β in some order, a "badmatch" run-time error occurs, and the control leaves the scope of X. Indeed, it is often unimportant which is the binding occurrence. Assume, for instance, that X is to be eliminated. The current implementation of *Eliminate Variable* will refuse the transformation, because the side condition about the unique definition of X is violated. However, if other side conditions allowed (e.g. neither α nor β had any side effects), in an improved implementation of this refactoring it would be possible to eliminate `X`: the expression `(X=`α`)+(X=`β`)` could become α`+(`α`=`β`)` (or, equivalently, `(`α`=`β`)+`β), and every applied occurrence of `X` could be replaced with either α or β – it does not matter which one, because if they are not equal, then the applied occurrences of `X` (or the expression replacing them) will never be evaluated.

Shadowing also raises an interesting issue. It is possible to introduce a new variable within the scope of another variable with the same name: variables occurring in the formal argument list of a function definition (either declared or unnamed) and in the generator patterns of (list or bit string) comprehensions are always new and unbound, and therefore they might shadow variables of the enclosing scope (see e.g. the discussion in Sect. 3.4 about the `Pid` formal argument of the unnamed function defined in the third branch of the `receive`-expression in `srv:loop/3`, Fig. 1). Now consider the following strange identity function.

```
identity(X) -> (fun(Y) -> Y end)(X).
```

It is legal to rename `Y` to `X`: the two variables remain distinct, and the `X` in the unnamed function shadows the `X` in `identity`. Some think that shadowing reduces readability. The Erlang compiler gives a warning message when it encounters such a situation. A refactoring tool may respect this opinion, and also give a warning when shadowing is introduced by a transformation, for instance, when `Y` is renamed to `X` in the above example. The current implementation of RefactorErl, however, does not report a warning in such cases.

To conclude the discussion on variables, the following observations are emphasized. Each clause of a declared function is independent with respect to variables. Although the rules for variable bindings are rather complex, the binding structure is possible to reveal by static analysis, therefore refactorings can be applied completely safely. The current implementation of some refactorings in RefactorErl can be improved by weakening their side conditions – this is ongoing work.

5.3 Side Effects

Erlang is impure, some expressions (typically those related to concurrency, distribution, communication, fault-tolerance and IO) have side effects. Refactorings that change the evaluation order of expressions or the number of times a certain expression is evaluated should be careful not to interfere with side effects.

Consider, for example, the program fragment in Fig. 13, which is a simple implementation of the parallel Divide and Conquer algorithm. The side conditions should disallow the performance of *Eliminate Variable* on `Pid` in `build_structure/3`, because `spawn_link/1` (a variant to `spawn/1`, used for spawning a new process) is not a pure function: each time it is called, a different result is returned. The transformation would move the call of `spawn_link/1` into the last two lines of `build_structure/3`, causing the creation of two processes with different process identifiers. This is obviously differs from what the current code is doing. Similarly, the side conditions should prevent the execution

```
% Fun should be a commutative and associative binary operation and
% be defined on the elements of the non-empty Values list
divide_and_conquer( Fun, Values=[_|_] ) when is_function(Fun) ->
    build_structure(self(), Fun, Values),
    recv().

build_structure( Parent, _Fun, [Value] ) -> Parent ! Value;
build_structure( Parent, Fun, Values ) ->
    Pid = spawn_link(fun() -> compute(Parent, Fun) end),
    {Left,Right} = lists:split( length(Values) div 2, Values ),
    build_structure( Pid, Fun, Left ),
    build_structure( Pid, Fun, Right ).

recv() -> receive Val -> Val end.
compute( Parent, Fun ) ->
    Parent ! Fun(recv(),recv()).
```

Fig. 13. A parallel implementation of the Divide and Conquer algorithm

of *Merge Expression Duplicates* on the calls to `recv/0` in `compute/2`, since the `receive`-expression makes `recv/0` a function with side effect. The transformation would turn `compute/2` into the following.

```
compute( Parent, Fun ) -> R = recv(), Parent ! Fun(R,R).
```

This definition would only receive one message and use it twice instead of receiving and using two messages.

These two examples illustrate that the refactoring tool should know about language constructs that have side effect (i.e. message send and receive expressions), about functions that have side effect (BIFs like `spawn`, `spawn_link` etc.), and also transitively about all those functions that use constructs and functions with side effect. Given the list of BIFs and library functions (functions with an implementation unknown to the refactoring tool) that have side effect it is fairly simple to determine the set of functions with possible side effect in the program code being analyzed. Unfortunately, in the current version of RefactorErl neither the BIFs and library functions with side effects are collected, nor the analysis for determining the transitive closure of functions with side effect is implemented.

As mentioned earlier, it is rational to assume that the program code on which the refactoring tool operates is legal. Legality, however, need not merely mean that the code is accepted by the compiler – violations of the rules of the dynamic semantics of the language are typically not detected statically. It is reasonable to assume that refactored programs are terminating and are free of dynamic errors. Without this assumption one should consider every function and operator impure: they all might fail on some input (partiality, lack of memory etc.) and they may define infinite computation (halting problem). Whenever the evaluation order of expressions in a program is changed by a transformation, the behaviour of the program might change. In the following example a block-expression is presented.

```
begin e(e1), e(e2), ok end
```

The value of a block-expression is determined by the last expression in the block, here it is the atom `ok`. Is it allowed to change the order of the first two expressions of the block? Since the order of the two expressions has no effect on the value of the atom `ok`, the answer is yes. However, this transformation might change the behaviour of the code, in particular if `e/1` raises a run-time error, and the block-expression is embedded in a `catch`-expression.

```
catch begin throw(e1), throw(e2), ok end
```

It would not be very practical to prohibit refactoring transformations just because they modify the evaluation order of expressions. Therefore in RefactorErl run-time errors and non-termination are considered as dynamic semantic errors and not as side effects.

6 Related Work

Opdyke's thesis [20] is known to be the first publication on refactoring, although program transformations were used by programmers long before. Refactoring became popular after Fowler's refactoring bible [9] appeared, which addressed a wide range of transformations for object-oriented software, providing examples in Java. Most research activities in this field focus on object-oriented environments. An exhaustive survey on the existing techniques and formalisms is [18]. Refactoring is generally based on static analyses of program text (RefactorErl being no exception), but dynamic refactoring (verifying the conditions of transformations by inserting run-time checks and testing) is also feasible [22].

