Draft Proceedings of the Sixth

Advanced Functional Programming school (AFP 2008)

May 19-24 2008
Center Parcs "Het Heijderbos"
The Netherlands

Edited by

Pieter Koopman
Rinus Plasmeijer
Doaitse Swierstra

Technical report: ICIS- R08008, May, Radboud University Nijmegen, 2008

Preface

The 6th Advanced Functional Programming school (AFP 2008) is being held in Center Parcs "Het Heijderbos," The Netherlands near the city of Nijmegen on May 19-24, 2008. It is co-located with the 9th Symposium on Trends in Functional Programming (TFP 2008). The co-location of these two events is intended to attract new researchers to the exciting area of functional programming. TFP 2008 is organized by the Computer Science Department of the Radboud University Nijmegen and Utrecht University.

The goals of the Advanced Functional Programming schools are:

- Bring computer scientists, in particular young researchers and programmers, up to date with the latest functional programming techniques.
- Use advanced functional programming techniques in "programming in the real world".
- Bridge the gap between results presented at programming conferences and material from introductory textbooks on functional programming.

The approach we take to achieve these goals in the schools is:

- In depth lectures about a selected number of advanced functional programming techniques that emerged or established recently. The lectures are taught by experts in the field that actively contribute to research and application of the techniques.
- Lectures are accompanied by practical problems to be solved by the students at the school. The problems guide the students' learning to a great extent. This implies that there has to be a lab at the school site. After the popularization of the laptop and WiFi, this can be almost anywhere.
- Group work is stimulated, especially because the practical problems will typically be too large for a single person.

By functional programming we mean programming in a style that emphasizes the evaluation of expressions rather than the execution of commands.

This proceedings represents the papers associated with the lectures presented at AFP 2008. As in previous versions of AFP it is called the *draft* proceedings to distinguish it from the formal peer-reviewed post- proceedings that will be published as a LNCS volume by Springer.

AFP 2008 gratefully acknowledges the generous support of Getronics Pink Roccade, The Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO), the Netherlands Defence Academy and the Radboud University Nijmegen. We are particularly grateful to the lectures of this summer school.

Peter Achten, Pieter Koopman, Simone Meeuwsen, Rinus Plasmeijer, and Doaitse Swierstra.
Nijmegen, The Netherlands
May 2008

School Organisation

Editors: Pieter Koopman, Rinus Plasmeijer (both Radboud University Nijmegen, NL), Doaitse Swierstra (Utrecht University, NL)
Local arrangements: Peter Achten, Pieter Koopman, Simone Meeuwsen (all Radboud University Nijmegen, NL)

Support

www.getronics.com www.nwo.nl www.nlda.nl www.cs.ru.nl

Program

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Morning 1 8:30-10:15	Mark Jones	Richard Bird	Ulf Norell	Johan Jeuring	Simon Peyton Jones & Satnam Singh	Umut Acar
Morning 2 10:45-12:30	Rinus Plasmeijer	Ulf Norell	Richard Bird	Olivier Danvy	Umut Acar	Simon Peyton Jones & Satnam Singh
Lunch 12:30-14:00						
Afternoon 1 14:00-15:45	Mark Jones	Mark Jones	Ulf Norell	Johan Jeuring	Olivier Danvy	Umut Acar
Afternoon 2 16:15-18:00	Rinus Plasmeijer	Rinus Plasmeijer	Richard Bird	Olivier Danvy	Johan Jeuring	Simon Peyton Jones & Satnam Singh

Table of contents

Specifying Interactive Work Flows for the Web

Rinus Plasmeijer, Peter Achten, Pieter Koopman, Bas Lijnse, and Thomas van Noort

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

Abstract. In these lecture notes we present the iTask system: a set of combinators to specify *workflows* in a pure functional language at a very high level of abstraction. Workflow 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 workflow patterns commonly found in commercial workflow systems. In addition, we introduce novel workflow 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 workflows can be specified, and last but not least, the specification is used to generate an executable web-based multi-user workflow application. With the iTask system, useful workflows 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

Workflow 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 workflow 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 workflow 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 workflow systems:

– Workflow 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.
- Workflow 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 workflow 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 workflow 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 *workflows*. It is built on top of the iData toolkit, and both can be used within the same program. The library covers all known *workflow patterns* that are found in contemporary commercial workflow tools [21]. 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 [13, 1] features of the iData toolkit yields compact, robust, reusable and understandable code. Workflows are defined on a very high level of abstraction. It truly is an executable specification, as much is done and generated automatically.

As a running example, we will study the architecture of a *conference management* (CM) systems, and implement a small prototype. CM is a good case study of a workflow because it controls the activities of people with various roles, such as program chairs and program committee members. It is also challenging because many of these activities run in parallel, and the system should not hamper the activities of the workers of the 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 and exercises 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 Programming Workflows with iTasks

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

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

2.1 A simple example

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

```
module example                                         1.
                                                       2.
import StdEnv, StdiTasks                               3.
                                                       4.
Start :: *World → *World                               5.
Start world = singleUserTask [] simple world           6.
                                                       7.
simple :: Task Int                                     8.
simple = editTask "Done" createDefault                 9.
```

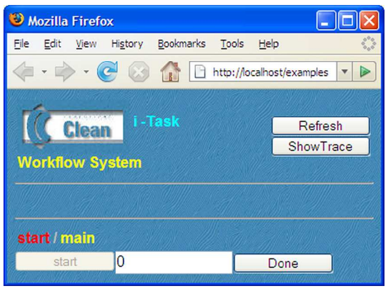


Fig. 1. An elementary `Int` iTask when started.

In line 3, the required modules are imported. `StdEnv` contains the standard functions, data structures, and type classes of Clean. `StdiTasks` 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 library function `singleUserTask` takes a workflow specification (here `simple`), provides it with a single worker infrastructure, and computes the corresponding HTML page that reflects the current state of the workflow system. In Sect. 2.8 we encounter the `multiUserTask` function that dresses up multi-user workflow 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 workflow 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           a | gForm {|*|}, iCreateAndPrint, iParse, iSpecialStore a
class iCreateAndPrint a | iCreate, iPrint a
class iCreate         a | gUpd  {|*|}       a
class iPrint          a | gPrint{|*|}       a
class iParse          a | gParse{|*|}       a
class iSpecialStore   a | TC                a
```

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 parse or de-serialize any value (`gParse`), 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 :: a | gUpd{|*|} a
```

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:

[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`.

```
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*
To get started quickly we have compiled a convenient distribution package which contains the latest Clean system for windows, all the additional iTask libraries and the examples and exercises for AFP2008.

Download this distribution at:

```
http://clean.cs.ru.nl/download/clean22/
     windows/Clean2.2-iTasks-AFP2008.zip
```

Unpack this zip archive and follow the instructions in the *"iTasks - Do Read This Read Me.doc"* file that can be found in the root folder of the archive.

When done, start the Clean IDE. Open the project file of the CM system case study, `CM.prj`. The project window should now be filled with all module names that the CM system uses. Compile and run the application. If everything is well, you should see a console window that asks you to open your browser and direct it to the given address. Follow this instruction, and you should be presented with the login screen of the CM system.

2.2 Playing with types

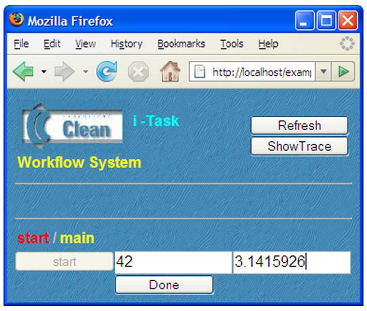
In this example we exploit the general purpose code of the previous example. The only modification we make is in line 8:

```
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).

In the CM case study, users are populated with *program chairs* (`Chair`) and *program members* (`PC`). We can define a record type, `User`, defined as:

```
:: User = { login  :: Login
          , email  :: TextInput
          , role   :: Role
          }
:: Role = Chair | PC
```



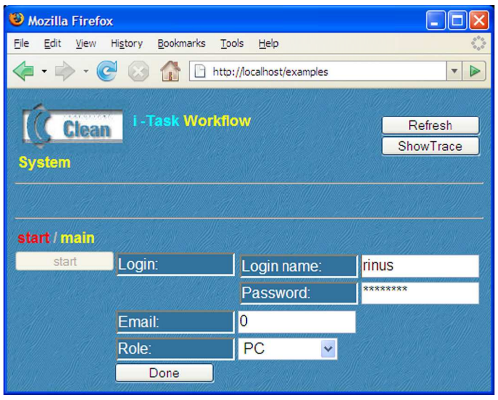
42 3.1415926

Fig. 2. An (`Int`,`Real`) iTask.

`Login` is a predefined algebraic data type for which an editor is created that allows the user to use a standard login input box for entering the account name and hidden password entry box. In order to use it, you need to include `iTaskUtil` to the **import** list at line 3. `TextInput` is also a predefined type for entering basic data (integers, reals, strings), and give the input box a desired width. In order to obtain an editor for `User` values we need to change the signature of `simple` into:

```
simple :: Task User                                                    8.
```

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



Login: Login name: rinus
Password: ********
Email: 0
Role: PC

Fig. 3. A `User` iTask in action.

Unfortunately, this does not compile successfully. A range of error messages is generated that complain that there are no instances of type `User`, `Role`, and `Login` for the generic `iData` class functions. The reason that the (`Int`,`Real`) example *does* compile, and the `User` 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 `User`, `Role`, and `Login` values as well, we only need to be polite and ask for them:

```
derive gForm  User, Role, Login
derive gUpd   User, Role, Login
derive gPrint User, Role, Login
derive gParse User, Role, Login
```

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. Copy the *"exercise1.icl"* file into the new directory, and rename it to *"exercise2.icl"*. Within the Clean IDE, open *"exercise2.icl"* and create a new project. Set the Environment to *"iTasks"*.

Define a new (set of) type(s), such as the `User` and `Role` 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, workflow 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 `<<@`:

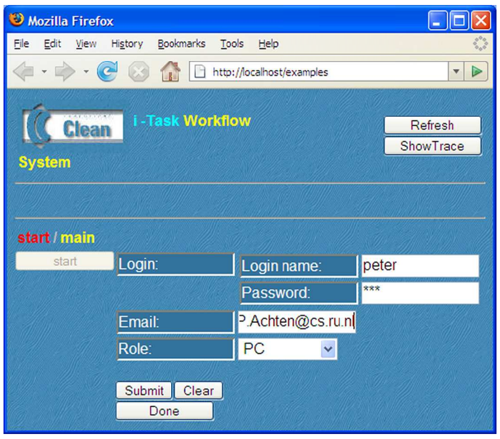
```
class    (<<@) infixl 3 b :: (Task a) b → Task a
instance <<@  Lifespan        // default: Session
            , Mode            // default: Edit
            , GarbageCollect  // default: Collect
            , StorageFormat   // default: PlainString

:: Lifespan       = Session | Page   | Database | TxtFile | TxtFileRO | DataFile
                  | Client  | 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). A novel attribute is to enforce *client side evaluation*, with the Client attribute. 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 Edited 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 Displayed 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 User
simple = editTask "Done" createDefault <<@ Submit <<@ TxtFile
```

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.



Login: Login name: peter
Password: ***
Email: P.Achten@cs.ru.nl
Role: PC

Fig. 4. A User iTask attributed to be a ‘classic’ form editor.

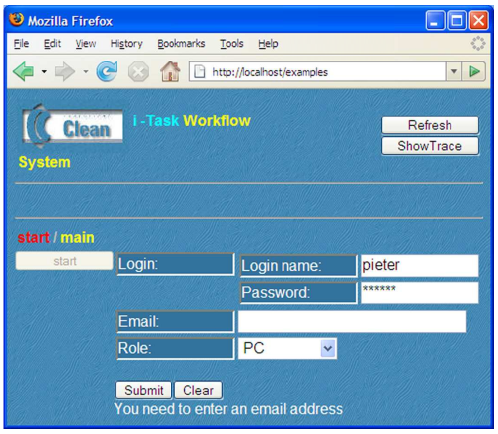
Editor tasks created with editTask allow the worker to enter any value, provided it is of the corresponding type of the editor. For many cases, this is sufficient. However, sometimes you wish to impose constraints on the edited values that cannot be expressed via the type system of Clean. Examples are editor tasks for even Int values, User values in which sensible values have been entered, and so on. For this purpose a predicate of type (a → (Bool, HtmlCode)) can be used to test the value of type a that is produced by the worker. If the value is correct, then the predicate returns True, otherwise it returns False and some explanation in the form of HtmlCode. The function editTaskPred does this:

```
editTaskPred :: !a !(a → (Bool, HtmlCode)) → Task a | iData a
```

The worker can edit values as usual, but these are checked with the predicate when the user submits the value. If the predicate does not hold, then the error message is displayed. Only if it holds, then then the editor task is finished, and the new value is propagated. Consider the following example from the CM case study:

```
simple :: Task User
simple = editTaskPred {createDefault & User.email = emptyTextInput} checkUser

checkUser :: User → (Bool, HtmlCode)
checkUser {User | login={loginName,password},email}
| loginName      == ""              = (False, [Txt "You need to enter a login name"])
| password       == PasswordBox "" = (False, [Txt "You need to enter a password"])
| fromTextInput email == ""         = (False, [Txt "You need to enter an email address"])
| otherwise                         = (True,[])
```

In this example, the predicate check checks a few simple properties of User values. Fig. 5 shows this editor task in action.



Login: Login name: pieter
Password: ******
Email:
Role: PC
You need to enter an email address

Fig. 5. A User iTask, now validating entered values.

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.

2.4 Simple database access

In the previous section we have shown that the programmer can decide where the state of a task editor is stored. This feature of editors can also be used to create a module for simple database access, which is named `iTasksDB.dcl`. We summarize the key ingredients of this module:

```
definition module iTasksDB

:: DBid a

mkDBid  :: !String !Lifespan → DBid a
readDB  :: !(DBid a)         → Task a | iData a
writeDB :: !(DBid a) !a      → Task a | iData a
```

(`mkDBid` *name* `Database`) returns a database identifier only if *name* is a proper file name, because the read and write operations will be performed on disk. (`mkDBid` *name lifespan*) (with *lifespan* ≠ `Database`) accepts any *name*. (`readDB` *name*) reads the current content of the identified database, and returns `createDefault` otherwise. (`writeDB` *name v*) sets the current content of the identified database to *v* and returns that value as well.

Suppose we wish to set up a `User` administration in the CM case study. We can introduce the following functions for that purpose (these are very similar to those in module `CMDatabase.icl`):

```
usersId      :: DBid [User]
usersId      = mkDBid "Users" TxtFile

readUsersDB  :: Task [User]
readUsersDB  = readDB usersId

writeUsersDB :: ([User] → Task [User])
writeUsersDB = writeDB usersId
```

We use them in the following section.

2.5 Sequencing with monads

In the previous examples, the workflow consisted of a single task. One obvious combination of workflows 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 =>> is the *bind* combinator, and `return_V` the *return* combinator. Hence, (m =>> $\lambda x \rightarrow n$) 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 #>> that is defined immediately in terms of =>> (m #>> $n \equiv m$ =>> λ _ $\rightarrow$ n) is provided as well.

As an example, we can extend the `User` adminstration that was given in Sect. 2.4 with a function to prepend a single user to the administration:

```
addUserDB :: User → Task [User]
addUserDB user = readUsersDB =>> λusers →
                 writeUsersDB [user:users]
```

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 =>> $\lambda v \rightarrow$*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.

The `example` at the end of Sect. 2.3 defined a `User` editor task with the predicate `checkUser`. With a minor change, it also checks whether the entered user value has a fresh name:

```
checkUser :: [User] User → (Bool, HtmlCode)
checkUser users {User | login={loginName,password},email}
| loginName == ""              = (False, [Txt "You need to enter a login name"])
| password  == PasswordBox ""  = (False, [Txt "You need to enter a password"])
| fromTextInput email == ""    = (False, [Txt "You need to enter an email address"])
| isMember loginName userNames = (False, [Txt "This login name already exists"]) // new
| otherwise                    = (True, [])
where
    userNames                  = [n \\ {User | login={loginName=n}} ← users]
```

With this predicate we can create a `User` editor task that tests for existing user names:

```
addUserForm :: Task User
addUserForm = readUsersDB =>> λusers →
              msg ?>> editTaskPred {createDefault & User.email = emptyTextInput}
                                   (checkUser users)
where msg   = [Txt "Please enter a username and password for the new user:"]
```

A sensible task for the program chair is to add users to the CM system. This can be expressed as:

```
addUser      :: Task Void
addUser      = addUserForm =>> λuser →
                addUserDB user #>>
                endMsg ?>> button "Ok"
where endMsg = [Txt "A new user has been added"]
```

Exercises

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

5. *To !>> or to ?>>*
Open the CM system project file, and find the function `addUser` (in the main module `CM.icl`). Alter the ?>> combinator into !>>. Compile and re-run the application. What is the effect of this change?