Tool support for refactoring was first provided by the refactoring browser for Smalltalk [21]. Many tools are available for Java, often embedded into a development environment (e.g. Eclipse [5], IntelliJ Idea etc.), and some for C# (ReSharper, C# Refactory) and C++ (SlickEdit, Ref++). These tools support various kinds of renamings, extracting/inlining code, and manipulating the class hierarchy. There is a good summary of the available tools and a catalog of well-known refactorings at [8].

Refactoring in functional languages has received much less attention. Haskell was the first functional language to gain tool support for refactoring, and so far the Haskell Refactorer [15] is the only functional refactorer software that is actually usable in practice. Refactoring functional programs using database representation first appeared in [4] for the Clean language, and a stand-alone prototype [23] is available from this research.

Refactoring Erlang programs is a joint research with the University of Kent, building on experiences with Haskell and Clean. While we are sharing ideas and experiences, Thompson and his research group are investigating a completely different implementation approach using traversals on annotated abstract syntax trees [14].

Type inference is a well-known technique for collecting semantic information on programs – this technique seems to be extremely useful for certain kinds of refactorings. Research on "success typings" in Erlang [13] might be a good starting point in this area.

7 Conclusions

This paper reports on RefactorErl, a tool for refactoring Erlang programs. The tool is integrated in the Emacs editor, and currently it has seven transformations implemented. A refactoring is made up of static analyses for checking the side conditions, and the transformation, which might include the generation of compensation code. In this paper examples were used for illustrating the conditions and the effect of the transformations, and for examining limitations and possible future improvements. Three major problems were identified and discussed: finding the calls to a given function, revealing the binding structure of a program, and avoiding interference with side effects. It turned out that static analyses cannot completely manage the first problem (due to reflective programming facilities), but they can manage the second one and the third one (the latter requiring the exclusion of non-termination and run-time errors). It is interesting to note that these three issues do not arise as problems when designing refactorings for languages like Haskell or Clean.

A refactoring tool will never be used in the industry if programmers do not trust in the refactoring transformations. The programmers expect that the transformations do not change the meaning of the program code being manipulated. However, a completely safe, conservative tool could hardly be used for refactoring real-world programs: it would consider too many transformations unsafe, and it would refuse to perform them. A trade-off between safety and serviceability is likely to be advantageous. The refactorings in RefactorErl are, and will be, designed to be semantically safe to an extent that should be acceptable for software engineers, although they should always be aware of the theoretical limitations of static analyses.

The approach presented here seems applicable in practice. However, the first public release of the tool is not matured enough for industrial use. First of all, it does not contain the analysis for distinguishing functions with side effects from those without, and for this reason it refuses to perform transformations in many situations when the transformations would be safe. This analysis is not a complicated one; its implementation is scheduled for the next release of RefactorErl. There are further cases when the computation of side conditions is suboptimal, and the tool is unnecessarily conservative. Macros and file inclusion (two constructs of the language that are used regularly in industrial code) are not supported in the first version of RefactorErl (they will be supported in the second one). Moreover, features like the undo facility and the preservation of code layout by the transformations (again very important for industrial use) will also be available only in the next release. Finally, performance and stability are not quite satisfactory in the first version, but will be greatly improved in the second one. The public release of this second version is expected in the near future – experiments with it on a large industrial code base (millions of LOC) are very promising. This release will contain even more refactorings.

References

1. Armstrong, J., Virding, R., Williams, M., Wikstrom, C.: Concurrent Programming in Erlang. Prentice-Hall, Englewood Cliffs (1996)
2. Barklund, J., Virding, R.: Erlang 4.7.3 Reference Manual (1999), `http://www.erlang.org/download/erl_spec47.ps.gz`
3. Distel: Distributed Emacs Lisp, `http://fresh.homeunix.net/~luke/distel/`
4. Diviánszky, P., Szabó-Nacsa, R., Horváth, Z.: Refactoring via database representation. In: The Sixth International Conference on Applied Informatics (ICAI 2004), Eger, Hungary, vol. 1, pp. 129–135 (2004)
5. Eclipse - an open development platform, `http://www.eclipse.org/`
6. GNU Emacs homepage, `http://www.gnu.org/software/emacs/`
7. Ericsson, A.B.: Erlang Reference Manual, Version 5.6 (2007), `http://www.erlang.org/download/`
8. Fowler, M.: Refactoring Home Page, `http://www.refactoring.com/`
9. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
10. Fredlund, L.A., Earle, C.B.: Model checking Erlang programs: The functional approach. In: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang, Portland, Oregon, USA, pp. 11–19. ACM Press, New York (2006)
11. Horváth, Z., et al.: Refactoring Erlang Programs, `http://plc.inf.elte.hu/erlang/`
12. Hudak, P., Peterson, J., Fasel, J.: A Gentle Introduction To Haskell, version 1.4, `http://www.cs.sfu.ca/CC/SW/Haskell/hugs/tutorial-1.4-html/index.html`
13. Jiménez, M., Lindahl, T., Sagonas, K.: A language for specifying type contracts in Erlang and its interaction with success typings. In: Proceedings of the 2007 ACM SIGPLAN Erlang Workshop, pp. 11–17. ACM Press, New York (2007)
14. Li, H., Thompson, S., Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T.: Refactoring Erlang programs. In: The Proceedings of 12th International Erlang/OTP User Conference, Stockholm, Sweden (November 2006)
15. Li, H., Thompson, S., Reinke, C.: The Haskell Refactorer, HaRe, and its API. Electronic Notes in Theoretical Computer Science 141(4), 29–34 (2005)
16. Lövei, L., Horváth, Z., Kozsik, T., Király, R.: Introducing records by refactoring. In: Proceedings of the 2007 ACM SIGPLAN Erlang Workshop, pp. 18–28. ACM Press, New York (2007)
17. Lövei, L., Horváth, Z., Kozsik, T., Király, R., Kitlei, R.: Static rules for variable scoping in Erlang. In: Proceedings of the 7th International Conference on Applied Informatics, vol. 2, pp. 137–145 (2008)
18. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Transactions on Software Engineering 30(2), 126–139 (2004)
19. MySQL AB homepage, `http://www.mysql.com/`
20. Opdyke, W.F.: Refactoring Object-oriented frameworks. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA (1992)

21. Roberts, D., Brant, J., Johnson, R.: A refactoring tool for Smalltalk. Theory and Practice of Object Systems (TAPOS) 3(4), 253–263 (1997)
22. Roberts, D.B.: Practical analysis for refactoring. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA (1999)
23. Szabó-Nacsa, R., Diviánszky, P., Horváth, Z.: Prototype environment for refactoring Clean programs. In: The Fourth Conference of PhD Students in Computer Science (CSCS 2004), Volume of extended abstracts, Szeged, Hungary, July 2004, p. 113 (2004), http://aszt.inf.elte.hu/~fun_ver/
24. Trapexit. String Eval, http://wiki.trapexit.org/index.php/String_Eval

From Interpretation to Compilation

Jan Martin Jansen[1], Pieter Koopman[2], Rinus Plasmeijer[2]

[1] Netherlands Defence Academy,
Faculty of Military Sciences,
Den Helder, The Netherlands
`jm.jansen.04@nlda.nl`

[2] Institute for Computing and Information Sciences (ICIS),
Radboud University Nijmegen, The Netherlands
`{pieter,rinus}@cs.ru.nl`

Abstract. In this paper we sketch some experiments with the construction of a simple compiler for a high level intermediate lazy functional language, with C++ as a target language. Because the compiler is intended for educational and experimental use, simplicity and clearness of construction are considered to be more important than efficiency. Starting point for the construction is a simple interpreter. In a first step this interpreter is turned into a simple compiler in a straightforward manner. The performance of a number of compiled benchmarks is analysed in a comparison with the interpreter and the Clean and GHC compilers. This analysis leads to some suggestions for optimisations. Of these optimisations tail recursion optimisation and optimisation of numerical functions and numerical (sub)expressions in functions are implemented. It turns out that in many cases these optimisations suffice to obtain a competitive performance.

1 Introduction

The construction of efficient compilers for lazy functional programming languages like Clean [8] and Haskell [1] is a complex task. Compilers like GHC and Clean are large complicated systems that are too complex for study in introductionary courses on the implementation of functional programming languages. Therefore there is a need for simple compilers for educational purposes. Our main goal is to give the reader some insight in what kind of optimisations are important for obtaining an efficient implementation of lazy functional languages.

In [2] we constructed a simple but efficient interpreter for the lazy functional language SAPL. SAPL can be used as an intermediate language for the interpretation of languages like Clean and Haskell. We already constructed a Clean to SAPL translator. Several versions of the SAPL interpreter exist. One of these versions is a Java applet implementation that can be loaded in Internet Browsers and which makes it possible to run Clean programs at the client side of internet applications ([6] and [7]).

In this paper we investigate how we can extend the SAPL interpreter to a SAPL compiler with a reasonable performance. We use C++ as target language.

The construction is made in two steps. In the first step we convert the interpreter into a straightforward but naive compiler. We then use a number of benchmarks to analyse the performance of the generated code in a comparison with the Clean and GHC compiler. It turns out that in some cases the performance is already quite good but that in other cases the performance is still very bad (more than 30 times slower). In an analysis of the characteristic of the poor performing benchmarks, it turns out that they often have some commonalities like the (heavy) use of tail recursive functions and the presence of many pure numeric functions or sub-expressions. Therefore, in the second step, we focus on improving the performance of the compiler by optimising tail recursions and numeric functions and sub-expressions. The resulting compiler is again compared with Clean and Haskell and the basic compiler using the same set of benchmarks. It turns out that the resulting performance is now acceptable in almost all cases.

Summarising, the contributions of this study are the stepwise construction of a simple compiler for a lazy (intermediate) functional programming language with the following characteristics:

– The compilers translates to concise and readable C++ functions (for a functional programmer knowing C++) that are in 1-1 correspondence with the original functions. The C++ functions give the programmer clear insight in how constructs from functional programming language are implemented.
– It gives the reader insight in what kind of optimisations are important for obtaining an efficient implementation of lazy functional languages.
– The user can easily add functions to the generated code and can modify generated functions to experiment with alternative optimisations.
– The performance of the resulting programs is in many cases competitive with that of Clean and Haskell.

The structure of this paper is as follows. In Section 2 we introduce the intermediate functional programming language **SAPL**. In Section 3 we sketch an interpreter for SAPL. This interpreter is the starting point for the construction of the compiler. The compiler is described in Section 4. We describe the compiler in a number of steps. First a basic version of the compiler is introduced that is a straightforward and simple extension of the interpreter. The performance of a set of benchmarks compiled with this compiler and the Clean and GHC compiler is used to make a comparison. The results of this comparison are analysed and this leads to the proposal of a number of candidate optimisations that are implemented. In the last section we give some conclusions.

2 The SAPL Programming Language

SAPL stands for **S**imple **A**pplication **P**rogramming **L**anguage. The basic version of SAPL has function application as only operation. SAPL is a simple functional programming language that can be used as an intermediate formalism for the interpretation of functional programming languages like Haskell and Clean. The main difference between SAPL and the intermediate formalisms normally used for these languages is the absence of algebraic data types and constructs for pattern matching in SAPL. This makes SAPL a compact and simple language. More details about SAPL can be found in [2].

In [2] we also showed how to represent data types and pattern-based function definitions in SAPL. Here we shortly repeat the definition of the list data type together with the *length* function.

$$\begin{array}{lcl} Nil & = & \lambda f\ g \rightarrow f \\ Cons\ x\ xs & = & \lambda f\ g \rightarrow g\ x\ xs \\ length\ ys & = & ys\ 0\ (\lambda x\ xs \rightarrow 1 + length\ xs) \end{array}$$

Now consider a pattern based Haskel function like *mappair*.

$$\begin{array}{llll} mappair\ f\ Nil & zs & = & Nil \\ mappair\ f\ (Cons\ x\ xs) & Nil & = & Nil \\ mappair\ f\ (Cons\ x\ xs) & (Cons\ y\ ys) & = & Cons\ (f\ x\ y)\ (mappair\ f\ xs\ ys) \end{array}$$

This definition can be transformed to the following SAPL function (using the above definitions of *Nil* and *Cons*).

$$\begin{array}{lcl} mappair\ f\ as\ zs & = & as\ Nil\ (\lambda x\ xs \rightarrow zs\ Nil\ (\lambda y\ ys \rightarrow \\ & & \quad Cons\ (f\ x\ y)\ (mappair\ f\ xs\ ys))) \end{array}$$