6. *Enter a prime number*
Create a workflow 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 `User` apart*
In Sect. 2.2, a `User` editor task was created with which complete `User` values can be edited. Create a new workflow in which the user has to enter values for the fields one by one, i.e. starting with the login name, and subsequently asking the password, email and role. Finally, the workflow should return the corresponding `User` value.

8. *Adding users*
Create a workflow that first asks the user a positive (but not too great) integer number n, and subsequently have him enter n values of type `User` (use the `seqTasks` combinator for this purpose – see Appendix A). When done, the workflow should display the names of these users.

2.6 Sequence and choice: breakable work

The monadic combinators presented in the previous section are useful for sequential composition. Obviously, realistic workflows also require *choice*, and this is provided by the iTask system with the following basic combinator:

```
(-||-) infixr 3 :: !(Task a) !(Task a) → Task a | iData a
```

(t_1 -||- t_2) is a task that terminates as soon as either t_1 or t_2 has terminated, or both.

The combination of monadic composition and choice leads to a number of useful derived combinators. Some of them have been defined in module `CMCombinators.icl` in the case study. Here we discuss some of them.

Tasks constructed with the monadic combinators rigidly force the worker to perform the given tasks in the prescribed order, and terminate only when the very last task has been performed. For real world cases, this is sometimes to restrictive: we want to model the fact that a worker can choose to *abort* her work. The `break` combinator models this behavior:

```
break :: (Task a) → Task (Maybe a) | iData a
break taska = (taska  =>> return_V o Just)
                       -||-
              (cancel #>> return_V Nothing)
```

(`break` t) is a task that performs t, and if that has terminated and yielded a value v yields (`Just` v). However, at any time before finishing t, the worker also has the choice to perform the `cancel` task, and return `Nothing` instead.

Together with `button`, `ok`, and `void`, `cancel` forms another group of tiny, but useful combinators:

```
button       :: String → Task Void
button label = editTask label Void

ok           :: Task Void
ok           = button "Ok"

cancel       :: Task Void
cancel       = button "Cancel" Void

void         :: Task Void
void         = return_V Void
```

`Void` is similar to Haskell's () value, and is defined as `:: Void = Void`.

The use of `Maybe` values, as done by `break`, is a common functional programming idiom. Because many tasks yield these values, it is useful to define an alternative =>> combinator:

```
try :: (Task (Maybe a)) (a → Task b) (Task b) → Task b | iData b
try taska taskfa taskb = taska =>> λx →
                           case x of
                             Nothing → taskb
                             Just x` → taskfa x`
```

(`try` *t succeed fail*) is a task that first performs *t*. If *t* succeeds, and yields (`Just` *v*), then the task proceeds as (*succeed v*). If *t* fails, and yields `Nothing`, then the task proceeds as *fail.* Another useful alternative =>> combinator is `breakable`:

```
breakable :: (Task a) (a → Task Void) → Task Void | iData a
breakable taska taskfa = try (break taska)
                             taskfa
                             void
```

(`breakable` *t succeed*) is a task that first performs *t*, while at the same time allowing the worker to abort *t*. If the worker chooses to finish *t* and yield a value *v*, then the task proceeds as (*succeed v*). If the worker chooses to abort *t* at any stage, the whole task returns `Void`.

As an example of this combinator, we can turn the `addUser` task for the program chair (defined at the end of Sect. 2.5) into a task that can be aborted:

```
addUser      :: Task Void
addUser      = breakable addUserForm
                         (λuser → addUserDB user #>>
                                  endMsg ?>> ok)
where endMsg = [Txt "A new user has been added"]
```

2.7 Recursive tasks

So far we have introduced sequential, monadic, composition and choice. The next key ingredient is to allow *recursive* workflow specifications. Recursion is fundamental to define computations that may run arbitrarily long. First we start with a useful combinator that can be found in the iTask API, `foreverTask`:

```
main :: User → Task Void
main user=:{User | login={loginName},role}
    = welcomeMsg ?>> foreverTask (chooseTask homeMsg userTasks)
where
    welcomeMsg  = [H1 [] ("Welcome " +++ loginName), Br]
    homeMsg     = [ Txt "Choose one of the tasks below or select a task that has been "
                  , Txt "assigned to you from the list on the left"
                  , Br, Br
                  ]
    userTasks   = case role of
                  Chair   = [ ("Show users",        showUsers)
                            , ("Add user",          addUser)
                            , ("Show papers",       showPapers)
                            , ("Assign reviewers",  assignReviewers)
                            , ("Judge papers",      judgePapers)
                            ]
                  PC      = [ ("Show papers",       showPapers)
                            , ("Mark papers",       markPapers user)
                            ]
```

(`foreverTask` *t*) repeats task *t* infinitely many times in sequence. It is used in this code fragment of the CM system to define the *main* part of the possible actions of a user, once he has successfully logged in. Because we do not know how long the user will keep logged in, she is offered a choice between several tasks infinitely many times. The `userTasks` function defines the possible tasks, depending on the `role` of the particular user.

2.8 Multi-User Workflows

So far the examples that have been shown are *single user* applications. Workflow systems usually involve arbitrarily many users. This is supported by the iTask system. The simplest way is to use the `multiUserTask` function, which has exactly the same type as the function `singleUserTask` that we have used so far. You can try this on any of your previous exercises and study the difference. However, most applications require some login ritual to allow only known users access to the application. Of course, the CM system is an example of such an application. For this purpose, a more elaborate function has been provided:

```
workFlowTask :: ![StartUpOptions]
                !(Task ((Bool,UserId),a))
                !(UserId a → LabeledTask b)
                !*HSt → (!Bool,Html,*HSt) | iData b
```

The second argument of `workFlowTask` is to determine whether the person who is attempting to log in is a known user, and return a `True` boolean value if so (as well as the user's `UserId` which is an integer value, and the initial data that that user requires). The third argument is the actual task that the user can continue to work on once successfully logged in. In the CM system case study, you can find this function right at the top at the `Start` function. The action that determines whether the user is known is called `public`, and the action that the user can continue with is called `main`, which we have already encountered in Sect. 2.7.

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 i`@:`(m,t). If there is no motivation, then one uses i`@::`t.

```
(@:)  infix 3 :: !UserId !(LabeledTask a) → Task a | iData a
(@::) infix 3 :: !UserId !(Task a)        → Task a | iData a
```

Exercises

9. *orTasks versus andTasks*
Create a workflow 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 `button` tasks. Study the different behavior of `orTasks` and `andTasks`.

10. *Number guessing*
Create a 2-person workflow 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 workflow should give feedback to person 2 whether the number guessed is too low, too high, or just right. In the latter case, the workflow returns `Just`N. Person 2 can also give up, in which case the workflow should return `Nothing`.
Optional: Person 1 is given the result of person 2, and has a chance to respond with a 'personal' message.

11. *Tic-tac-toe*
Create a 2-person workflow 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 workflow has to ensure that players enter marks only when it is their turn to do so.

2.9 Speculative tasks and multiple users: deadlines

Workflow 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 workflow 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 workflow 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. 2.7. 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

12. *Delayed task*
Create a workflow 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.

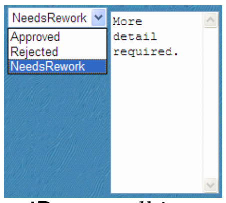
13. *Number guessing with deadline*
Use the delegation example of Sect. 2.9 in such a way that the number guessing game of exercise 10 can be created with it.

14. *Tic-tac-toe with deadline*
Use the delegation example of Sect. 2.9 in such a way that the tic-tac-toe game of exercise 11 can be created with it.

2.10 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
```



`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.

2.11 Spawning tasks and controlling them

A novel feature of the iTask toolkit is the ability to *spawn* and *delete* arbitrarily complex new tasks. Existing tasks can use a number of functions to check or wait for completion of such a spawned task. Tasks can get suspended and activated again, and tasks can suspend or delete themselves. These functions can be found in the module `iTasksProcessHandling.dcl`. We show the main definitions here:

```
definition module iTasksProcessHandling

:: Wid a
:: WorkflowStatus  = WflActive    UserId
                   | WflSuspended UserId
                   | WflFinished
                   | WflDeleted

spawnWorkflow      :: !UserId !Bool !(LabeledTask a) → Task (Wid   a) | iData a
waitForWorkflow    :: !(Wid a)                       → Task (Maybe a) | iData a
getWorkflowStatus  :: !(Wid a)                       → Task WorkflowStatus
activateWorkflow   :: !(Wid a)                       → Task Bool
suspendWorkflow    :: !(Wid a)                       → Task Bool
deleteWorkflow     :: !(Wid a)                       → Task Bool
changeWorkflowUser :: !UserId !(Wid a)               → Task Bool

suspendMe          :: (Task Void)
deleteMe           :: (Task Void)
```

A spawned task *t* of type (`Task a`) is identified and manipulated by means of an identification value of type (`Wid a`). Now (`spawnWorkflow` *uid active* (*label*,*t*)) creates a new task *t* that runs in parallel to the currently existing tasks. This new task *t* is handled by user *uid*, and if *active* holds, it will be an active task the user can engage in immediately. If *active* does not hold, then the task is initially suspended. `spawnWorkflow` returns the handle *ht* to the spawned task.

It should be noted that the behavior described above is very similar to the use of `@:` and `@::` combinators that have described in Sect. 2.8. However, because we now have a handle to such a spawned task, we can create more complicated, and more realistic, workflow cases. Consider for instance the need of the program chair in the case study to assign reviewing tasks to program members. Only after every review task has been finished, the program chair can proceed to collect the information and make a decision on the papers. This can be expressed as a single `andTasks` expression, sequentially followed by the task to make the decision. Unfortunately, real life is usually less structured: for a subset of papers it becomes rapidly clear that they should be accepted, and another subset gets rejected; some papers require additional reviewing; and some reviewers may fail to deliver before the deadline. Hence, it makes more sense to structure this workflow as a set of spawned tasks. In the case study, this is done in the function `assignReviewersForm`.

The functions mentioned above are fairly self-explanatory. One interesting function is `changeWorkflowUser`, which, when given a user identification, transfers the indicated task to the given user. This is of course a useful construct that occurs many times in the real world: workers may get ill, change jobs, have holidays, but the work remains to be done. For these cases new resources need to found and work has to be reallocated.

Exercises

15. *Number guessing in a group*
In this exercise you extend the number guessing game of exercises 10 and 13 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.

2.12 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 workflow 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. 2.5. 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 [17] 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
```

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] }
```

[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.

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. 2.5, 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. 2.5), 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 workflows. As we have stated in Sect. 2.7, 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 workflow:

```
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 workflow 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 workflow 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 workflow 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. 2.8, 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 2.9). 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 workflow, 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
newTask t = doTask (λtst=:{tasknr,hst}
 ♯ (taskval,hst) = mkStoreForm (Init,storeId) id hst
 ♯ (done,v)      = taskval.value
 | done          = (v,{tst & hst = hst})
 ♯ (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. **??**). 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 workflow patterns that can be found in contemporary workflow tools (see [21]). We show that the iTask toolkit does support this workflow 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 HaskellCGI library [16]; the Curry approach [12]; writing XML applications [9] in *SMLserver* [8]. One sophisticated system is WASH/CGI by [20], 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 [19]. 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” [18]. 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 [?], GdH [?], 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 workflow patterns of over more than 30 contemporary commercial workflow systems [21]. 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. **??** 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 workflow 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 workflow tools use a graphical formalism to specify workflow 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 workflow 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 workflows 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 workflows, embedded in Clean. The specification is used to generate a multi-user interactive web-based workflow 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 workflow systems, who design workflows, 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 workflow 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 workflow system on a sophisticated high level of abstraction.

Future work will be the investigation of more "unusual" useful workflow 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 workflow systems and as a source of many examples, and Wil van der Aalst for commenting on the difference between the combinator approach and contemporary workflow specification languages.

References

1. A. Alimarine. *Generic Functional Programming - Conceptual Design, Implementation and Applications.* PhD thesis, University of Nijmegen, The Netherlands, 2005. ISBN 3-540-67658-9.
2. D. Atkins, T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a Domain Specific Language for Form-based Services. In *Usenix Conference on Domain Specific Languages*, Oct. 1997.
3. C. Brabrand, A. Møller, M. Ricky, and M. Schwartzbach. Powerforms: Declarative client-side form field validation. *World Wide Web Journal*, 3(4):205–314, 2000.

4. C. Brabrand, A. Møller, and M. Schwartzbach. The <bigwig> Project. In *ACM Transactions on Internet Technology (TOIT)*, 2002.
5. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects (FMCO'06)*, volume 4709, CWI, Amsterdam, The Netherlands, 7 - 10 November 2006. Springer-Verlag.
6. S. Ducasse, A. Lienhard, and L. Renggli. Seaside - A Multiple Control Flow Web Application Framework. In S. Ducasse, editor, *Proceedings ESUG 2004 International Conference – Research Track*, volume Technical Report IAM-04-008, pages 231–254. Institut für Informatik und Angewandte Mathematik, University of Bern, Switzerland, November 7 2004.
7. C. Elliot. Tangible Functional Programming. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 59–70, Freiburg, Germany, Oct 1–3 2007. ACM.
8. M. Elsman and N. Hallenberg. Web programming with SMLserver. In *Fifth International Symposium on Practical Aspects of Declarative Languages (PADL'03)*. Springer-Verlag, January 2003.
9. M. Elsman and K. F. Larsen. Typing XHTML Web applications in ML. In *International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, volume 3057 of *LNCS*, pages 224–238. Springer-Verlag, June 2004.
10. P. Graunke, S. Krishnamurthi, R. Bruce Findler, and M. Felleisen. Automatically Restructuring Programs for the Web. In M. Feather and M. Goedicke, editors, *Proceedings 16th IEEE International Conference on Automated Software Engineering (ASE'01)*. IEEE CS Press, Sept. 2001.
11. K. Hanna. A Document-Centered Environment for Haskell. In A. Butterfield, C. Grelck, and F. Huch, editors, *Proceedings Implementation and Application of Functional Languages, 17th InternationalWorkshop, IFL 2005 – Revised Selected Papers*, volume 4015, pages 196–211, Dublin, Ireland, September 19-21 2005. Springer.
12. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.
13. R. Hinze. A new approach to generic functional programming. In *The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. Boston, Massachusetts, January 2000.
14. J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37:67–111, May 2000.
15. J. Jansen, P. Koopman, and R. Plasmeijer. Efficient Interpretation by Transforming Data Types and Patterns to Functions. In H. Nilsson, editor, *Proceedings Seventh Symposium on Trends in Functional Programming, TFP 2006*, pages 157–172, Nottingham, UK, The University of Nottingham, April 19-21 2006.
16. E. Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, 2000.
17. R. Plasmeijer and P. Achten. The Implementation of iData - A Case Study in Generic Programming. In A. Butterfield, editor, *Proceedings Implementation and Application of Functional Languages - Revised Selected Papers, 17th International Workshop, IFL05*, LNCS 4015, pages 106–123, Department of Computer Science, Trinity College, University of Dublin, September 19-21 2006.
18. C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *Proceedings Fifth International Conference on Functional Programming (ICFP'00)*, Sept. 2000.

19. M. Serrano, E. Gallesio, and F. Loitsch. 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)*, pages 975 – 985, Portland, Oregon, USA, October 22-26 2006.
20. P. Thiemann. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. In S. Krishnamurthi and C. Ramakrishnan, editors, *Practical Aspects of Declarative Languages: 4th International Symposium, PADL 2002*, volume 2257 of *LNCS*, pages 192–208, Portland, OR, USA, January 19-20 2002. Springer-Verlag.
21. W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. QUT Technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane, 2002.