3 An Interpreter for SAPL

The only operations in SAPL programs are function application and a number of (build-in) integer operations. Therefore an interpreter can be kept small and elegant. The interpreter is based on straightforward graph reduction techniques as described in Peyton Jones [4], Plasmeijer and van Eekelen [5] and Kluge [3]. We assume that a pre-compiler has eliminated all algebraic data types and pattern definitions (as described earlier), removed all let(rec)- and where- clauses and lifted all lambda expressions to the global level. Only constant let-expressions are allowed to enable sharing and cyclic expressions. The interpreter is only capable of executing function rewriting and the basic operations on integers. The most important features of the interpreter are:

- It uses 4 types of memory Cells. A Cell corresponds to a node in the syntax tree and is either an: Integer, (Binary) Application, Variable or Function Call. To keep memory management simple, all Cells have the same size. A type byte in the Cell distinguishes between the different types. Each Cell uses 12 bytes of memory.
- The memory heap consists only of Cells. The heap has a fixed size, definable at start-up. We use mark and sweep garbage collection.
- It uses a single argument stack containing only references to Cells. The C (function) stack is used as the dump for keeping intermediate results when evaluating strict functions (numeric operations only).

- The state of the interpreter consists of the stack, the heap, the dump, an array of function definitions and a reference to the node to be evaluated next. In each state the next step to be taken depends on the type of the current node: either an application node or a function node.
- It reduces an expression to head-normal-form. The printing routine causes further reduction. This is only necessary for arguments of curried functions.

The interpreter pushes arguments on the stack until a function call is met. In that case the function body is instantiated while the arguments are substituted, the top application node is overwritten and evaluation continues on the new expression until we arrive at a curried call or an integer value.

3.1 Optimisations in the Interpreter

The interpreter can be optimised in several ways. Simple optimisations are the use of a more efficient memory representations of function calls with 1 or 2 arguments and the marking of curried calls (if possible) to avoid the useless evaluation of them. Applying these optimisations result in speed-ups up to 50%.

A more significant optimisation can be realized by marking the application of a function representing an algebraic data type element to its arguments by the keyword *select* (semantically equivalent to the identity function). This triggers the interpreter not to instantiate the entire function body at once, but first to evaluate the data type and only *select* and instantiate the relevant part of the remainder expression (more details can be found in [2]).

As a last optimisation, anonymous functions that are the argument of a *select* are not lifted to the global level, but are called inline (see [2]).

As an example we show how the *select* optimisation is applied in the *mappair* function (the lambda expressions in this example are not lifted to the global level).

$$\begin{aligned} &\mathit{mappair}\ f\ \mathit{as}\ \mathit{zs}\ = \\ &\qquad\qquad \mathit{select}\ \mathit{as}\ \mathit{Nil}\ (\lambda\ x\ \mathit{xs}\ \rightarrow \\ &\qquad\qquad \mathit{select}\ \mathit{zs}\ \mathit{Nil}\ (\lambda\ y\ \mathit{ys}\ \rightarrow\ \mathit{Cons}\ (f\ x\ y)\ (\mathit{mappair}\ f\ \mathit{xs}\ \mathit{ys}))) \end{aligned}$$

The *select* optimisation is essential and may result in speed-ups of more than 100 times. Normally the *select* annotations are added while translating Haskell or Clean programs to SAPL, but it is possible to add the *select* annotations during a compile time analysis of a SAPL program. During this analysis it is determined where applications of data type functions to other arguments occur. This analysis can only be performed in case of complete programs and not for separately compiled files (modules). For example, if we consider the definition of *mappair* in isolation it is not clear that *as* and *zs* are *selectors*. One needs an example of the usage of *mappair* to determine that.

3.2 Considerations

The interpreter without the *select* optimisation and the integer operations is a pure graph reductor. The only operations are graph reduction (push arguments on the stack until a function call is met) and graph instantiation (copy a function body and meanwhile substitute the arguments from the stack).

Numeric operations are strict in the sense that the arguments have to be evaluated before the operation can be performed. The same holds for the *select* optimisation. Also in this case the first argument of *select* has to be evaluated before the operation (selection of the appropriate argument) can take place. The optimisation prevents the instantiation of large graphs. In the remainder of this paper we show that many of the optimisations we implement in the compiler involve the use of strictness to prevent the instantiation of unnecessary graphs.

4 A SAPL Compiler

We present two versions of the compiler: a basic version and an optimised version. The optimisations are a result of an analyses of the performance of the basic version for a number of benchmarks.

The benchmarks we use for the comparison are the same we used for comparing the SAPL interpreter with several other interpreters and compilers in [2]. We briefly repeat the description of the benchmarks (their code can be found in [9]):

1. **Prime Sieve.** The prime number sieve program (*primes !! 5000*).
2. **Symbolic Primes.** Prime sieve using Peano numbers (*sprimes !! p280*).
3. **Interpreter.** A small SAPL interpreter. As an example we coded the prime number sieve for this interpreter and calculated the 100th prime number.
4. **Fibonacci.** The (naive) Fibonacci function, calculating *fib 35*.
5. **Match.** Nested pattern matching (5 levels deep), repeated 2000000 times.
6. **Hamming.** The generation of the list of Hamming numbers (a cyclic definition) and taking the 1000th Hamming number, repeated 10000 times.
7. **Twice.** A higher order function (*twice twice twice twice (add 1) 0*), repeated 400 times.
8. **Queens.** Number of placements of 11 Queens on a 11 * 11 chess board.
9. **Knights.** Finding all Knight tours on a 5 * 5 chess board.
10. **Parser Combinators.** A parser for Prolog programs based on Parser Combinators parsing a 17000 lines Prolog program.
11. **Prolog.** A small Prolog interpreter based on unification only (no arithmetic operations), calculating all descendants in a six generations family tree.
12. **Sorting.** Quick Sort (20000 elements), Merge Sort (200000 elements) and Insertion Sort (10000 elements).

Three of the benchmarks (*Interpreter*, *Prolog* and *Parser Combinators*) are realistic programs, the others are typical benchmark programs that are often used for comparing implementations.

We use C++ as a target language for our compiler. We do not use the object oriented properties of C++ (classes and member functions). But we use some specific features of C++ like reference variables. In all versions of the compiler there is a one-to-one correspondence between SAPL and C(++) functions. Because we want to use the compiler for educational purposes we strive at readable and understandable generated code.

The generic structure of a translated function is:

$$int\ funcname(Reduct\ t)\ \{\ instantiate\_body;\ \ return\ eval\_body;\ \}$$

Here *funcname* is the name of the translated SAPL function. We assume that all arguments of a function are already on the stack when the function is called. The argument t of the function is a reference to the top node of the call for this function. To enable sharing we have to overwrite this top node with the result of the function. The function returns an integer. This is because functions that result in an algebraic data type have to return the selection number needed in a *select* construction. Because we want to use the same type signature for all functions, all functions have to return an integer. Note that we cannot give the C function the same arguments as the original function because we can make curried calls to a function which is, of course, not possible in C.

4.1 A Basic SAPL Compiler

If we take a closer look at the SAPL interpreter, the most obvious candidate for compilation is the instantiation of function bodies. The interpreter uses a recursive function *instantiate* to copy the body and substitute the arguments. It is straightforward to generate C++ code that does this instantiation directly.

Due to the *select* optimisation the body of a function containing a *select* is not copied at once but in parts. Therefore, in the translation to C++, we add the control structure (using *if* or *switch/case* statements) to enable this copying in parts. Also the generation of this control structure is entirely straightforward.

Examples. As an example consider the translation of the functions *sieve* and *el* from the prime number sieve program.

$$\begin{aligned} &sieve\ xs &&= cons\ (hd\ xs)\ (sieve\ (filter\ (nmz\ (hd\ xs))\ (tl\ xs))) \\ &el\ n\ xs &&= select\ xs\ error\ (\lambda\ a\ as\ \rightarrow\ if\ (eq\ n\ 0)\ a\ (el\ (sub\ n\ 1)\ as)) \end{aligned}$$

The translation of *sieve* results in:

```
int sieve(Reduct t) {
  testmem();
  setCell(t,SELB,newR(OPFUNC,get(0),0,9),newR(OFUNC,
     newR(BPFUNC,newR(OPFUNC,newR(OPFUNC,get(0),0,9),0,7),
     newR(OPFUNC,get(0),0,10),3),0,5),2);
  pop(1);
  return eval(t);
}
```

testmem() checks if garbage collection is necessary. This check is done before every body instantiation. *setCell(t,...)* overwrites *t*. Although the *setCell* call looks quite complicated the only thing that is happening here is the allocation of a new graph in memory. Due to the memory optimisations for applications with one and two arguments and the marking of curried applications there are a large number of cell types (*SELB*, *OPFUNC*, etc.). *get(i)* returns a reference to the *i-th* element on the stack. *pop(i)* removes *i* elements from the stack. In the last line *eval(t)* recursively starts evaluating the resulting expression. The only thing the *eval* function does is pushing arguments on the stack and calling the resulting function.

The translation of *el* results in:

```
int el(Reduct t) {
  Reduct res = get(1);
  if(eval(res)) {
    pushs(res->r); pushs(res->l);
    testmem();
    res = newR(BINOPER,get(2),newR(NUM,Reduct(0),0),5);
    if(eval(res)) {
      testmem();
      setCell(t,BPFUNC,newR(BINOPER,get(2),
                       newR(NUM,Reduct(1),0),1),get(1),4);
      pop(4);
    }
    else {overwrite(t,get(0)); pop(4);}
  }
  else    {setCell(t,SFUNC,0,Reduct(0),0);   pop(2);}
  return eval(t);
}
```

In this example we see that the control structure of the original function is clearly reflected in the C++ function. In the first line *xs* is assigned to *res*. *res* is evaluated. In case the result is a *cons* (returns 1) the arguments of *cons* are pushed on the stack. Next the expression *eq n 0* is instantiated and evaluated. If *n != 0* the expression *el (sub n 1) xs* is instantiated and the stack is cleared. In case *n == 0*, *t* is overwritten with *x*. Also in this case the stack is cleared. The last *else* handles the case that the list was *nil*.

We conclude that the basic compiler results in concise code that clearly reflects how the graph reduction process is conducted. For a function acting on a data structure with 3 or more cases a C++ *switch* statement is generated. The adaptations to the interpreter needed to generate the C++ functions are modest. An interesting aspect is that the resulting C++ functions are integrated in the interpreter environment. The only difference for the user is the increase in speed (and an extra compilation round before starting the interpreter).

Although the Basic Compiler compiles to C++, it is essentially still an interpreter. The way graphs are reduced is the same as in the original interpreter.

In the remainder of this paper we sometimes abbreviate the instantiation of graphs with: `instantiate('expression')` or `overwrite(t,'expression')`.

	Pri	Sym	Inter	Fib	Match	Ham	Twi	Qns	Kns	Parse	Plog	Qsort	Isort	Msort
SAPL Int	6.1	17.6	7.8	7.3	8.5	15.7	7.9	6.5	47.1	4.4	4.0	16.4	9.4	4.4
SAPL Bas	4.3	13.2	6.0	6.5	5.9	9.8	5.6	5.1	38.3	3.8	2.6	10.1	6.7	2.6
GHC	2.0	1.7	8.2	4.0	4,1	8.4	6.6	3.7	17.7	2.8	0.7	4.4	2.3	3.2
GHC -O	0.9	1.5	1.8	0.2	1.0	4.0	0.1	0.4	5.7	1.9	0.4	3.2	1.9	1.0
Clean	0.9	0.8	0.8	0.2	1.4	2.4	2.4	0.4	3.0	4.5	0.4	1.6	1.0	0.6

Fig. 1. Comparison Speed of Basic Compiler (Time in seconds)

4.2 Performance of the Basic Compiler

In Fig. 1 we compare the performance of the basic compiler with that of the interpreter and of the GHC and Clean compilers. If we compare the basic compiler with the interpreter we see that the basic compiler is about 40% faster (speed-ups between 10 and 60%).

If we compare the basic compiler with GHC (without optimiser) we see that in three cases (*Interpreter*, *Mergesort* and *Twice*) the basic SAPL compiler is already faster. In the other cases GHC is mostly less than 2 times faster. Relatively slow SAPL benchmarks are *Symbolic Primes* (7 times) and *Prolog* (3.7 times).

Comparing the basic compiler with GHC -O and Clean we measure large differences in performance, varying from 10% faster (compared to *Parser Combinators* in Clean) to more than 30 times slower (*Fibonacci* for Clean, GHC -O and *Twice* for GHC -O).