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
```

Spider Spinning for Dummies

Richard S. Bird

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
bird@comlab.ox.ac.uk

Oh what a tangled web we weave
when first we practise to derive.
(With apologies to Sir Walter Scott)

Abstract. Spider spinning is a snappy name for the problem of listing the ideals of a totally acyclic poset in such a way that each ideal is computed from its predecessor in constant time. Such an algorithm is said to be *loopless*. Our aim in these lectures is to show how to *calculate* a loopless algorithm for spider spinning. The calculation makes use of the fundamental laws of functional programming and the real purpose of the exercise is to show these laws in action.

1 Introduction

Consider the problem of generating all bit strings $a_1a_2 \ldots a_n$ of fixed length n satisfying given constraints of the form $a_i \leq a_j$ for various i and j. The generation is to be in *Gray path order*, meaning that exactly one bit changes from one bit string to the next. The *transition code* is a list of integers naming the bit that is to be changed at each step.

For example, with $n = 3$, consider the constraints $a_1 \leq a_2$ and $a_3 \leq a_2$. One possible Gray path is `000, 010, 011, 111, 110` with transition code $[2, 3, 1, 3]$ and starting string 000.

The snag is that the problem does not always have a solution. For example, with $n = 4$ and the constraints $a_1 \leq a_2 \leq a_4$ and $a_1 \leq a_3 \leq a_4$, the six possible bit strings, namely

```
0000, 0001, 0011, 0101, 0111, 1111
```

cannot be permuted into a Gray path.

Exercise 1. Why not?

Constraints of the form $a_i \leq a_j$ on bit strings of length n can be represented by a digraph with n nodes in which a directed edge $i \leftarrow j$ is associated with a constraint $a_i \leq a_j$. Knuth and Ruskey [6] proved that a Gray path exists whenever the digraph is *totally acyclic*, meaning that the undirected graph obtained by dropping the directions on the edges is acyclic. They called a connected totally acyclic digraph a *spider* because when an edge $i \leftarrow j$ is drawn with i below j the digraph can be made to look like an arachnid (see Figure 1 for a three-legged spider). They called a totally acyclic digraph a *tad*, but since its connected components are spiders, we will continue the arachnid metaphor and call it a *nest* of spiders.

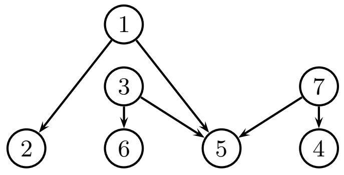


Fig. 1. A three-legged spider

Knuth called the problem of generating the associated bit strings in Gray path order, *spider squishing*. The more formal rendering of the task is: "generating all ideals [1] of a totally acyclic poset". Since spiders are good for the environment and should never be squished, we will call it *spider spinning* instead.

But there is a twist: we want the generation to be *loopless*, meaning that the first transition should be produced in *linear* time in the size of the nest, and each subsequent transition in *constant* time. Note that the idea of a loopless algorithm is defined in terms of the transitions between bit strings, not the bit strings themselves. Writing out a bit string is not possible in constant time.

Knuth and Ruskey gave an algorithm for spider spinning but it wasn't loopless. There is a program, SPIDERS, on Knuth's web site [4] that does perform loopless spider spinning. It is quite complicated, as Knuth readily admits:

> "But I apologize at the outset that the algorithm seems to be rather subtle, and I have not been able to think of any way to explain it to dummies".

Our aim in these lectures is to *calculate* a loopless algorithm for spider spinning. I have no idea if my algorithm bears any relationship to Knuth's algorithm, since I don't understand either. While it is generally true in mathematics that calculations simplify complicated things, in programming it is usually the other way around: simple programs are transformed into programs that can be completely opaque. So it is with loopless spider spinning.

[1] By an *ideal* of a poset S is meant a subset I of S such that if $x \in I$ and $x \leq y$, then $y \in I$.

2 Loopless algorithms

The term *loopless* was first introduced by Ehrlich in [1]. Imagine a program to list all combinatorial patterns of some kind, such as the subsequences or permutations of a list. Suppose each pattern is obtained from its predecessor by a single *transition*. For subsequences a transition i could mean "insert or delete the element at position i". For permutations a transition i could mean "swap the item in position i with the one in position $i-1$". An algorithm for generating all patterns is called *loopless* if the first transition is produced in linear time and each subsequent transition in constant time.

Loopless algorithms were formulated in a procedural setting and many clever tricks, such as the use of focus pointers, doubly-linked lists, and coroutines, have been used to construct them. See for example [5], which contains references to much of the literature on looplessness. Bear in mind though that loopless algorithms are not necessarily faster than their non-loopless counterparts. To quote again from [4]:

> "The extra contortions that we need to go through in order to achieve looplessness are usually ill-advised, because they actually cause the total execution time to be longer than it would be with a more straightforward algorithm. But hey, looplessness carries an academic cachet. So we might as well treat this task as a challenging exercise that might help us to sharpen our algorithmic wits."

Change the penultimate word to 'calculational' and you will appreciate the real point of the lectures.

2.1 Unfoldr

Being functional rather than procedural programmers, we will formulate the idea of a loopless algorithm in terms of the standard function `unfoldr`. Recall the Haskell standard type `Maybe`:

```
> data Maybe a = Nothing | Just a
```

The function `unfoldr` is defined by

```
> unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
> unfoldr step b
>         = case step b of
>             Nothing     -> []
>             Just (a,b') -> a : unfoldr step b'
```

By definition, a loopless algorithm is one that is expressed in the form

```
       unfoldr step . prolog
```

where `step` takes constant time and `prolog x` takes $O(n)$ steps, where n is the *size* of `x`. For instance, if `x` is a list, then n is the length of `x`, and if `x` is a tree, then n is the number of nodes in the tree. Every loopless algorithm has to be of this form.

2.2 Warm-up 1

By way of warming-up for the spider spinning to come, we first take a look at some loopless versions of functions that return lists. The obvious place to start is the identity function on lists. We have

```
> id :: [a] -> [a]
> id = unfoldr uncons

> uncons :: [a] -> Maybe (a,[a])
> uncons []     = Nothing
> uncons (x:xs) = Just (x,xs)
```

That was easy, so now let us consider the function `reverse` that reverses a list. In Haskell this function is defined by

```
> reverse :: [a] -> [a]
> reverse = foldl (flip (:)) []
```

The useful combinator `flip`, which we will meet again later on, is defined by

```
> flip f x y = f y x
```

A loopless program for reversing a list is now given by

```
> reverse = unfoldr uncons . foldl (flip (:)) []
```

Of course, all the real work is done in the prolog.

2.3 Warm-up 2

For another warm-up, consider the function `concat` that concatenates a list of lists. Here is a loopless version:

```
> concat :: [[a]] -> [a]
> concat = unfoldr step . filter (not . null)

> step :: [[a]] -> Maybe (a,[[a]])
> step []           = Nothing
> step ((x:xs):xss) = Just (x,consList xs xss)

> consList :: [a] -> [[a]] -> [[a]]
> consList xs xss = if null xs then xss else xs : xss
```

The prolog filters out nonempty lists from the input and takes linear time in the length of the list. The function `step` maintains the invariant that it takes and returns a list of *nonempty* lists.

Exercise 2. Why is it necessary to exclude empty lists?

Exercise 3. Would the alternative definition

```
> concat = unfoldr uncons . foldr (++) []
```

also serve as a loopless program?

The answer to the first question is that empty lists have to be filtered out of the input, otherwise `step` would not take constant time. For example, consider an input of the form $[[1], [\,], [\,], \ldots, [\,], [2]]$ in which there are n empty sequences between the first and last singleton lists. After producing the first element 1, it takes n steps to produce the second element 2 of the final list.

The answer to the second question is no, the prolog does not take time proportional to the length of its argument; it takes time proportional to the length of its result.

2.4 Warm-up 3

For the next warm-up consider the preorder traversal of a binary tree:

```
> data Tree a = Nil | Bin a (Tree a) (Tree a)

> preorder :: Tree a -> [a]
> preorder Nil         = []
> preorder (Bin x l r) = [x] ++ preorder l ++ preorder r
```

We can make `preorder` loopless in two stages. First we have

```
> preorder t = preForest [t]
```

where `preForest` returns the preorder traversal of a forest:

```
> preForest :: [Tree a] -> [a]
> preForest []            = []
> preForest (Nil:ts)      = preForest ts
> preForest (Bin x l r:ts) = [x] ++ preForest (l:r:ts)
```

Now we make `preForest` loopless:

```
> preorder = unfoldr step . wrapTree

> step :: [Tree a] -> Maybe (a,[Tree a])
> step []             = Nothing
> step (Bin x l r:ts) = Just (x,consTree l (consTree r ts))

> consTree t ts = if nil t then ts else t:ts

> wrapTree t = consTree t []
```

Note the invariant on `step` , namely that it takes and returns a forest of *non-empty* binary trees.

As a generalistaion, consider the preorder traversal of a forest of rose trees:

```
> type Forest a = [Rose a]
> data Rose a   = Node a (Forest a)

> preorder :: Forest a -> [a]
> preorder []              = []
> preorder (Node x ts:us) = x:preorder (ts ++ us)
```

We have `preorder = unfoldr step`, where

```
> step :: Forest a -> Maybe (a,Forest a)
> step []             = Nothing
> step (Node x ts:us) = Just (x,ts ++ us)
```

Ah, but `step` is not constant time because `++` isn't.

We can make `step` take constant time with a change of type. Instead of taking a forest as argument, we can make `step` take a list of *nonempty* forests, revising its definition to read

```
> step :: [Forest a] -> Maybe (a,[Forest a])
> step [] = Nothing
> step ((Node x ts:us):vss)
>         = Just (x,consList ts (consList us vss))
```

This is essentially the same trick as we performed for `concat`. Now we have

```
> preorder    = unfoldr step . wrapList
> wrapList ts = consList ts []
```

2.5 Warm-up 4

For the final warm-up let us go back to binary trees and this time consider *inorder* traversal:

```
> inorder :: Tree a -> [a]
> inorder Nil         = []
> inorder (Bin x l r) = inorder l ++ [x] ++ inorder r
```

How do we make `inorder` loopless?

Well, one answer is to convert the binary tree into a forest of rose trees in such a way that the inorder traversal of the former is the preorder traversal of the latter. For example, consider Figure 2. The conversion is performed by a function `convert` :

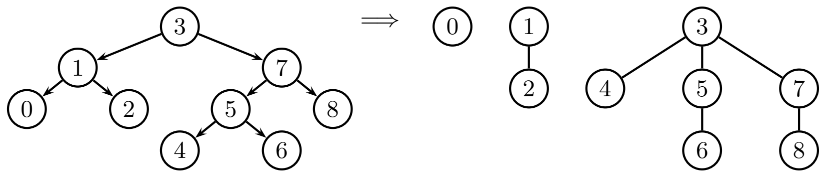


Fig. 2. Converting a binary tree to a forest of rose trees

```
> convert :: Tree a -> [Rose a]
> convert t = addTree t []

> addTree :: Tree a -> [Rose a] -> [Rose a]
> addTree Nil rs      = rs
> addTree (Bin x l r) = addTree l (Node x (convert r): rs)
```

Converting a binary tree into a forest of rose trees takes linear time in the size of the tree. Now we have

```
> inorder = unfoldr step . wrapList . convert
```

where `step` is as defined for preorder traversal.

Exercise 4. Construct a loopless program for the *postorder* traversal of a forest of rose trees.

---

3 Spider spinning with legless spiders

Here is a picture of some poor legless spiders:

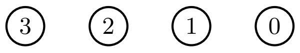


Fig. 3. A nest of four legless spiders

There are no constraints on the nodes in a nest of legless spiders, so we are seeking a method for listing *all* bit strings of given length in such a way that each string differs from its predecessor in just one bit. One possible transition code for the four spiders of Figure 3 is

```
[0,1,0,2,0,1,0,3,0,1,0,2,0,1,0]
```

We seek a loopless algorithm whose prolog is linear in n, the number of legless spiders in the nest. In other words we want a *Gray code* [2]. For example, one possible listing of all bits strings $a_3\,a_2\,a_1\,a_0$ of length four is as follows:

```
0000   0110  1100  1010
0001   0111  1101  1011
0011   0101  1111  1001
0010   0100  1110  1000
```

This particular ordering is called *The Gray binary code*. The least significant bit is on the right and varies the most often. But there are many other Gray codes.

Changing the meaning of transition i to read "insert/delete element a_i", we can also read the transition code as instructions to generate all subsequences of a list of length 4. For example, applying the transitions above to `abcd` yields

```
a, ab, b, bc, abc, ac, c, cd, acd, abcd, bcd, bd, abd, ad, d
```

To generate all strings $a_n\,a_{n-1}\dots a_0$ we can start off with all bits 0 and first generate all bit strings $0a_{n-1}\dots a_0$. Then a_n is changed to 1, and again all strings of length n are generated but in reverse order by running through the transitions backwards. (This is the advantage of having the least significant bit on the right.)

The description translates easily into a recursive definition of a function `gray` for generating transitions:

```
> gray :: Int -> [Int]
> gray 0     = []
> gray (n+1) = gray n ++ [n] ++ reverse (gray n)
```

In order to make `gray` loopless we would like to get rid of `reverse`. This function takes linear time before it delivers its first element.

Exercise 5. How can `reverse` be eliminated?

The answer to the question is: just remove it! The function `gray n` returns a *palindrome*, so the second clause can be replaced by

```
> gray (n+1) = gray n ++ [n] ++ gray n
```

Exercise 6. Prove formally by induction that `gray` returns a palindrome.

Next, one way to make `gray` loopless is to observe that the recursive case looks very similar to an instance of the inorder traversal of a certain binary tree:

[2] According to Knuth, named after Frank Gray, not Elisha Gray

```
> grayTree :: Int -> Tree Int
> grayTree 0     = Nil
> grayTree (n+1) = Bin t n t  where t = grayTree n
```

We have `gray = inorder . grayTree` . We know from the warm-up section how to make `inorder` loopless, so we have

```
> gray = unfoldr step . wrapList . convert . grayTree
```

Exercise 7. What is wrong with this 'loopless' definition of `gray` ?

The answer is that the prolog takes exponential time! The problem is not with `grayTree` , which takes only n steps to build the tree, but with `convert` , which takes linear time in the size of the tree. But the tree has exponential size once the sharing of subtrees is destroyed.

What we need to do is construct `convert . grayTree` directly. We define

```
> grayForest :: Int -> Forest Int
> grayForest 0     = []
> grayForest (n+1) = ts ++ [Node n ts]
>                    where ts = grayforest n
```

Then we have

```
> gray = unfoldr step . wrapList . grayforest
```

where `step` is as defined in the second warm-up.

Exercise 8. There is still a snag. What is it?

The snag is that `grayforest n` takes quadratic time because adding an element to the end of a list is a linear-time operation. Much better than exponential time, but not good enough.

There are various ways to jump this hurdle. The simplest is to introduce *queues*, redefining a forest to be a *queue* of trees rather than a list. Okasaki's implementation of queues [7] provides a type `Queue a` for which the following operations all take constant time:

```
       insert  :: Queue a -> a -> Queue a
       remove  :: Queue a -> (a, Queue a)
       empty   :: Queue a
       isempty :: Queue a -> Bool
```

To install queues, we redeclare the type `Forest` to read

```
> type Forest a = Queue (Tree a)
```

The function `grayForest` is redefined to read

```
> grayForest 0     = empty
> grayForest (n+1) = insert tq (Node n tq)
>                    where tq = grayForest n
```

Now we have

```
> gray =  unfoldr step . wrapQueue . grayForest
```

where `step` and `wrapQueue` are defined as follows:

```
> wrapQueue :: Queue a -> [Queue a]
> wrapQueue xq = consQueue xq []

> consQueue :: Queue a -> [Queue a] -> [Queue a]
> consQueue xq xqs = if isempty xq then xqs else xq:xqs

> step :: [Forest a] -> Maybe (a,[Forest a])
> step []       = Nothing
> step (zq:zqs) = Just (x,consQueue xq (consQueue yq zqs))
>                 where (Fork x xq,yq) = remove zq
```

This is a loopless algorithm for `gray`.

3.1 Another loopless program

Here is another loopless algorithm for `gray` , one that uses a cyclic structure rather than a queue. Consider the forest that arises with `gray 4` and pictured in Figure 4.

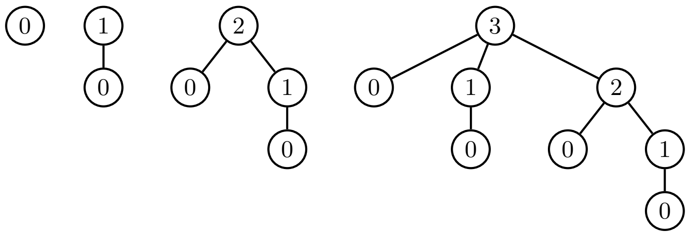


Fig. 4. A forest for gray 4.

In general a Gray forest has the property that a node labelled `k` has exactly `k` children; moreover these children are given by the list `take k ts` , where `ts` is the Gray forest. Suppose we define `grayCycle` by

```
> grayCycle :: Int -> Forest Int
> grayCycle n = ts  where ts = [Node k ts | k <- [0..n-1]]
```

The result returned by `grayCycle` is a cyclic structure. Moreover, building this structure takes linear time.

We can process this cycle of trees by removing the first tree, outputting its label `k` and adding `k` subtrees of the cycle to the cycle. We can represent the information about how many trees to take by a pair `(k,ts)`. Hence

```
> gray = unfoldr step . wrapPair . grayCycle

> wrapPair ts = [(length ts,ts)]

> step :: [(Int,Forest Int)] -> Maybe (Int,[(Int,Forest Int)])
> step [] = Nothing
> step ((j,Node k ts:us):vss) =
>             Just (k,consPair (k,ts) (consPair (j-1,us) vss))

> consPair (k,ts) kts = if k==0 then kts else (k,ts):kts
```

The function `step` maintains the invariant that it takes and returns lists of pairs whose first components are nonzero. This is another loopless program.

4 Spider spinning with tree spiders

Next, consider spider spinning when each spider is just a tree, such as pictured in Figure 5

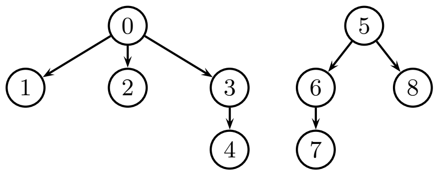


Fig. 5. A nest of two tree spiders

One possible transition code for this nest is

```
586878  0  878685  3  586878  4  878685  2  586878  4
878685  3  586878  1  878685  3  586878  4  878685  2
586878  4  878685  3  586878
```

This special case of spider spinning was considered by Koda and Ruskey [3]. A useful way to think of the problem is in terms of colourings. Think of the nodes as being coloured black if the associated bit is 1, and coloured white if the bit is 0. Thus every descendent of a white node has to be white. The black nodes of a single spider identifies a *prefix* of the underlying tree, so the black nodes in a nest identify a prefix of the forest. The transition code for a nest of tree spiders is a list of node labels, showing which node should change colour at each step. In the literature, the prefixes of a forest are also known as the "principal subforests" and the "ideals of a forest poset". Ingenious loopless algorithms for this problem are described in [3–5]. A non-loopless algorithm based on continuations appeared in [2].

4.1 Boustrophedon product

As the example above showed, the transitions for a nest of spiders can be formed by weaving together the transitions for each individual spider in the nest. It is easy to see how the weaving should proceed. Suppose the nest contains just two spiders, x and y. Begin, say, by going through the transitions for y. Follow this with the first transition for x and then repeat the transitions for y but in reverse order. We saw essentially the same idea in the Gray code problem. Continue in this fashion, weaving single transitions of x with complete transitions of y, alternating forwards and backwards, rather like the shuttle on a loom or an ox ploughing a field. Indeed, Knuth uses the name *boustrophedon product* for essentially this operation. We will call it `box` because it's short, pronounceable, and contains an 'ox'.[3] Here is the definition of `box`:

```
> box :: [a] -> [a] -> [a]
> box [] bs     = bs
> box (a:as) bs = bs ++ [a] ++ box as (reverse bs)
```

For example, `box [3,4,5,6] [0,1,2]` returns

```
[0,1,2,3,2,1,0,4,0,1,2,5,2,1,0,6,0,1,2]
```

Exercise 9. Prove that `box` is associative, a property that can be expressed as the identity

```
box (box as bs) = box as . box bs
```

Just as `concat` concatenates a list of lists, so the function `boxall` applies `box` to a list of lists:

```
> boxall :: [[a]] -> [a]
> boxall = foldr box []
```

Since `box` is associative with unit `[]` we could just as well have defined

```
> boxall = foldl box []
```

---

[3] There is another reason for the name *box*, but we'll get to that much later on.

Exercise 10. Prove this assertion.

For a list of length n of lists each of length m, the output of `boxall` has length $(m+1)^n - 1$, which is exponential in mn, the total length of the input.

We can now define `ncode` and `scode`, the transition codes for a nest of spiders and a single spider, respectively. First we introduce the type declarations

```
> type Nest a   = [Spider a]
> data Spider a = Node a (Nest a)
```

Now we can define

```
> ncode :: Nest a -> [a]
> ncode = boxall . map scode

> scode :: Spider a -> [a]
> scode (Node a xs) = a : ncode xs
```

The transition code for a single spider consists of an initial transition to change the colour of the root node (in fact, from white to black), followed by a complete list of the transitions for the nest of its subspiders. The definition of `ncode` is short and sweet, but not loopless.

4.2 Map fusion

So far there has been very little, if anything, in the way of program calculation. Now we start. The best advice when faced with calculating a program is to be guided by the form of the definitions under manipulation. In other words, begin by simply taking out the toolbox of functional laws and seeing which tools fit the problem in hand. The other indispensable piece of advice is to begin with a simpler problem, and we did that by starting out with legless spider spinning. Guided by the simpler example we know that at some point we are going to have to eliminate occurrences of `reverse` and then represent lists by queues of rose trees. But that is in the future.

The first step, dictated solely by the form of the definition of `ncode`, is an application of the *map-fusion law* for `foldr`. This law states that

```
       foldr f e . map g = foldr (f . g) e
```

In words, a fold after a map is just a fold. Applying it to `ncode` gives the alternative definition

```
> ncode = foldr (box . scode) []
```

The form of this definition suggests that we now concentrate on the function `box . scode` .

We calculate:

```
      box (scode (Node a xs)) bs
   =   {definition of scode}
      box (a:ncode xs) bs
   =   {definition of box}
      bs ++ a:box (ncode xs) (reverse bs)
   =   {definition of ncode}
      bs ++ b:box (foldr (box . scode) [] xs) (reverse bs)
```

This easy calculation consists of no more that instantiating the definitions of `scode` , `box` and `ncode` . But it brings into focus a new expression namely

```
      box . foldr (box . scode) []
```

The form of this expression suggests the use of another fundamental law of functional programming, the *fold-fusion law* of `foldr` . This law states that

```
      f . foldr g a = foldr h b
```

provided `f` is strict; `f a = b` ; and

```
      f (g x y) = h x (f y)
```

for all `x` and `y`. Fold-fusion is one of the fundamental laws of functional programming and crops up in nearly every calculation. Indeed the map-fusion law is a special case of fold-fusion.

The function `box` is strict and `box [] = id`, so it remains to find an `h` such that

```
      box (box (scode x) bs) = h x (box bs)
```

Of course, there is no guarantee that `h` exists.

Exercise 11. Calculate `h` .

The essential fact that guarantees success is that `box` is associative, so

```
      box (box (scode x) bs) = box (scode x) . box bs
```

This immediately gives us a definition of `h` :

```
      h x f = box (scode x) . f
```

Thus

```
      box . foldr (box . scode) [] = foldr h id
```

But this identity can be simplified. Observe for example that

```
  foldr h id [x,y,z] = box (scode x) . box (scode y) . box (scode z)
```

which suggests the truth of the identity

```
       foldr h id xs bs = foldr (box . scode) bs xs
```

More briefly,

```
       foldr h id  = flip (foldr (box . scode))
```

Exercise 12. Suppose `f :: a -> a -> a` is strict, associative, and has unit `e`, so `f e = id`. Prove that

```
       f . foldr (f . g) e = flip (foldr (f . g))
```

This result will be useful in general spider spinning, so we will call it the *monoid law* of `foldr`.

Hence, setting `bop = box . scode`[4], we have calculated that

```
> ncode = foldr bop []
> bop (Node a xs) bs = bs ++ [a] ++ foldr bop (reverse bs) xs
```

In this version of `ncode` the function `boxall` no longer appears. Neither does `scode`.

4.3 Eliminating reverse

Having eliminated `boxall`, we now eliminate `reverse`. However, unlike the case of legless spider spinning, this elimination requires a little more work.

The idea is that, instead of reversing a sequence explicitly, we construct both `ncode` and `reverse . ncode` at the same time. To do so we make use of another basic law of functional programming, the *tupling* law of `foldr`. This law states that

```
  fork (foldr f a,foldr g b) = foldr (cross . fork (f,g)) (a,b)
```

where the combinators `fork` and `cross` are defined by

```
> fork (f,g) x      = (f x,g x)
> cross (f,g) (x,y) = (f x,g y)
```

To apply tupling we need a function, `opb` say, so that

```
       reverse . foldr bop bs = foldr opb (reverse bs)
```

Then we obtain

```
       fork (ncode,reverse . ncode) = foldr op ([],[])
```

[4] We could have introduced this abbreviation earlier, but didn't because of another piece of useful advice about calculation: don't rush into naming things until the moment is ripe, because calculation involving the named thing has both to unpack the name and repackage it again.

where

```
       op x (bs,sb) = (bop x bs,opb x sb)
```

Appealing to fold fusion again, we have to satisfy the fusion condition

```
       reverse (bop x bs) = opb x (reverse bs)
```

We calculate:

```
       reverse (bop (Node a xs) bs)
    =   {definition of bop}
       reverse (bs ++ [a] ++ foldr bop (reverse bs) xs)
    =   {property of reverse}
       reverse (foldr bop (reverse bs) xs) ++ [a] ++ reverse bs
    =   {putative definition of opb}
       foldr opb bs xs ++ [a] ++ reverse bs
```

Hence we can define

```
> opb (Node a xs) sb = foldr opb (reverse sb) xs ++ [a] ++ sb
```

That gives

```
       op (Node a xs) (bs,sb)
    =   {definition of op}
       (bop (Node a xs) bs, opb (Node a xs) sb)
    =   {definitions of bop and opb}
       (bs ++ [a] ++ foldr bop sb xs,
        foldr opb bs xs ++ [a] ++ sb)
    =   {definition of op}
       (bs ++ [a] ++ cs, sc ++ [a] ++ sb)
        where (cs,sc) = foldr op (sb,bs) xs
```

In summary, we have arrived at

```
> ncode = fst . foldr op ([],[])
> op (Node a xs) (bs,sb) = (bs ++ [a] ++ cs, sc ++ [a] ++ sb)
>                          where (cs,sc) = foldr op (sb,bs) xs
```

Occurrences of `reverse` have been eliminated.

So far, so good. But computation of `foldr op ([],[])` takes quadratic time because `++` takes linear time. As with legless spider spinning this problem can be solved by representing lists as the preorder traversals of forests of trees:

```
> type Forest a = Queue (Tree a)
> data Tree a   = Fork a (Tree a)
```

We now have

```
> ncode  = unfoldr step . prolog
> prolog = wrapQueue . fst . foldr op (empty,empty)
> op (Node a xs) (bq,qb)
>        = (insert bq (Fork a cq), insert qc (Fork a qb))
>           where (cq,qc) = foldr op (qb,bq) xs
```

The definitions of `step` and `wrapQueue` are exactly the same as in legless spider spinning. This is a loopless algorithm for `ncode`.

5 Spider spinning with general spiders

It is now time to tackle the general spider spinning problem. First, observe that by picking a spider up by one of its nodes we get a tree with directed edges, such as that shown in Figure 5. Different trees arise depending on which node is picked up, but they all represent the same constraints.

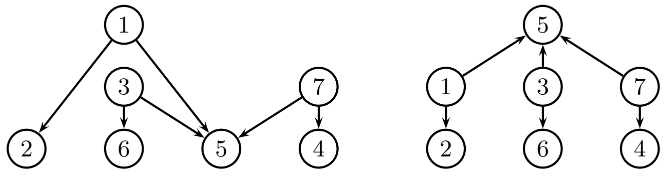


Fig. 6. A spider and an associated tree

Thus we can model general spiders with the type declarations

```
> type Nest a   = [Spider a]
> data Spider a = Node a [(Dir,Spider a)]
> data Dir      = Dn | Up
```

There is one complication when dealing with general spiders that does not arise with simpler species: the starting bit string is not necessarily a string consisting of all 0s. For example, with $n = 3$ and the constraints $a_1 \geq a_2 \leq a_3$, the five possible bit strings, namely `000 001 100 101 111`, can only be arranged in Gray path order by starting with one of the odd-weight strings: `001`, `100`, or `111`. However we are going to ignore consideration of the definition of the function `seed :: Nest Int -> [Bit]` for determining the starting string.

As with tree spiders we can define

```
> ncode :: Nest a -> [a]
> ncode = boxall . map scode
```

We define `scode` to be the concatenation of two lists, a white code and a black code. The white code for a spider `x` is a valid transition sequence for `x` when its head is coloured white (corresponding to a 0 bit), and the black code a valid sequence when its head is coloured black (corresponding to a 1 bit):

```
> scode :: Spider a -> [a]
> scode (Node a ls) = wcode ls ++ [a] ++ bcode ls
```

Note that when the spiders are tree spiders, `wcode` should return the empty sequence.

For `scode` to be correct, the final spider colouring generated by executing `wcode ls` has to be the initial colouring on which `bcode ls` starts. In order for the colourings to match up we need to define `wcode` in terms of a variant of `box` which we will call `cox`.[5]

```
cox [] bs        = bs
cox (as++[a]) bs = cox as (reverse bs) ++ [a] ++ bs
```

For example,

```
box [2,3,4] [0,1] = 01 2 10 3 01 4 10
cox [2,3,4] [0,1] = 10 2 01 3 10 4 01
```

Whereas `box as bs` begins with `bs` and ends with either `bs` or `reverse bs` depending on whether `as` has even length, `cox as bs` ends with `bs` and begins with either `bs` or `reverse bs`.

Exercise 13. Give a valid Haskell definition of `cox`.

Exercise 14. Prove that `cox` is associative.

Now, setting `coxall = foldr cox []`, we define

```
> wcode, bcode :: [(Dir,Spider a)] -> [a]
> wcode = coxall . map wc
> bcode = boxall . map bc
```

where `wc, bc :: (Dir,Spider a) -> [a]`. Use of `coxall` in the definition of `wcode` means that the final colouring after executing `wcode` will be the union of the final colourings generated by the `wc` transitions, and use of `boxall` in the definition of `bcode` means that this colouring will also be the union of the colourings on which the `bc` transitions start.

It remains to define `wc` and `bc`. Given the choices above, the following definitions are forced:

```
> wc (Up,Node a ls) = wcode ls ++ [a] ++ bcode ls
> wc (Dn,Node a ls) = reverse (wcode ls)
> bc (Up,Node a ls) = reverse (bcode ls)
> bc (Dn,Node a ls) = wcode ls ++ [a] ++ bcode ls
```

[5] By the way, 'to box and cox' means 'to take turns', which is certainly what both operations do and is the real reason for their names. The term comes from the comic play 'Box and Cox - A Romance of Real Life in One Act', by John Maddison Morton. Box and Cox were two lodgers who shared their rooms - one occupying them by day and the other by night.

Look first at `wc (Up,x)`. When the head of the mother spider of `x` is white and is connected to `x` by an upwards edge, there are no constraints on `wc (Up,x)`, so we can define it to be either `scode x` or its reverse. But the subsequent transitions are those in the list `bc (Up,x)` and the only way to match up the final colouring of the former with the initial colouring of the latter is with the definitions above. The reasoning is dual with `bc (Dn,x)` and `wc (Dn,x)`.

Finally, we show that `ncode` can be expressed in terms of the transitions for a single spider. Suppose we define a mother spider

```
> mother :: Nest a -> Spider a
> mother xs = Node undefined [(Up,x) | x <- xs]
```

Setting `legs (Node a ls) = ls` we calculate

```
   (wcode . legs . mother) xs
 =  wcode [(Up,x) | x <- xs]
 =  coxall [wc (Up,x) | x <- xs]
 = (coxall . map scode) xs
 =  ncode xs
```

Hence `ncode = wcode . legs . mother`. Again, the program is short and sweet but not loopless.

6 Transformation

The transformation to loopless form follows the same path as the simpler problem of a nest of tree spiders. Specifically, we are going to:

1. Eliminate `boxall` and `coxall` from the definition of `ncode` by appeal to map fusion and fold fusion.
2. Eliminate `reverse` by appeal to tupling.
3. Eliminate the remaining complexity by introducing queues.

It is the appeal to fold fusion in the first step that is the trickiest.

6.1 First steps

As an easy first step we apply map fusion to the definitions of `wcode` and `bcode`, obtaining