4.3 Analysis of Basic Compiler

Compared with GHC (without optimiser) the Basic Compiler is already doing a reasonable job. The only poor performing benchmark is *Symbolic Primes*. This is an a-typical program, because there is no integer arithmetic in this example and the functions bodies are all very small. For SAPL this means a lot of interpretation overhead. More important, the performance dominating functions *Mod* and *Subtract* are tail recursive. In the sequel we show that, using tail recursion optimisation, the performance of this benchmark can be improved significantly.

If we take a closer look at the benchmarks for the comparison with GHC -O and Clean, we see that there is only one benchmark that performs good in this comparison: *Parser Combinators*. This is the most 'functional' of all benchmarks in the sense that it manipulates mostly higher order functions. For a compiler this means that a lot of closures must be maintained. Closures are represented by structures comparable to the graphs in SAPL. Every compiler should analyse (destruct) these closures at a certain moment in a way similar to the way the Basic SAPL compiler does this.

The worst performing benchmarks are: *Symbolic Primes*, *Fibonacci*, *Queens* and *Twice*.

– **Symbolic Primes** we already discussed above. It contains a number of tail recursive functions for which SAPL does no optimisations yet.

- **Fibonacci** is a pure numeric function (numeric arguments and numeric operations only). In SAPL every time the function is called in the recursion, a complete instantiation of the function body is made (on the heap). The Clean and GHC -O compilers optimise this function and do not use closures but instead only use the stack to execute it.
- **Queens** has a number of numeric sub-expressions and has a (hidden) tail recursion in function *safe*. Also in this case Clean and GHC -O use strictness analysis to eliminate the building of many closures.
- **Twice** is a special case. GHC -O has a much better performance than both SAPL and Clean. If we study the generated code for GHC -O we see that some very specific inline optimisations are made. We did not make any special optimisations for this example.

Conclusions and Plan for Optimisations. The basic compiler has already a nice performance for programs manipulating mostly higher order functions. Therefore, we may expect that the poorer performance is caused by the overhead involved in building instantiations (closures) that are not really necessary. The optimisations we apply are aimed at either preventing the building of closures or at building smaller closures. In the light of the discussion above we focus on tail recursive functions and on numeric functions and (sub)expressions, also because they can be recognized and optimised easily. But before that we look at some straightforward optimisations.

4.4 Reducing the Size of Closures and Removal of Interpretation Overhead

Consider the following function g:

$$g\ a\ b\ c\ d\ =\ f\ a\,(h\ b\ c)\ d$$

In the basic compiler this is compiled to:

```
int g(Reduct t) {
  testmem();
  setCell(t,APP,newR(APP,newR(APP,newR(FUNC,0,0,2),get(0)),
            newR(BFUNC,get(1),get(2),1)),get(3));pop(4);
  return eval(t);
}
```

In the body of g a large instantiation is build for which *eval* is called immediately. *eval* pushes the arguments of f on the stack and calls the function f. But if we already know this, we can hard code the pushing of the arguments and the call to f. In this way we both save instantiation and interpretation overhead.

```
int g(Reduct t) {
  testmem();
  Reduct a0,a1,a2;
  a0 = get(0);
  a1 = newR(BFUNC,get(1),get(2),1);
  a2 = get(3);
  pop(4);
  pushs(a2);pushs(a1);pushs(a0);
  return f(t);
}
```

In this example the number of allocated nodes is reduced from 4 to 1!

We apply this optimisation whenever possible. This means that an, at compile time, known function should be called with enough arguments.

4.5 Numerical Functions and Expressions

If a function has numeric arguments only and its body is a pure numerical expression we can avoid the creation of closures altogether. Consider for example the Fibonacci function:

$$\mathit{fib}\; n \;=\; \mathit{if}\;(n \;<\; 2)\;1\;(\mathit{fib}\;(n \;-\; 1) \;+\; \mathit{fib}\;(n \;-\; 2))$$

The Basic SAPL compiler translates this to:

```
int fib(Reduct t) {
  Reduct res;
  testmem();
  res = newR(BINOPER,newR(NUM,Reduct(2),0),get(0),7);
  if(eval(res)) {
    testmem();
    setCell(t,BINOPER,newR(OPFUNC,newR(BINOPER,get(0),
                               newR(NUM,Reduct(1),0),1),0,35),
                      newR(OPFUNC,newR(BINOPER,get(0),
                      newR(NUM,Reduct(2),0),1),0,35),0);
    pop(1);
  }
  else {
    setCell(t,NUM,Reduct(1),0);
    pop(1);
  }
  return eval(t);
}
```

In the optimised translation *fib* is translated to:

```
int fibh(int n) {
  if (n < 2) return 1;
  else return fibh(n-1) + fibh(n-2);
}

int fib(Reduct t) {
  eval(get(0));
  setCell(t,NUM,Reduct(fibh(getNum(get(0)))),0);
  pop(1);
  return 0;
}
```

fibh is a pure C++ function without any instantiations of cells and *fib* is a wrapper function for calling *fibh* from a functional context. The speed-up obtained in this way is more than 30 times. This version of *fib* now has a performance comparable to that of Clean and GHC -O.

Numerical expressions with a Boolean result. A special case of numeric expressions are those with a Boolean result. They often occur in the condition of an *if* statement. The *el* function we studied already before is an example of such a function. Using the numeric expression optimisation the compiled function becomes:

```
int el(Reduct t) {
  Reduct res = get(1);
  if(eval(res)) {
    pushs(res->r); pushs(res->l);
    eval(get(2));
    if(getNum(get(2) == 0){overwrite(t,get(0)); pop(4);}
    else {
      testmem();
      setCell(t,BPFUNC,newR(BINOPER,get(2),
                       newR(NUM,Reduct(1),0),1),get(1),4);
      pop(4);
    }
  }
  else    {setCell(t,SFUNC,0,Reduct(0),0);   pop(2);}
  return eval(t);
}
```

This saves allocation and interpretation overhead.

4.6 Optimising Tail Recursion Functions

Replacing tail recursions by while loops are a common optimisation also applied for strict functional and imperative languages. In these cases the optimisation is used to eliminate calling and stack overhead. But in the lazy functional context we have an extra benefit. Also the building of a closure (and the destruction of it) for the recursive call is prevented. Therefore, the speed-up is even higher.