```
> bcode = foldr (box . bc) []
> wcode = foldr (cox . wc) []
```

We now concentrate on the term `box . bc`. Everything we discover will apply to the second term `cox . wc` with the obvious changes. Instantiating the first clause of the definition of `bc`, we find

```
 box (bc (Up,Node a ls)) = box (reverse (foldr (box . bc) [] ls))
```

The right-hand side suggests an appeal to fold fusion: find an `h` so that

```
box . reverse . foldr (box . bc) [] = foldr h id
```

The problem, however, is that no such `h` exists. The monoid law of `foldr` (see above) does give an `h` for which

```
box . foldr (box . bc) [] = foldr h id
```

But this doesn't help. As we will show later on, we can also find an `h` such that

```
cox . reverse . foldr (box . bc) [] = foldr h id
```

But this only helps if we can change a `box` into a `cox`.

Fortunately, we can. In fact there are two properties that we will need. Firstly, we have the *morphing* property

```
box as bs = if even (length as)
            then cox as bs
            else cox as (reverse bs)
```

As a functional identity:

```
box as = cox as . revif (odd (length as))
```

where

```
> revif p = if p then reverse else id
```

Secondly, we also have the *conjugate* property

```
reverse (box as bs) = cox (reverse as) (reverse bs)
```

Exercise 15. Prove the morphing and conjugate properties.

Using the morphing property we obtain

```
box (bc (Up,Node a ls)) = cox (reverse (foldr (box . bc) [] ls)) .
                          revif (not (bp ls))
```

where

```
> bp = even . length . foldr (box . bc) []
```

Occurrences of `cox` in `cox . wc` can be similarly replaced by `box`.

Returning to the fusion condition, we calculate

```
   cox (reverse (box (bc l) bs))
 =   {conjugate property}
   cox (cox (reverse (bc l)) (reverse bs))
 =   {associativity of cox}
   cox (reverse (bc l)) . cox (reverse bs)
 =   {conjugate property}
   reverse . box (bc l) . reverse . cox (reverse bs)
```

Hence

```
   h l f = reverse . box (bc l) . reverse . f
```

Appeal to fold fusion now yields

```
   cox . reverse . foldr (box . bc) [] = foldr h id
```

Exercise 16. With the above definition of `h` , prove that

```
   foldr h id ls xs = reverse (foldr (box . bc) (reverse xs) ls
```

Using the result of the exercise above, we obtain

```
   cox (reverse (foldr (box . bc) [] ls)) =
     reverse . flip (foldr (box . bc)) ls . reverse
```

Putting all the above together we have

```
   box (bc (Up,Node a ls))
 =   {definition of bc}
   box (reverse (foldr (box . bc) [] ls))
 =   {morphing box into cox}
   cox (reverse (foldr (box . bc) [] ls)) .
   revif (not (bp ls))
 =   {fold fusion}
   reverse . flip (foldr (box . bc)) ls . reverse .
   revif (not (bp ls))
 =  {definition of revif}
   reverse . flip (foldr (box . bc)) ls .
   revif (bp ls)
```

In summary, and introducing `bop = box . bc` and `wop = cox . wc` , we have shown that

```
> bop (Up,Node a ls) = reverse . ff bop ls . revif (bp ls)
> wop (Dn,Node a ls) = reverse . ff wop ls . revif (wp ls)
```

where

```
> ff = flip . foldr
> bp = even . length . foldr bop []
> wp = even . length . foldr wop []
```

Two clauses remain.

6.2 The remaining clauses

Here is one of the remaining two clauses:

```
   box (bc (Dn,Node a ls)) = box (wcode ls ++ [a] ++ bcode ls)
```

Clearly, we now need a fact about `box (ws ++ [a] ++ bs)`. Here it is:

```
   box (ws ++ [a] ++ bs) xs =
     if even (length ws)
     then  box ws xs ++ [a] ++ box bs (reverse xs)
     else  box ws xs ++ [a] ++ box bs xs
```

Equivalently, defining

```
> add a (ws,bs) = ws ++ [a] ++ bs
```

we can express this *distributive law* as a functional identity

```
box (add a (ws,bs)) = add a . fork (box ws, box bs . revif w)
                      where w = even (length ws)
```

Dually,

```
cox (add a (ws,bs)) = add a . fork (cox ws . revif b, cox bs)
                      where b = even (length bs)
```

Now, setting `ws = wcode ls` and `bs = bcode ls`, we calculate:

```
   box (add a (ws,bs))
 =  {distributive law}
   add a . fork (box ws, box bs . revif w)
 =  {since box (bcode ls) = box (foldr (box . bc) [] ls)
     and   box (foldr (box . bc) [] ls) = ff bop ls}
   add a . fork (box ws, ff bop ls . revif w)
=   {morphing: box ws = cox ws . revif (not w)
          and  cox (wcode ls) = ff wop ls}
   add a . fork (ff wop ls . revif (not w),
                 ff bop ls . revif w)
```

In summary we have calculated that

```
> bop (Up,Node a ls) = reverse . ff bop ls . revif (bp ls)
> wop (Dn,Node a ls) = reverse . ff wop ls . revif (wp ls)

> bop (Dn,Node a ls) =
>           add a . fork (ff wop ls . revif (not (wp ls)),
>                         ff bop ls . revif (wp ls))
> wop (Up,Node a ls) =
>           add a . fork (ff wop ls . revif (bp ls),
>                         ff bop ls . revif (not (bp ls)))

> add a (ws,bs) = ws ++ [a] ++ bs
```

6.3 The parity functions

Calculationally speaking, everything so far is tickety-boo. We have eliminated occurrences of `box` and `cox`, which is what we set out to do. However, constant recomputation of the parity functions

```
> bp  = even . length . foldr bop []
> wp  = even . length . foldr wop []
```

is inefficient both in time and space. Instead of repeatedly computing these values we will install them in a *parity spider*, an element of

```
> data PSpider a = PNode (Bool,Bool) a [(Dir,PSpider a)]
```

Parity and other information is retrieved from a parity spider by

```
> wp (PNode (w,b) a ls)   = w
> bp (PNode (w,b) a ls)   = b
> head (PNode (w,b) a ls) = a
> legs (PNode (w,b) a ls) = ls
```

Parity information is installed in a spider by

```
> decorate :: Spider a -> PSpider a
> decorate (Node a ls) = pnode a [(d,decorate x) | (d,x) <- ls]
```

The smart constructor `pnode` is defined by

```
> pnode a ls = PNode (foldr op (True,True) ls) a ls
```

where

```
> op :: (Dir,PSpider a) -> (Bool,Bool) -> (Bool,Bool)
> op (Up,x) (w,b) = ((wp x /= bp x) && w, bp x && b)
> op (Dn,x) (w,b) = (wp x && w, (wp x /= bp x) && b)
```

Installing parity information takes linear time in the size of a spider.

Exercise 17. Justify the above definition of `op`.

The result is the following completely opaque program for `ncode`:

```
> ncode = foldr wop [] . legs . decorate . mother

> wop (Dn,x) = reverse . ff wop (legs x) . revif (wp x)
> wop (Up,x) = add (head x) .
>              fork (ff wop (legs x) . revif (bp x),
>                    ff bop (legs x) . revif (not (bp x)))
> bop (Up,x) = reverse . ff bop (legs x) . revif (bp x)
> bop (Dn,x) = add (head x) .
>              fork (ff wop (legs x) . revif (not (wp x)),
>                    ff bop (legs x) . revif (wp x))
```

6.4 Eliminating reverse

The next step is to eliminate `reverse` by an appeal to the tupling law for `foldr` . Instead of going into details we will just sketch the reasoning. In effect, every sequence `as` is represented by a pair of sequences `(as,sa)` where `sa = reverse as` . Reversal is then implemented by swapping the two lists.

There are only three changes to the program above. First, the definition of `add` is changed to read

```
> add :: a -> (([a],[a]),([a],[a])) -> ([a],[a])
> add a ((ws,sw),(bs,sb)) = (ws ++ [a] ++ bs, sb ++ [a] ++ sw)
```

Second, all occurrences of `reverse` are changed to `swap` :

```
> swap (x,y) = (y,x)
```

Third, all occurrences of `revif` are changed to `swapif` :

```
> swapif p   = if p then swap else id
```

As a result of these changes, we obtain

```
> ncode = fst . foldr wop ([],[]) . legs . decorate . mother

> wop, bop :: (Dir,PSpider a) -> ([a],[a]) -> ([a],[a])
> wop (Dn,x) = swap . ff wop (legs x) . swapif (wp x)
> wop (Up,x) = add (head x) .
>              fork (ff wop (legs x) . swapif (bp x),
>                    ff bop (legs x) . swapif (not (bp x)))
> bop (Up,x) = swap . ff bop (legs x) . swapif (bp x)
> bop (Dn,x) = add (head x) .
>              fork (ff wop (legs x) . swapif (not (wp x)),
>                    ff bop (legs x) . swapif (wp x))
```

Ignoring the cost of `add` the computation of `ncode` takes linear time in the size of the nest.

6.5 Queues again

Now we are ready for looplessness. As in the simpler problem of tree spiders, we represent a list by the preorder traversal of a forest of Rose trees, where a forest is a *queue* of Rose trees:

```
> type Forest a = Queue (Rose a)
> data Rose a   = Fork a (Forest a)
```

We change `add` once more, this time to read

```
> add :: a -> Pair (Pair (Forest a)) -> Pair (Forest a)
> add a ((wf,fw),(bf,fb)) =
>        (insert wf (Fork a bf), insert fb (Fork a fw))
```

We replace the previous definition of `ncode` by

```
> ncode = preorder . fst . foldr bop (empty,empty) .
>         legs . decorate . mother
```

Now all the work is done by `preorder`, which we can implement just as we did for tree spiders:

```
> preorder :: Forest a -> [a]
> preorder = unfoldr step . wrapQueue
```

The definition of `step` is the same as it was for the tree spider problem. Summarising, we have

```
> ncode = unfoldr step . prolog
```

where

```
> prolog   = wrapQueue . fst . foldr wop (empty,empty) .
>            legs . decorate . mother
```

Even though the prolog is a six-act play, involving characters such as spiders, lists, queues and trees, and strange actions like flipping, swapping and folding, it nevertheless takes linear time in the size of the nest, so this finally is a loopless program for spider spinning.

References

1. Ehrlich, G. (1973) Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. ACM*, 20: pages 500–513.
2. Filliatre, J-C, and Pottier, F. (2003) Producing all ideals of a forest, functionally. *Journal of Functional Programming*, 13, 5: pages 945–956.
3. Koda, Y. and Ruskey, R. (1993) A Gray code for the ideals of a forest poset. *Journal of Algorithms* 15: pages 324–340.
4. Knuth, D.E. (2001) SPIDERS: a program downloadable from `www-cs-faculty.stanford.edu/~knuth/programs.html`.
5. Knuth, D.E. (2005) *The Art of Computer Programming, Vol 4, Fascicles 2,3,4*. Addison-Wesley.
6. Knuth, D.E. and Ruskey, F. (2003) Efficient Coroutine Generation of Constrained Gray Sequences (aka Deconstructing Coroutines) *Object-Orientation to Formal Methods: Dedicated to The Memory of Ole-Johan Dahl.* LNCS 2635, Springer-Verlag.
7. Okasaki, C. (1995) Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5, 4: pages 583–592.

Dependently Typed Programming in Agda

Draft

Ulf Norell

Chalmers University, Gothenburg

1 Introduction

In Hindley-Milner style languages, such as Haskell and ML, there is a clear separation between types and values. In a dependently typed language the line is more blurry – types can contain (*depend on*) arbitrary values and appear as arguments and results of ordinary functions.

The standard example of a dependent type is the type of lists of a given length: `Vec A n`. Here `A` is the type of the elements and `n` is the length of the list. Many languages allow you to define lists (or arrays) of a given size, but what makes `Vec` a true dependent type is that the length of the list can be an arbitrary term, which need not be known at compile time.

Since dependent types allows types to talk about values, we can encode properties of values as types whose elements are proofs that the property is true. This means that a dependently typed programming language can be used as a logic. In order for this logic to be consistent we need to require programs to be total, i.e. they are not allowed to crash or non-terminate.

The rest of these notes are structured as follows: Section 2 introduces the dependently typed language Agda and its basic features, and Section 3 explains a couple of programming techniques made possible by the introduction of dependent types.

2 Agda Basics

Agda is a dependently typed language based on Martin-Löf type theory developed at Chalmers University in Gothenburg. This section introduces the basic features of Agda and how they can be employed in the construction of dependently typed programs. Information on how to obtain the Agda system and further details on the topics discussed here can be found on the Agda wiki[1].

[1] `http://www.cs.chalmers.se/~ulfn/Agda`

This section is a literate Agda file which can be compiled by the Agda system. Hence, we need to start at the beginning: Every Agda file contains a single top-level module whose name corresponds to the name of the file. In this case the file is called `AgdaBasics.lagda`[2].

```
module AgdaBasics where
```

The rest of your program goes inside the top-level module. Let us start by defining some simple datatypes and functions.

2.1 Datatypes and pattern matching

Similar to languages like Haskell and ML, a key concept in Agda is pattern matching over algebraic datatypes. With the introduction of dependent types pattern matching becomes even more powerful as we shall see in Section 2.4 and Section 3. But for now, let us start with simply typed functions and datatypes.

Datatypes are introduced by a `data` declaration, giving the name and type of the datatype as well as the constructors and their types. For instance, here is the type of booleans

```
data Bool : Set where
  true  : Bool
  false : Bool
```

The type of `Bool` is `Set`, the type of small[3] types. Functions over `Bool` can be defined by pattern matching in a for Haskell programmers familiar way:

```
not : Bool -> Bool
not true  = false
not false = true
```

Agda functions are not allowed to crash, so a function definition must cover all possible cases. This will be checked by the type checker and an error is raised if there are missing cases.

In Haskell and ML the type of `not` can be inferred from the defining clauses and so in these languages the type signature is not required. However, in the presence of dependent types this is no longer the case and

[2] Literate Agda files have the extension `lagda` and ordinary Agda files have the extension `agda`.

[3] There is hierarchy of increasingly large types. The type of `Set` is `Set1`, whose type is `Set2`, and so on.

we are forced to write down the type signature of `not`. This is not a bad thing, since by writing down the type signature we allow the type checker, not only to tell us when we make mistakes, but also to guide us in the construction of the program. When types grow more and more precise the dialog between the programmer and the type checker gets more and more interesting.

Another useful datatype is the type of (unary) natural numbers.

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat
```

Addition on natural numbers can be defined as a recursive function.

```
_+_ : Nat -> Nat -> Nat
zero  + m = m
suc n + m = suc (n + m)
```

In the same way as functions are not allowed to crash, they must also be terminating. To guarantee termination recursive calls have to be made on structurally smaller arguments. In this case `_+_` passes the termination checker since the first argument is getting smaller in the recursive call (`n` $<$ `suc n`). Let us define multiplication while we are at it

```
_*_ : Nat -> Nat -> Nat
zero  * m = zero
suc n * m = m + n * m
```

Agda supports a flexible mechanism for mixfix operators. If a name of a function contains underscores (`_`) it can be used as an operator with the arguments going where the underscores are. Consequently, the function `_+_` can be used as an infix operator writing `n + m` for `_+_ n m`. There are (almost) no restrictions on what symbols are allowed as operator names, for instance we can define

```
_or_ : Bool -> Bool -> Bool
true  or x = x
false or _ = false

if_then_else_ : {A : Set} -> Bool -> A -> A -> A
if true  then x else y = x
if false then x else y = y
```

In the second clause of the `_or_` function the underscore is a wildcard pattern, indicating that we don't care what the second argument is and we can't be bothered giving it a name. This, of course, means that we cannot refer to it on the right hand side. The precedence and fixity of an operator can be declared with an `infix` declaration:

```
infixl 60 _*_
infixl 40 _+_
infixr 20 _or_
infix  5 if_then_else_
```

There are some new and interesting bits in the type of `if_then_else_`. For now, it is sufficient to think about `{A : Set} ->` as declaring a polymorphic function over a type `A`. More on this in Sections 2.2 and 2.3.