Simple tail recursive functions have the form:

$$f\ a\ arg = if\ (cond\ a)\ (default\ a\ arg)\ (f\ (dec\ a)\ (update\ a\ arg))$$

The recursion runs over *a*. For the sake of simplicity we assume that there is only one other argument. The function contains a simple *if* construction at the top level. In the *else* case the same function is called with an *a* argument that is in some way smaller than the original argument. We compile this function to a C++ function containing a while-loop.

```
int f(Reduct t) {
  Reduct res  = instantiate(‘cond a’);
  Reduct &a   = get(0);
  Reduct &arg = get(1);
  while(eval(res)) {
      arg  = instantiate('update a arg');
      a    = instantiate('dec a');
      res  = instantiate('cond a');
    }
  overwrite(t,'default a arg'); pop(2);
  return eval(t);
}
```

Note that we use reference variables for *a* and *arg*, so they remain on the SAPL stack, which is necessary for garbage collection purpose. In the while loop we instantiate the new versions of the arguments and the condition. The while condition determines if the recursion is finished. Because the arguments of the tail recursion are maintained by variables we can easily optimise numeric or Boolean arguments (see Subsection 4.5). As an example, consider the function *length* (note the use of an accumulating parameter).

$$\mathit{length}\ n\ \mathit{xs}\ =\ \mathit{select}\ \mathit{xs}\ n\ (\lambda\ a\ \mathit{as}\ \rightarrow\ \mathit{length}\ (n\ +\ 1)\ \mathit{as})$$

This function is translated to:

```
int length(Reduct t) {
  eval(get(0));
  int n = getNum(get(0));
  Reduct &xs = get(1);
  while(eval(xs)) {
    n = n + 1;  xs = xs -> r;
  }
  overwrite(t,newR(NUM,Reduct(n),0)); pop(2); return 0;
}
```

Here the argument *n* is numerical and therefore assigned to the *int* variable *n*. The expression *n+1* is not instantiated, but directly translated to C. This saves an instantiation and a reduction. After the while loop we have to wrap the numeric result in a cell.

Note that this function also does not build the large closure *0+1+1+1+..* that is only evaluated at the end, which happens in the SAPL interpreter and the Basic Compiler. In this way a basic form of strictness analysis is realized. Furthermore, there is another optimisation. The arguments of *Cons* are not pushed on the stack, but can be found as the left and right child of *xs*. In the while loop of this function no instantiations are made.

A tail recursion may also runs over several arguments. In that case the condition is a conjunction of all the conditions. As an example, consider the following definitions of *Zero* and *Suc* and the tail recursive function *Sub* running over 2 arguments, all occurring in the *Symbolic Primes* benchmark:

$$\begin{aligned}&\mathit{Zero}\ f\ g\ =\ f\\ &\mathit{Suc}\ n\ f\ g\ =\ g\ n\\ &\mathit{Sub}\ m\ n\ =\ \mathit{select}\ n\ m\ (\lambda\ \mathit{pn}\ \rightarrow\ \mathit{select}\ m\ \mathit{Zero}\ (\lambda\ \mathit{pm}\ \rightarrow\ \mathit{Sub}\ \mathit{pm}\ \mathit{pn}))\end{aligned}$$

Sub is translated to:

```
int Sub(Reduct t) {
  Reduct &m = get(0);
  Reduct &n = get(1);
  while(eval(n) && eval(m)) {
    m = m -> l;
    n = n -> l;
  }
  if(eval(n)) {
    overwrite(t,‘Zero’);pop(2);return 0;
  }
  else {
    overwrite(t,‘m’);pop(2);return eval(t);
  }
}
```

Note that after the while we have ‘to check’ why the loop stopped to return the result of the right stopping case. Note also that we made use of the fact that the && operator in C++ is conditional (lazy). Again, no instantiations are made in the while loop.

Tail recursion that run over 3 or more variables are handled in a similar way.

Hidden Tail Recursions. Sometimes a function can be easily converted to a tail recursion. For example in the *safe* function used in the *Queens* benchmark an *and* condition with a recursive call to *safe* itself occurs.

$$\begin{aligned} \mathit{safe}\ xs\ d\ x\ =&\mathit{select}\ xs\ \mathit{True} \\ &(\lambda\ y\ ys\ \rightarrow\ \ \mathit{and}\ (\mathit{and}\ (\mathit{neq}\ x\ y)\ (\mathit{neq}\ (\mathit{add}\ x\ d)\ y)) \\ &\qquad\qquad\qquad (\mathit{and}\ (\mathit{neq}\ (\mathit{sub}\ x\ d)\ y)\ (\mathit{safe}\ ys\ (\mathit{add}\ d\ 1)\ x))) \end{aligned}$$

safe is translated to:

```
int safe(Reduct t) {
  Reduct xs = get(0);
  eval(get(1)); eval(get(2));
  int d = getNum(get(1));
  int x = getNum(get(2));
  int y;
  while(eval(xs) && (eval(xs -> l),y = getNum(xs -> l),x != y) &&
                    (x + d != y) && (x - d != y)) {
    xs = xs -> r;
    d = d + 1;
  }
  if (eval(xs)) {
      setCell(t,FALSE,0,0);
      pop(3);
      return 1;
    }
    else {
      setCell(t,TRUE,0,0);
      pop(3);
      return 0;
    }
}
```

Also in this case we make use of the conditionality of the && operator in C++.

4.7 Results and Discussion

Figure 2 gives the results of the comparison of the optimised compiler with the other compilers and the Interpreter. We see that the optimisations result in a significant speed-up in almost all cases. We briefly discuss the speed-up obtained for the benchmarks.

1. **Prime Sieve.** Speed-up 1.65: numeric optimisations and a tail recursion in *elem*.
2. **Symbolic Primes.** Speed-up 7.3: tail recursions in functions *Mod*, *Gt*, *Neq* and *Sub*.
3. **Interpreter.** Speed-up 1.82: tail recursions in *length*, *drop* and *elem* and several small numeric optimisations.
4. **Fibonacci.** Speed-up 33: pure numeric function.
5. **Match.** Speed-up 1.9: numeric optimisations.
6. **Hamming.** Speed-up 1.66: small numeric optimisations.
7. **Twice.** Speed-up 1.24: small numeric optimisations.
8. **Queens.** Speed-up 5.7: tail recursion in *safe* and several numeric optimisations.
9. **Knights.** Speed-up 2.1: numeric optimisations.
10. **Parser Combinators.** Speed-up 1.3: small numeric optimisations and minor tail recursions.
11. **Prolog.** Speed-up 2.0: tail recursions in several (minor) functions and some numeric optimisations.
12. **Sorting.** Quick Sort (1.7), Merge Sort (2.2) and Insertion Sort (2.7): numeric optimisations.

	Pri	Sym	Inter	Fib	Match	Ham	Twi	Qns	Kns	Parse	Plog	Qsort	Isort	Msort
SAPL Int	6.1	17.6	7.8	7.3	8.5	15.7	7.9	6.5	47.1	4.4	4.0	16.4	9.4	4.4
SAPL Bas	4.3	13.2	6.0	6.5	5.9	9.8	5.6	5.1	38.3	3.8	2.6	10.1	6.7	2.6
SAPL Opt	2.6	1.8	3.3	0.2	3.1	5.9	4.5	0.9	18.0	2.9	1.3	6.0	2.5	1.2
GHC	2.0	1.7	8.2	4.0	4,1	8.4	6.6	3.7	17.7	2.8	0.7	4.4	2.3	3.2
GHC -O	0.9	1.5	1.8	0.2	1.0	4.0	0.1	0.4	5.7	1.9	0.4	3.2	1.9	1.0
Clean	0.9	0.8	0.8	0.2	1.4	2.4	2.4	0.4	3.0	4.5	0.4	1.6	1.0	0.6

Fig. 2. Comparison Speed of Optimized Compiler (Time in seconds)

Even for the higher order examples *Twice* and *Parser Combinators* there is a (small) speed-up due to the numeric optimisations. The greatest speed-up is obtained for the *Fibonacci* benchmark. An interesting speed-up is obtained for the *Symbolic Primes* benchmark. This result could be obtained because the functions *Mod* and *Sub* are tail recursive and dominate the performance of the benchmark. Also for *Queens* a high speed-up is obtained because the tail recursive *safe* function dominates the performance.

Compared with GHC the optimised compiler is faster in almost all cases. Only for *Primes*, *Prolog* and *QSort* GHC is slightly faster. For *Fibonacci*, *Interpreter*, *Queens* and *Mergesort* the optimised SAPL compiler is much faster (more than 2.5 times).

Compared with GHC -O we see that only for *Twice* GHC -O is an order of magnitude faster (45 times). The GHC -O optimiser recognizes the repetition in this higher order function and replaces it with an iteration. Note that GHC -O is also much faster than Clean in this case. In all other cases the difference is less than 3 times and in several cases SAPL is even competitive. On the average the difference in performance stays within a factor of 2.

Compared with Clean we see that the greatest difference in performance stays within a factor of 6 (*Knights*). On the average Clean is about 2.5 times faster. For *Parser Combinators* the SAPL compiler is faster (1.5 times).

Considering only the more realistic applications (*Interpreter*, *Parser Combinators* and *Prolog*) we see that for *Parser Combinators* the SAPL compiler has competitive performance. For *Interpreter* the SAPL compiler is competitive with GHC and GHC -O but is 4 times slower than Clean. In case of *Prolog* the SAPL compiler is significant slower than all others. This is not surprising, because the performance dominating function *unify* in *Prolog* cannot be optimised with the techniques used in the SAPL compiler. Here more sophisticated optimisations based on strictness analyses are needed.

5 Conclusions

In this paper we presented a compiler for lazy functional languages for educational and experimental use, based on a straightforward interpreter. For optimising this compiler we did not use the more sophisticated techniques normally used for compilers but took a more opportunistic approach, applying only two easy to detect and apply optimisations. This has as an advantage that the generated functions have a simple structure. This makes it possible for the user to inspect how the optimisations are applied and it also enables the user to experiment with other (hand-made) optimisations.

The compiler generates comprehensible C++ code that gives the programmer clear insight in how contructs from functional programming languages are implemented. This in contrast with the GHC compiler that also uses C as an intermediate language, but for which the generated C code is difficult to understand and looks more like assembly than like an ordinary C program.

We have learned that sometimes applying simple optimisations result in significant speed-ups (e.g *Fibonacci* and *Symbolic Primes*), but in other cases the optimisations do not suffice. In these examples (e.g. *Prolog*) the difference with Clean and GHC is still too big. We also learned that optimising a function always boils down to trying to prevent the building of unnecessary graphs (closures). In our approach this was always realized by replacing 'functional code' by 'imperative code' in the generated C++ functions.

An interesting question is, if it is possible to extend the set of optimisations in such a way that the performance becomes competitive to that of GHC and Clean in all cases while maintaining readable and comprehensive generated code.

References

1. The Haskell Home Page, `http://www.Haskell.org`
2. Martin Jansen, J.M., Koopman, P., Plasmeijer, R.: Efficient interpretation by transforming data types and patterns to functions. In: Nilsson, H. (ed.) Proceedings Seventh Symposium on Trends in Functional Programming, TFP 2006, Trends in Functional Programming, Nottingham, UK, April 19-21, vol. 7. Intellect Publisher (2006)
3. Kluge, W.: Abstract Computing Machines. In: Texts in Theoretical Computer Science. Springer, Heidelberg (2004)
4. Peyton Jones, S.L.: The Implementation of Functional Programming Languages. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1987)
5. Plasmeijer, R., van Eekelen, M.: Functional Programming and Parallel Graph Rewriting. International Computer Science Series. Addison-Wesley, Reading (1993)
6. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: Executable specifications of interactive work flow systems for the web. In: Ramsey, N. (ed.) Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming, CFP 2007 of International Conference on Functional Programming, Freiburg, Germany, October 1-3, 2007, pp. 141–152. ACM, New York (2007)
7. Plasmeijer, R., Jansen, J.M., Koopman, P., Achten, P.: Declarative Ajax and client side evaluation of workflows using iTasks. In: Principles and Practice of Declarative Programming PPDP 2008, Valencia, Spain (July 2008)
8. Software Technology Research Group, Radboud University Nijmegen. The Clean Home Page, `www.cs.ru.nl/~clean`
9. Software Technology Research Group, Radboud University Nijmegen. The SAPL Home Page, `home.hetnet.nl/~janmartinjansen/sapl`

Author Index