Just as in Haskell and ML datatypes can be parameterised by other types. The type of lists of elements of an arbitrary type is defined by

```
infixr 40 _::_
data List (A : Set) : Set where
  []   : List A
  _::_ : A -> List A -> List A
```

Again, note that Agda is quite liberal about what is a valid name. Both `[]` and `_::_` are accepted as sensible names. In fact, Agda names can contain arbitrary non-whitespace unicode characters, with a few exceptions, such as parenthesis and curly braces. So, if we really wanted (which we don't) we could define the list type as

```
data _★ (α : Set) : Set where
  ε : α ★
  _◁_ : α -> α ★ -> α ★
```

This liberal policy of names means that being generous with whitespace becomes important. For instance, `not:Bool->Bool` would not be a valid type signature for the `not` function, since it is in fact a valid name.

2.2 Dependent functions

Let us now turn our attention to dependent types. The most basic dependent type is the dependent function type, where the result type depends on the value of the argument. In Agda we write `(x : A) -> B` for the type of functions taking an argument `x` of type `A` and returning a result of type `B`, where `x` may appear in `B`. A special case is when `x` itself is a type. For instance, we can define

```
identity : (A : Set) -> A -> A
identity A x = x

zero' : Nat
zero' = identity Nat zero
```

This is a dependent function taking a type argument `A` and an element of `A` and returns the element. This is how polymorphic functions are encoded in Agda. Here is an example of a more intricate dependent function; the function which takes a dependent function and applies it to an argument:

```
apply : (A : Set)(B : A -> Set) ->
        ((x : A) -> B x) -> (a : A) -> B a
apply A B f a = f a
```

Agda accepts some short hands for dependent function types:

– `(x : A)(y : B) -> C` for `(x : A) -> (y : B) -> C` , and
– `(x y : A) -> B` for `(x : A)(y : A) -> B` .

The elements of dependent function types are lambda terms which may carry explicit type information. Some alternative ways to define the identity function above are:

```
identity₂ : (A : Set) -> A -> A
identity₂ = \A x -> x

identity₃ : (A : Set) -> A -> A
identity₃ = \(A : Set)(x : A) -> x

identity₄ : (A : Set) -> A -> A
identity₄ = \(A : Set) x -> x
```

2.3 Implicit arguments

We saw in the previous section how dependent functions taking types as arguments could be used to model polymorphic types. The thing with polymorphic functions, however, is that you don't have to say at which type you want to apply it – that is inferred by the type checker. However, in the example of the identity function we had to explicitly provide the type argument when applying the function. In Agda this problem is solved by a general mechanism for *implicit arguments*. To declare a function argument implicit we use curly braces instead of parenthesis in the type: `{x : A} -> B` means the same thing as `(x : A) -> B` except that when you use a function of this type the type checker will try to figure out the argument for you.

Using this syntax we can define a new version of the identity function, where you don't have to supply the type argument.

```
id : {A : Set} -> A -> A
id x = x

true' : Bool
true' = id true
```

Note that the type argument is implicit both when the function is applied and when it is defined.

There is no restrictions on what arguments can be made implicit, nor is there any guarantees that an implicit argument can be inferred by the type checker. For instance, we could be silly and make the second argument of the identity function implicit as well:

```
silly : {A : Set}{x : A} -> A
silly {_}{x} = x

false' : Bool
false' = silly {x = false}
```

Clearly, there is no way the type checker could figure out what the second argument to `silly` should be. To provide an implicit argument explicitly you use the implicit application syntax `f {v}`, which gives `v` as the left-most implicit argument to `f`, or as shown in the example above, `f {x = v}`, which gives `v` as the implicit argument called `x`. The name of an implicit argument is obtained from the type declaration.

Conversely, if you want the type checker to fill in a term which needs to be given explicitly you can replace it by an underscore. For instance,

```
one : Nat
one = identity _ (suc zero)
```

It is important to note that the type checker will not do any kind of search in order to fill in implicit arguments. It will only look at the typing constraints and perform unification[4].

Even so, a lot can be inferred automatically. For instance, we can define the fully dependent function composition. (Warning: the following type is not for the faint of heart!)

[4] Miller pattern unification to be precise.

```
_∘_ : {A : Set}{B : A -> Set}{C : (x : A) -> B x -> Set}
      (f : {x : A}(y : B x) -> C x y)(g : (x : A) -> B x)
      (x : A) -> C x (g x)
(f ∘ g) x = f (g x)

plus-two = suc ∘ suc
```

The type checker can figure out the type arguments `A`, `B`, and `C`, when we use `_∘_`.

We have seen how to define simply typed datatypes and functions, and how to use dependent types and implicit arguments to represent polymorphic functions. Let us conclude this part by defining some familiar functions.

```
map : {A B : Set} -> (A -> B) -> List A -> List B
map f []        = []
map f (x :: xs) = f x :: map f xs

_++_ : {A : Set} -> List A -> List A -> List A
[]        ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

2.4 Datatype families

So far, the only use we have seen of dependent types is to represent polymorphism, so let us look at some more interesting examples. The type of lists of a certain length, mentioned in the introduction, can be defined as follows:

```
data Vec (A : Set) : Nat -> Set where
  []   : Vec A zero
  _::_ : {n : Nat} -> A -> Vec A n -> Vec A (suc n)
```

This declaration introduces a number of interesting things. First, note that the type of `Vec A` is `Nat -> Set`. This means that `Vec A` is a family of types indexed by natural numbers. So, for each natural number `n`, `Vec A n` is a type. The constructors are free to construct elements in an arbitrary type of the family. In particular, `[]` constructs an element in `Vec A zero` and `_::_` an element in `Vec A (suc n)` for some `n`.

There is a distinction between *parameters* and *indices* of a datatype. We say that `Vec` is parameterised by a type `A` and indexed over natural numbers.

In the type of `_::_` we see an example of a dependent function type. The first argument to `_::_` is an implicit natural number `n` which is the length of the tail. We can safely make `n` implicit since the type checker can infer it from the type of the third argument.

Finally, note that we chose the same constructor names for `Vec` as for `List`. Constructor names are not required to be distinct between different datatypes.

Now, the interesting part comes when we start pattern matching on elements of datatype families. Suppose, for instance, that we want to take the head of a non-empty list. With the `Vec` type we can actually express the type of non-empty lists, so we define `head` as follows:

```
head : {A : Set}{n : Nat} -> Vec A (suc n) -> A
head (x :: xs) = x
```

This definition is accepted by the type checker as being exhaustive, despite the fact that we didn't give a case for `[]`. This is fortunate, since the `[]` case would not even be type correct – the only possible way to build an element of `Vec A (suc n)` is using the `_::_` constructor.

The rule for when you have to include a particular case is very simple: *if it is type correct you have to include it.*

Dot patterns Here is another function on `Vec`:

```
vmap : {A B : Set}{n : Nat} -> (A -> B) -> Vec A n -> Vec B n
vmap f []        = []
vmap f (x :: xs) = f x :: vmap f xs
```

Perhaps surprisingly, the definition map on `Vec` is exactly the same as on `List`, the only thing that changed is the type. However, something interesting is going on behind the scenes. For instance, what happens with the length argument when we pattern match on the list? To see this, let us define new versions of `Vec` and `vmap` with fewer implicit arguments:

```
data Vec₂ (A : Set) : Nat -> Set where
  nil  : Vec₂ A zero
  cons : (n : Nat) -> A -> Vec₂ A n -> Vec₂ A (suc n)

vmap₂ : {A B : Set}(n : Nat) -> (A -> B) -> Vec₂ A n -> Vec₂ B n
vmap₂ .zero    f nil           = nil
vmap₂ .(suc n) f (cons n x xs) = cons n (f x) (vmap₂ n f xs)
```

What happens when we pattern match on the list argument is that we learn things about its length: if the list turns out to be `nil` then the length argument must be `zero`, and if the list is `cons n x xs` then the only type correct value for the length argument is `suc n`. To indicate that the value of an argument has been deduced by type checking, rather than observed by pattern matching it is prefixed by a dot (`.`).

In this example we could choose to define `vmap` by first pattern matching on the length rather than on the list. In that case we would put the dot on the length argument of `cons`[5]:

```
vmap₃ : {A B : Set}(n : Nat) -> (A -> B) -> Vec₂ A n -> Vec₂ B n
vmap₃ zero    f nil           = nil
vmap₃ (suc n) f (cons .n x xs) = cons n (f x) (vmap₃ n f xs)
```

The rule for when an argument should be dotted is: *if there is a unique type correct value for the argument it should be dotted.*

In the example above, the terms under the dots were valid patterns, but in general they can be arbitrary terms. For instance, we can define the image of a function as follows:

```
data Image_∋_ {A B : Set}(f : A -> B) : B -> Set where
  im : (x : A) -> Image f ∋ f x
```

Here we state that the only way to construct an element in the image of `f` is to pick an argument `x` and apply `f` to `x`. Now if we know that a particular `y` is in the image of `f` we can compute the inverse of `f` on `y`:

```
inv : {A B : Set}(f : A -> B)(y : B) -> Image f ∋ y -> A
inv f .(f x) (im x) = x
```

Absurd patterns Let us define another datatype family, name the family of numbers smaller than a given natural number.

```
data Fin : Nat -> Set where
  fzero : {n : Nat} -> Fin (suc n)
  fsuc  : {n : Nat} -> Fin n -> Fin (suc n)
```

Here `fzero` is smaller than `suc n` for any `n` and if `i` is smaller than `n` then `fsuc i` is smaller than `suc n`. Note that there is no way of constructing a number smaller than zero. When there are no possible constructor patterns for a given argument you can pattern match on it with the absurd pattern `()`:

```
magic : {A : Set} -> Fin zero -> A
magic ()
```

[5] In fact the dot can be placed on any of the `n`s. What is important is that there is a unique binding site for each variable in the pattern.

Using an absurd pattern means that you do not have to give a right hand side, since there is no way anyone could provide an argument to your function. One might think that the clause would not have to be given at all, that the type checker would see that the matching is exhaustive without any clauses, but remember that a case can only be omitted if there is no type correct way of writing it. In the case of `magic` a perfectly type correct left hand side is `magic x`.

It is important to note that an absurd pattern can only be used if there are no valid constructor patterns for the argument, it is not enough that there are no closed inhabitants of the type[6]. For instance, if we define

```
data Empty : Set where
  empty : Fin zero -> Empty
```

Arguments of type `Empty` can not be matched with an absurd pattern, since there is a perfectly valid constructor pattern that would do: `empty x`. Hence, to define the `magic` function for `Empty` we have to write

```
magic' : {A : Set} -> Empty -> A
magic' (empty ())
-- magic' ()  -- not accepted
```

Now, let us define some more interesting functions. Given a list of length `n` and a number `i` smaller than `n` we can compute the `i`th element of the list (starting from 0):

```
_!_ : {n : Nat}{A : Set} -> Vec A n -> Fin n -> A
[]        ! ()
(x :: xs) ! fzero    = x
(x :: xs) ! (fsuc i) = xs ! i
```

The types ensure that there is no danger of indexing outside the list. This is reflected in the case of the empty list where there are no possible values for the index.

The `_!_` function turns a list into a function from indices to elements. We can also go the other way, constructing a list given a function from indices to elements:

```
tabulate : {n : Nat}{A : Set} -> (Fin n -> A) -> Vec A n
tabulate {zero}  f = []
tabulate {suc n} f = f fzero :: tabulate (f ∘ fsuc)
```

Note that `tabulate` is defined by recursion over the length of the result list, even though it is an implicit argument. There is in general no correspondance between implicit data and computationally irrelevant data.

[6] Since checking type inhabitation is undecidable.

2.5 Programs as proofs

As mentioned in the introduction, Agda's type system is sufficiently powerful to represent (almost) arbitrary propositions as types whose elements are proofs of the proposition. Here are two very simple propositions, the true proposition and the false proposition:

```
data   False : Set where
record True  : Set where

trivial : True
trivial = _
```

The false proposition is represented by the datatype with no constructors and the true proposition by the record type with no fields (see Section 2.8 for more information on records). The record type with no fields has a single element which is the empty record. We could have defined `True` as a datatype with a single element, but the nice thing with the record definition is that the type checker knows that there is a unique element of `True` and will fill in any implicit arguments of type `True` with this element. This is exploited in the definition of `trivial` where the right hand side is just underscore. If you nevertheless want to write the element of `True`, the syntax is `record{}`.

These two propositions are enough to work with decidable propositions. We can model decidable propositions as booleans and define

```
isTrue : Bool -> Set
isTrue true  = True
isTrue false = False
```

Now, `isTrue b` is the type of proofs that `b` equals `true`. Using this technique we can define the safe list lookup function in a different way, working on simply typed lists and numbers.

```
_<_ : Nat -> Nat -> Bool
_      < zero  = false
zero   < suc n = true
suc m  < suc n = m < n

length : {A : Set} -> List A -> Nat
length []        = zero
length (x :: xs) = suc (length xs)

lookup : {A : Set}(xs : List A)(n : Nat) ->
         isTrue (n < length xs) -> A
lookup []        n       ()
lookup (x :: xs) zero    p = x
lookup (x :: xs) (suc n) p = lookup xs n p
```

In this case, rather than there being no index into the empty list, there is no proof that a number `n` is smaller than `zero`. In this example using indexed types to capture the precondition is a little bit nicer, since we don't have to pass around an explicit proof object, but some properties cannot be easily captured by indexed types, in which case this is a nice alternative.

We can also use datatype families to define propositions. Here is a definition of the identity relation

```
data _==_ {A : Set}(x : A) : A -> Set where
  refl : x == x
```

For a type `A` and an element `x` of `A`, we define the the family of proofs of "being equal to `x`". This family is only inhabited at index `x` where the single proof is `refl`.

Another example is the less than or equals relation on natural numbers. This could be defined as a boolean function, as we have seen, but we can also define it inductively

```
data _≤_ : Nat -> Nat -> Set where
  leq-zero : {n : Nat} -> zero ≤ n
  leq-suc  : {m n : Nat} -> m ≤ n -> suc m ≤ suc n
```

One advantage of this approach is that we can pattern match on the proof object. This makes proving properties of `_≤_` easier. For instance,

```
leq-trans : {l m n : Nat} -> l ≤ m -> m ≤ n -> l ≤ n
leq-trans leq-zero     _           = leq-zero
leq-trans (leq-suc p) (leq-suc q) = leq-suc (leq-trans p q)
```

2.6 More on pattern matching

We have seen how to pattern match on the arguments of a function, but sometimes you want to pattern match on the result of some intermediate computation. In Haskell and ML this is done on the right hand side using a case or match expression. However, as we have learned, when pattern matching on an expression in a dependently typed language, you not only learn something about the shape of the expression, but you can also learn things about other expressions. For instance, pattern matching on an expression of type `Vec A n` will reveal information about `n`. This is not captured by the usual case expression, so instead of a case expression Agda provides a way of matching on intermediate computations on the left hand side.

The *with* construct The idea is that if you want to pattern match on an expression `e` in the definition of a function `f`, you abstract `f` over the value of `e`, effectively adding another argument to `f` which can then be matched on in the usual fashion. This abstraction is performed by the `with` construct. For instance,

```
min : Nat -> Nat -> Nat
min x y with x < y
min x y | true  = x
min x y | false = y
```

The equations for `min` following the with abstraction have an extra argument, separated from the original arguments by a vertical bar, corresponding to the value of the expression `x < y`. You can abstract over multiple expressions at the same time, separating them by vertical bars and you can nest with abstractions. In the left hand side, with abstracted arguments should be separated by vertical bars.

In this case pattern matching on `x < y` doesn't tell us anything interesting about the arguments of `min`, so repeating the left hand sides is a bit tedious. When this is the case you can replace the left hand side with `...`:

```
filter : {A : Set} -> (A -> Bool) -> List A -> List A
filter p [] = []
filter p (x :: xs) with p x
... | true  = x :: filter p xs
... | false = filter p xs
```

Here is an example when we do learn something interesting. Given two numbers we can compare them to see if they are equal. Rather than returning an uninteresting boolean, we can return a proof that the numbers are indeed equal when this is the case, and an explanation of why they are different when this is the case:

```
data _≠_ : Nat -> Nat -> Set where
  z≠s : {n : Nat} -> zero ≠ suc n
  s≠z : {n : Nat} -> suc n ≠ zero
  s≠s : {m n : Nat} -> m ≠ n -> suc m ≠ suc n

data Equal? (n m : Nat) : Set where
  eq  : n == m -> Equal? n m
  neq : n ≠ m -> Equal? n m
```

Two natural numbers are different if one is `zero` and the other `suc` of something, or if both are successors but their predecessors are different. Now we can define the function `equal?` to check if two numbers are equal:

```
equal? : (n m : Nat) -> Equal? n m
equal? zero    zero    = eq refl
equal? zero    (suc m) = neq z≠s
equal? (suc n) zero    = neq s≠z
equal? (suc n) (suc m) with equal? n m
equal? (suc n) (suc .n) | eq refl = eq refl
equal? (suc n) (suc m)  | neq p   = neq (s≠s p)
```

Note that in the case where both numbers are successors we learn something by pattern matching on the proof that the predecessors are equal. We will see more examples of this kind of informative datatypes in Section 3.1.

When you abstract over an expression using `with`, that expression is abstracted from the entire context. This means that if the expression occurs in the type of an argument to the function or in the result type, this occurrence will be replaced by the with-argument on the left hand side. For example, suppose we want to prove something about the `filter` function. That the only thing it does is throwing away some elements of its argument, say. We can define what it means for one list to be a sublist of another list:

```
infix 20 _⊆_
data _⊆_ {A : Set} : List A -> List A -> Set where
  stop : [] ⊆ []
  drop : forall {xs y ys} -> xs ⊆ ys ->      xs ⊆ y :: ys
  keep : forall {x xs ys} -> xs ⊆ ys -> x :: xs ⊆ x :: ys
```

The intuition is that to obtain a sublist of a given list, each element can either be dropped or kept. When the type checker can figure out the type of an argument in a function type you can use the `forall` syntax:

– `forall {x y} a b -> A` is short for `{x : _}{y : _}(a : _)(b : _) -> A`.

Using this definition we can prove that `filter` computes a sublist of its argument:

```
lem-filter : {A : Set}(p : A -> Bool)(xs : List A) -> filter p xs ⊆ xs
lem-filter p []        = stop
lem-filter p (x :: xs) with p x
... | true  = keep (lem-filter p xs)
... | false = drop (lem-filter p xs)
```

The interesting case is the `_::_` case. Let us walk through it slowly:

```
-- lem-filter p (x :: xs) = ?
```

At this point the goal that we have to prove is

```
-- (filter p (x :: xs) | p x) ⊆ x :: xs
```

In the goal `filter` has been applied to its with abstracted argument `p x` and will not reduce any further. Now, when we abstract over `p x` it will be abstracted from the goal type so we get

```
-- lem-filter p (x :: xs) with p x
-- ... | px = ?
```

where `p x` has been replaced by `px` in the goal type

```
-- (filter p (x :: xs) | px) ⊆ x :: xs
```

Now, when we pattern match on `px` the call to `filter` will reduce and we get

```
-- lem-filter p (x :: xs) with p x
-- ... | true  = ?  {- x :: filter p xs ⊆ x :: xs -}
-- ... | false = ?  {-      filter p xs ⊆ x :: xs -}
```

In some cases, it can be helpful to use `with` to abstract over an expression which you are not going to pattern match on. In particular, if you expect this expression to be instantiated by pattern matching on something else. Consider the proof that `n + zero == n`:

```
lem-plus-zero : (n : Nat) -> n + zero == n
lem-plus-zero zero = refl
lem-plus-zero (suc n) with n + zero | lem-plus-zero n
... | .n | refl = refl
```

In the step case we would like to pattern match on the induction hypothesis `n + zero == n` in order to prove `suc n + zero == suc n`, but since `n + zero` cannot be unified with `n` that is not allowed. However, if we abstract over `n + zero`, calling it `m`, we are left with the induction hypothesis `m == n` and the goal `suc m == suc n`. Now we can pattern match on the induction hypothesis, instantiating `m` to `n`.

2.7 Modules

The module system in Agda is primarily used to manage name spaces. In a dependently typed setting you could imagine having modules as first class objects that could be passed around and created on the fly, but in Agda this is not the case.

We have already seen that each file must define a single top-level module containing all the declarations in the file. These declarations can in turn be modules.

```
module Maybe where
  data Maybe (A : Set) : Set where
    nothing : Maybe A
    just    : A -> Maybe A

  maybe : {A B : Set} -> B -> (A -> B) -> Maybe A -> B
  maybe z f nothing  = z
  maybe z f (just x) = f x
```

By default all names declared in a module are visible from the outside. If you want to hide parts of a module you can declare it `private`:

```
module A where
  private
    internal : Nat
    internal = zero

  exported : Nat -> Nat
  exported n = n + internal
```

To access public names from another module you can qualify the name by the name of the module.

```
mapMaybe₁ : {A B : Set} -> (A -> B) -> Maybe.Maybe A -> Maybe.Maybe B
mapMaybe₁ f Maybe.nothing  = Maybe.nothing
mapMaybe₁ f (Maybe.just x) = Maybe.just (f x)
```

Modules can also be opened, locally or on top-level:

```
mapMaybe₂ : {A B : Set} -> (A -> B) -> Maybe.Maybe A -> Maybe.Maybe B
mapMaybe₂ f m = let open Maybe in maybe nothing (just ∘ f) m

open Maybe

mapMaybe₃ : {A B : Set} -> (A -> B) -> Maybe A -> Maybe B
mapMaybe₃ f m = maybe nothing (just ∘ f) m
```

When opening a module you can control which names are brought into scope with the `using`, `hiding`, and `renaming` keywords. For instance, to open the `Maybe` module without exposing the `maybe` function, and using different names for the type and the constructors we can say

```
open Maybe hiding (maybe)
           renaming (Maybe to _option; nothing to none; just to some)

mapOption : {A B : Set} -> (A -> B) -> A option -> B option
mapOption f none     = none
mapOption f (some x) = some (f x)
```

Renaming is just cosmetic, `Maybe A` and `A option` are interchangable.

```
mtrue : Maybe Bool
mtrue = mapOption not (just false)
```

Parameterised modules Modules can be parameterised by arbitrary types[7].

```
module Sort (A : Set)(_<_ : A -> A -> Bool) where
  insert : A -> List A -> List A
  insert y [] = y :: []
  insert y (x :: xs) with x < y
  ... | true  = x :: insert y xs
  ... | false = y :: x :: xs

  sort : List A -> List A
  sort []        = []
  sort (x :: xs) = insert x (sort xs)
```

When looking at the functions in parameterised module from the outside, they take the module parameters as arguments, so

```
sort₁ : (A : Set)(_<_ : A -> A -> Bool) -> List A -> List A
sort₁ = Sort.sort
```

You can apply the functions in a parameterised module to the module parameters all at once, by instantiating the module

```
module SortNat = Sort Nat _<_
```

This creates a new module `SortNat` with functions `insert` and `sort`.

```
sort₂ : List Nat -> List Nat
sort₂ = SortNat.sort
```

Often you want to instantiate a module and open the result, in which case you can simply write

```
open Sort Nat _<_ renaming (insert to insertNat; sort to sortNat)
```

without having to give a name to the instantiated module.

Sometimes you want to export the contents of another module from the current module. In this case you can open the module *publicly* using the `public` keyword:

```
module Lists (A : Set)(_<_ : A -> A -> Bool) where
  open Sort A _<_ public
  minimum : List A -> Maybe A
  minimum xs with sort xs
  ... | []       = nothing
  ... | y :: ys = just y
```

Now the `Lists` module will contain `insert` and `sort` as well as the `minimum` function.

[7] But not by other modules.

Importing modules from other files Agda programs can be split over multiple files. To use definitions from a module defined in another file the module has to be *imported*. Modules are imported by their names, so if you have a module `A.B.C` in a file `/some/local/path/A/B/C.agda` it is imported with the statement `import A.B.C`. In order for the system to find the file `/some/local/path` must be in Agda's search path.[8].

I have a file `Logic.agda` in the same directory as these notes, defining logical conjunction and disjunction. To import it we say

```
import Logic using (_∧_; _∨_)
```

Note that you can use the same namespace control keywords as when opening modules. Importing a module does not automatically open it (like when you say `import qualified` in Haskell). You can either open it separately with an open statement, or use the short form

```
open import Logic
```

Splitting a program over several files will improve type checking performance, since when you are making changes the type checker only has to type check the files that are influenced by the changes.

2.8 Records

We have seen a record type already, namely the record type with no fields which was used to model the true proposition. Now let us look at record types with fields. A record type is declared much like a datatype where the fields are indicated by the `field` keyword. For instance

```
record Point : Set where
  field x : Nat
        y : Nat
```

This declares a record type `Point` with two natural number fields `x` and `y`. To construct an element of `Point` you write

```
mkPoint : Nat -> Nat -> Point
mkPoint a b = record{ x = a; y = b }
```

To allow projection of the fields from a record, each record type comes with a module of the same name. This module is parameterised by an element of the record type and contains projection functions for the fields. In the point example we get a module

[8] The search path can be set from emacs by executing `M-x customize-group agda2`.

```
-- module Point (p : Point) where
--   x : Nat
--   y : Nat
```

This module can be used as it is or instantiated to a particular record.

```
getX : Point -> Nat
getX = Point.x

abs² : Point -> Nat
abs² p = let open Point p in x * x + y * y
```

At the moment you cannot pattern match on records, but this will hopefully be possible in a later version of Agda.

It is possible to add your own functions to the module of a record by including them in the record declaration after the fields.

```
record Monad (M : Set -> Set) : Set1 where
  field
    return : {A : Set} -> A -> M A
    _>>=_  : {A B : Set} -> M A -> (A -> M B) -> M B

  mapM : {A B : Set} -> (A -> M B) -> List A -> M (List B)
  mapM f [] = return []
  mapM f (x :: xs) = f x       >>= \y  ->
                     mapM f xs >>= \ys ->
                     return (y :: ys)

mapM' : {M : Set -> Set} -> Monad M ->
        {A B : Set} -> (A -> M B) -> List A -> M (List B)
mapM' Mon f xs = Monad.mapM Mon f xs
```

3 Programming Techniques

In this section we will describe and exemplify a couple of programming techniques which are made available in dependently typed languages: *views* and *universe constructions*.

3.1 Views

As we have seen pattern matching in Agda can reveal information not only about the term being matched but also about terms occurring in the type of this term. For instance, matching a proof of `x == y` against the `refl` constructor we (and the type checker) will learn that `x` and `y` are the same.

We can exploit this, and design datatypes whose sole purpose is to tell us something interesting about its indices. We call such a datatype a *view*. To use the view we define a view function, computing an element of the view for arbitrary indices.

This section on views is defined in the file `Views.lagda` so here is the top-level module declaration:

```
module Views where
```

Natural number parity Let us start with an example. We all know that any natural number n can be written on the form $2k$ or $2k+1$ for some k. Here is a view datatype expressing that. We use the natural numbers defined in the summer school library module `Data.Nat`.

```
open import Data.Nat

data Parity : Nat -> Set where
  even : (k : Nat) -> Parity (k * 2)
  odd  : (k : Nat) -> Parity (1 + k * 2)
```

An element of `Parity n` tells you if `n` is even or odd, i.e. if $n = 2k$ or $n = 2k + 1$, and in each case what k is. The reason for writing `k * 2` and `1 + k * 2` rather than `2 * k` and `2 * k + 1` has to do with the fact that `_+_` and `_*_` are defined by recursion over their first argument. This way around we get a better reduction behaviour.

Now, just defining the view datatype isn't very helpful. We also need to show that any natural number can be viewed in this way. In other words, given an arbitrary natural number `n` we need to compute an element of `Parity n`.

```
parity : (n : Nat) -> Parity n
parity zero = even zero
parity (suc n) with parity n
parity (suc .(k * 2))     | even k = odd k
parity (suc .(1 + k * 2)) | odd k  = even (suc k)
```

In the `suc n` case we use the view recursively to find out the parity of `n`. If `n = k * 2` then `suc n = 1 + k * 2` and if `n = 1 + k * 2` then `suc n = suc k * 2`.

In effect, this view gives us the ability to pattern match on a natural number with the patterns `k * 2` and `1 + k * 2`. Using this ability, defining the function that divides a natural number by two is more or less trivial:

```
half : Nat -> Nat
half n with parity n
half .(k * 2)     | even k = k
half .(1 + k * 2) | odd k  = k
```

Note that `k` is bound in the pattern for the view, not in the dotted pattern for the natural number.

Finding an element in a list Let us turn our attention to lists. First some imports: we will use the definitions of lists and booleans from the summer school library.

```
open import Data.Function
open import Data.List
open import Data.Bool
```

Now, given a predicate `P` and a list `xs` we can define what it means for `P` to hold for all elements of `xs`:

```
infixr 30 _:all:_
data All {A : Set}(P : A -> Set) : List A -> Set where
  all[]   : All P []
  _:all:_ : forall {x xs} -> P x -> All P xs -> All P (x :: xs)
```

A proof of `All P xs` is simply a list of proofs of `P x` for each element `x` of `xs`. Note that `P` does not have to be a decidable predicate. To turn a decidable predicate into a general predicate we define a function `Sat`.

```
Sat : {A : Set} -> (A -> Bool) -> A -> Set
Sat p x = isTrue (p x)
```

Using the `All` datatype we could prove the second part of the correctness of the `filter` function, namely that all the elements of the result of `filter` satisfies the predicate: `All (Sat p) (filter p xs)`. This is left as an exercise. Instead, let us define some interesting views on lists.

Given a decidable predicate on the elements of a list, we can either find an element in the list that satisfies the predicate, or else all elements satifies the negation of the predicate. Here is the corresponding view datatype:

```
data Find {A : Set}(p : A -> Bool) : List A -> Set where
  found : (xs : List A)(y : A) -> Sat p y -> (ys : List A) ->
          Find p (xs ++ y :: ys)
  not-found : forall {xs} -> All (Sat (not ∘ p)) xs -> Find p xs
```

We don't specify which element to use as a witness in the `found` case. If we wanted the view to always return the first (or last) matching element we could force the elements of `xs` (or `ys`) to satisfy the negation of `p`. To complete the view we need to define the view function computing an element of `Find p xs` for any `p` and `xs`. Here is a first attempt:

```
find₁ : {A : Set}(p : A -> Bool)(xs : List A) -> Find p xs
find₁ p [] = not-found all[]
find₁ p (x :: xs) with p x
... | true  = found [] x {! !} xs
... | false = {! !}
```

In the case where `p x` is `true` we want to return `found` (hence, returning the first match), but there is a problem. The type of the hole (`{! !}`) is `isTrue (p x)`, even though we already matched on `p x` and found out that it was `true`. The problem is that when we abstracted over `p x` we didn't know that we wanted to use the `found` constructor, so there were no `p x` to abstract over. Remember that `with` doesn't remember the connection between the with-term and the patterns. One solution to this problem is to make this connection explicit with a proof object. The idea is to not abstract over the term itself but rather over an arbitrary term of the same type and a proof that it is equal to the original term. Remember the type of equality proofs:

```
data _==_ {A : Set}(x : A) : A -> Set where
  refl : x == x
```

Now we define the type of elements of a type `A` together with proofs equal to some given `x` in `A`.

```
data Inspect {A : Set}(x : A) : Set where
  it : (y : A) -> x == y -> Inspect x
```

There is one obvious way to construct an element of `Inspect x`, namely to pick `x` as the thing which is equal to `x`.

```
inspect : {A : Set}(x : A) -> Inspect x
inspect x = it x refl
```

We can now define `find` by abstracting over `inspect (p x)` rather than `p x` itself. This will provide us with proofs `p x == true` or `p x == false` which we can use in the arguments to `found` and `not-found`. First we need a couple of lemmas about `isTrue` and `isFalse`:

```
trueIsTrue : {x : Bool} -> x == true -> isTrue x
trueIsTrue refl = _
```

```
falseIsFalse : {x : Bool} -> x == false -> isFalse x
falseIsFalse refl = _
```

Now we can define `find` without any problems.

```
find : {A : Set}(p : A -> Bool)(xs : List A) -> Find p xs
find p [] = not-found all[]
find p (x :: xs) with inspect (p x)
... | it true prf = found [] x (trueIsTrue prf) xs
... | it false prf with find p xs
find p (x :: ._) | it false _   | found xs y py ys = found (x :: xs) y py ys
find p (x :: xs) | it false prf | not-found npxs =
  not-found (falseIsFalse prf :all: npxs)
```

In the case where `p x` is true, `inspect (p x)` matches `it true prf` where `prf : p x == true`. Using our lemma we can turn this into the proof of `isTrue (p x)` that we need for the third argument of `found`. We get a similar situation when `p x` is false and `find p xs` returns `not-found`.

Indexing into a list In Sections 2.4 and Section 2.5 we saw two ways of safely indexing into a list. In both cases the type system guaranteed that the index didn't point outside the list. However, sometimes we have no control over the value of the index and it might well be that it is pointing outside the list. One solution in this case would be to wrap the result of the lookup function in a maybe type, but maybe types don't really tell you anything very interesting and we can do a lot better. First let us define the type of proofs that an element `x` is in a list `xs`.

```
data _∈_ {A : Set}(x : A) : List A -> Set where
  hd : forall {xs}   -> x ∈ x :: xs
  tl : forall {y xs} -> x ∈ xs -> x ∈ y :: xs
```

The first element of a list is a member of the list, and any element of the tail of a list is also an element of the entire list. Given a proof of `x ∈ xs` we can compute the index at which `x` occurs in `xs` simply by counting the number of `tl`s in the proof.

```
index : forall {A}{x : A}{xs} -> x ∈ xs -> Nat
index hd     = zero
index (tl p) = suc (index p)
```

Now, let us define a view on natural numbers `n` with respect to a list `xs`. Either `n` indexes some `x` in `xs` in which case it is of the form `index p` for some proof `p : x ∈ xs`, or `n` points outside the list, in which case it is of the form `length xs + m` for some `m`.

```
data Lookup {A : Set}(xs : List A) : Nat -> Set where
  inside  : (x : A)(p : x ∈ xs) -> Lookup xs (index p)
  outside : (m : Nat) -> Lookup xs (length xs + m)
```

In the case that `n` is a valid index we not only get the element at the corresponding position in `xs` but we are guaranteed that this is the element that is returned. There is no way a lookup function could cheat and always return the first element, say. In the case that `n` is indexing outside the list we also get some more information. We get a proof that `n` is out of bounds and we also get to know by how much.

Defining the lookup function is no more difficult than it would have been to define the lookup function returning a maybe.

```
_!_ : {A : Set}(xs : List A)(n : Nat) -> Lookup xs n
[] ! n = outside n
(x :: xs) ! zero  = inside x hd
(x :: xs) ! suc n with xs ! n
(x :: xs) ! suc .(index p)       | inside y p = inside y (tl p)
(x :: xs) ! suc .(length xs + n) | outside n  = outside n
```

A type checker for λ-calculus To conclude this section on views, let us look at a somewhat bigger example: a type checker for simply typed λ-calculus. This example was first implemented by Conor McBride in Epigram many years ago. His version not only guaranteed that when the type checker said ok things were really ok, but also provided a detailed explanation in the case where type checking failed. We will focus on the positive side here and leave the reporting of sensible and guaranteed precise error message as an exercise.

First, let us define the type language. We have one base type `ı` and a function type.

```
infixr 30 _⟶_
data Type : Set where
  ı   : Type
  _⟶_ : Type -> Type -> Type
```

When doing type checking we will inevitably have to compare types for equality, so let us define a view.

```
data Equal? : Type -> Type -> Set where
  yes : forall {τ} -> Equal? τ τ
  no  : forall {σ τ} -> Equal? σ τ

_=?=_ : (σ τ : Type) -> Equal? σ τ
ı         =?= ı       = yes
```

```
ı           =?= (_ → _) = no
(_ → _)    =?= ı        = no
(σ1 → τ1) =?= (σ2 → τ2) with σ1 =?= σ2 | τ1 =?= τ2
(σ → τ)   =?= (.σ → .τ) | yes | yes = yes
(σ1 → τ1) =?= (σ2 → τ2) | _   | _   = no
```

Note that we don't give any justification in the `no` case. The `_=?=_` could return `no` all the time without complaints from the type checker. In the `yes` case, however, we guarantee that the two types are identical.

Next up we define the type of raw lambda terms. We use unchecked deBruijn indices to represent variables.

```
infixl 80 _$_
data Raw : Set where
  var : Nat -> Raw
  _$_ : Raw -> Raw -> Raw
  lam : Type -> Raw -> Raw
```

We use Church style terms in order to simplify type inference. The idea with our type checker is that it should take a raw term and return a well-typed term, so we need to define the type of well-typed λ-terms with respect to a context Γ and a type τ.

```
Cxt = List Type

data Term (Γ : Cxt) : Type -> Set where
  var : forall {τ} -> τ ∈ Γ -> Term Γ τ
  _$_ : forall {σ τ} -> Term Γ (σ → τ) -> Term Γ σ -> Term Γ τ
  lam : forall σ {τ} -> Term (σ :: Γ) τ -> Term Γ (σ → τ)
```

We represent variables by proofs that a type is in the context. Remember that the proofs of list membership provided us with an index into the list where the element could be found. Given a well-typed term we can erase all the type information and get a raw term.

```
erase : forall {Γ τ} -> Term Γ τ -> Raw
erase (var x)   = var (index x)
erase (t $ u)   = erase t $ erase u
erase (lam σ t) = lam σ (erase t)
```

In the variable case we turn the proof into a natural number using the `index` function.

Now we are ready to define the view of a raw term as either being the erasure of a well-typed term or not. Again, we don't provide any justification for giving a negative result. Since, we are doing type inference the type is not a parameter of the view but computed by the view function.

```
data Infer (Γ : Cxt) : Raw -> Set where
  ok  : (τ : Type)(t : Term Γ τ) -> Infer Γ (erase t)
  bad : {e : Raw} -> Infer Γ e
```

The view function is the type inference function taking a raw term and computing an element of the `Infer` view.

```
infer : (Γ : Cxt)(e : Raw) -> Infer Γ e
```

Let us walk through the three cases (variable, application, and lambda).

```
infer Γ (var n)   with Γ ! n
infer Γ (var .(length Γ + n)) | outside n  = bad
infer Γ (var .(index x))      | inside σ x = ok σ (var x)
```

In the variable case we need to take case of the fact that the raw variable might be out of scope. We can use the lookup function `_!_` we defined above for that. When the variable is in scope the lookup function provides us with the type of the variable and the proof that it is in scope.

```
infer Γ (e1 $ e2)
  with infer Γ e1
infer Γ (e1 $ e2)           | bad     = bad
infer Γ (.(erase t1) $ e2) | ok ı t1 = bad
infer Γ (.(erase t1) $ e2) | ok (σ → τ) t1
  with infer Γ e2
infer Γ (.(erase t1) $ e2) | ok (σ → τ) t1 | bad = bad
infer Γ (.(erase t1) $ .(erase t2)) | ok (σ → τ) t1 | ok σ' t2
  with σ =?= σ'
infer Γ (.(erase t1) $ .(erase t2))
  | ok (σ → τ) t1 | ok .σ t2 | yes = ok τ (t1 $ t2)
infer Γ (.(erase t1) $ .(erase t2))
  | ok (σ → τ) t1 | ok σ' t2 | no = bad
```

The application case is the bulkiest simply because there are a lot of things we need to check: that the two terms are type correct, that the first term has a function type and that the type of the second term matches the argument type of the first term. This is all done by pattern matching on recursive calls to the `infer` view and the type equality view.

```
infer Γ (lam σ e) with infer (σ :: Γ) e
infer Γ (lam σ .(erase t)) | ok τ t  = ok (σ → τ) (lam σ t)
infer Γ (lam σ e)          | bad     = bad
```

Finally, the lambda case is very simple. If the body of the lambda is type correct in the extended context, then the lambda is well-typed with the corresponding function type.

Without much effort we have defined a type checker for simply typed λ-calculus that not only is guaranteed to compute well-typed terms, but also guarantees that the erasure of the well-typed term is the term you started with.

3.2 Universes

The second programming technique we will look at that is not available in non-dependently typed languages is *universe construction*. First the module header.

```
module Universes where
```

A universe is a set of types (or type formers) and a universe construction consists of a type of codes and a decoding function mapping codes to types in the universe. The purpose of a universe construction is to be able to define functions over the types of the universe by inspecting their codes. In fact we have seen an example of a universe construction already.

A familiar universe The universe of decidable propositions consists of the singleton type `True` and the empty type `False`. Codes are booleans and the decoder is the `isTrue` function.

```
data   False : Set where
record True  : Set where

data Bool : Set where
  true  : Bool
  false : Bool

isTrue : Bool -> Set
isTrue true  = True
isTrue false = False
```

Now functions over decidable propositions can be defined by manipulating the boolean codes. For instance, we can define negation and conjunction as functions on codes and prove some properties of the corresponding propositions.

```
infix  30 not_
infixr 25 _and_

not_ : Bool -> Bool
not true  = false
not false = true
```

```
_and_ : Bool -> Bool -> Bool
true  and x = x
false and _ = false

notNotId : (a : Bool) -> isTrue (not not a) -> isTrue a
notNotId true  p = p
notNotId false ()

andIntro : (a b : Bool) -> isTrue a -> isTrue b -> isTrue (a and b)
andIntro true  _ _  p = p
andIntro false _ () _
```

A nice property of this universe is that proofs of `True` can be found automatically. This means that if you have a function taking a proof of a precondition as an argument, where you expect the precondition to be trivially true at the point where you are calling the function, you can make the precondition an implicit argument. For instance, if you expect to mostly divide by concrete numbers, division of natural numbers can be given the type signature

```
open import Data.Nat

nonZero : Nat -> Bool
nonZero zero    = false
nonZero (suc _) = true

postulate _div_ : Nat -> (m : Nat){p : isTrue (nonZero m)} -> Nat

three = 16 div 5
```

Here the proof obligation `isTrue (nonZero 5)` will reduce to `True` and solved automatically by the type checker. Note that if you tell the type checker that you have defined the type of natural numbers, you are allowed to use natural number literal like `16` and `5`. This has been done in the library.

Universes for generic programming Generic programming deals with the problem of defining functions generically over a set of types. We can achieve this by defining a universe for the set of types we are interested in. Here is a simple example of how to program generically over the set of types computed by fixed points over polynomial functors.

First we define a type of codes for polynomial functors.

```
data Functor : Set1 where
  |Id|  : Functor
  |K|   : Set -> Functor
  _|+|_ : Functor -> Functor -> Functor
  _|x|_ : Functor -> Functor -> Functor
```

A polynomial functor is either the identity functor, a constant functor, the disjoint union of two functors, or the cartesian product of two functors. Since codes for functors can contain arbitrary `Sets` (in the case of the constant functor) the type of codes cannot itself be a `Set`, but lives in `Set1`.

Before defining the decoding function for functors we define datatypes for disjoint union and cartesian product.

```
data _⊕_ (A B : Set) : Set where
  inl : A -> A ⊕ B
  inr : B -> A ⊕ B

data _×_ (A B : Set) : Set where
  _,_ : A -> B -> A × B

infixr 50 _|+|_ _⊕_
infixr 60 _|x|_ _×_
```

The decoding function takes a code for a functor to a function on `Sets` and is computed recursively over the code.

```
[_] : Functor -> Set -> Set
[ |Id|    ] X = X
[ |K| A   ] X = A
[ F |+| G ] X = [ F ] X ⊕ [ G ] X
[ F |x| G ] X = [ F ] X × [ G ] X
```

Since it's called a functor it ought to support a map operation. We can define this by recursion over the code.

```
map : (F : Functor){X Y : Set} -> (X -> Y) -> [ F ] X -> [ F ] Y
map |Id|      f x       = f x
map (|K| A)   f c       = c
map (F |+| G) f (inl x) = inl (map F f x)
map (F |+| G) f (inr y) = inr (map G f y)
map (F |x| G) f (x , y) = map F f x , map G f y
```

Next we define the least fixed point of a polynomial functor.

```
data μ_ (F : Functor) : Set where
  <_> : [ F ] (μ F) -> μ F
```

To ensure termination, recursive datatypes must be strictly positive and this is checked by the type checker. Our definition of least fixed point goes through, since the type checker can spot that `[_]` is strictly positive in its second argument.

With this definition we can define a generic fold operation on least fixed points. Grabbing for the closest category theory text book we might try something like this

```
-- fold : (F : Functor){A : Set} -> ([ F ] A -> A) -> μ F -> A
-- fold F φ < x > = φ (map F (fold F φ) x)
```

Unfortunately, this definition does not pass the termination checker since the recursive call to `fold` is passed to the higher order function `map` and the termination checker cannot see that `map` isn't applying it to bad things.

To make `fold` pass the termination checker we can fuse `map` and `fold` into a single function `mapFold F G φ x = map F (fold G φ) x` defined recursively over `x`. We need to keep two copies of the functor since `fold` is always called on the same functor, whereas `map` is defined by taking its functor argument apart.

```
mapFold : forall {X} F G -> ([ G ] X -> X) -> [ F ] (μ G) -> [ F ] X
mapFold |Id|        G φ < x >   = φ (mapFold G G φ x)
mapFold (|K| A)     G φ c       = c
mapFold (F₁ |+| F₂) G φ (inl x) = inl (mapFold F₁ G φ x)
mapFold (F₁ |+| F₂) G φ (inr y) = inr (mapFold F₂ G φ y)
mapFold (F₁ |x| F₂) G φ (x , y) = mapFold F₁ G φ x , mapFold F₂ G φ y

fold : {F : Functor}{A : Set} -> ([ F ] A -> A) -> μ F -> A
fold {F} φ < x > = φ (mapFold F F φ x)
```

There is a lot more fun to be had here, but let us make do with a couple of examples. Both natural numbers and lists are examples of least fixed points of polynomial functors:

```
NatF = |K| True |+| |Id|
NAT  = μ NatF

Z : NAT
Z = < inl _ >

S : NAT -> NAT
S n = < inr n >

ListF = \A -> |K| True |+| |K| A |x| |Id|
LIST  = \A -> μ (ListF A)

nil : {A : Set} -> LIST A
nil = < inl _ >
```

```
cons : {A : Set} -> A -> LIST A -> LIST A
cons x xs = < inr (x , xs) >
```

To make implementing the argument to fold easier we introduce a few helper functions:

```
[_||_] : {A B C : Set} -> (A -> C) -> (B -> C) -> A ⊕ B -> C
[ f || g ] (inl x) = f x
[ f || g ] (inr y) = g y

uncurry : {A B C : Set} -> (A -> B -> C) -> A × B -> C
uncurry f (x , y) = f x y

const : {A B : Set} -> A -> B -> A
const x y = x
```

Finally some familiar functions expressed as folds.

```
foldr : {A B : Set} -> (A -> B -> B) -> B -> LIST A -> B
foldr {A}{B} f z = fold [ const z || uncurry f ]

plus : NAT -> NAT -> NAT
plus n m = fold [ const m || S ] n
```

Universes for overloading At the moment, Agda does not have a class system like the one in Haskell. However, a limited form of overloading can be achieved using universes. The idea is simply if you know in advance at which types you want to overload a function, you can construct a universe for these types and define the overloaded function by pattern matching on a code.

A simple example: suppose we want to overload equality for some of our standard types. We start by defining our universe:

```
open import Data.Nat
open import Data.List

data Type : Set where
  bool : Type
  nat  : Type
  list : Type -> Type
  pair : Type -> Type -> Type

El : Type -> Set
El nat        = Nat
El bool       = Bool
El (list a)   = List (El a)
El (pair a b) = El a × El b
```

In order to achieve proper overloading it is important that we don't have to supply the code explicitly everytime we are calling the overloaded function. In this case we won't have to since the decoding function computes distinct datatypes in each clause. This means that the type checker can figure out a code from its decoding. For instance, the only code that can decode into `Bool` is `bool`, and if the decoding of a code is a product type then the code must be `pair` of some codes.

Now an overloaded equality function simply takes an implicit code and computes a boolean relation over the semantics of the code.

```
infix 30 _==_
_==_ : {a : Type} -> El a -> El a -> Bool

_==_ {nat} zero    zero    = true
_==_ {nat} (suc _) zero    = false
_==_ {nat} zero    (suc _) = false
_==_ {nat} (suc n) (suc m) = n == m

_==_ {bool} true  x = x
_==_ {bool} false x = not x

_==_ {list a} [] []       = true
_==_ {list a} (_ :: _) [] = false
_==_ {list a} [] (_ :: _) = false
_==_ {list a} (x :: xs) (y :: ys) = x == y and xs == ys

_==_ {pair a b} (x₁ , y₁) (x₂ , y₂) = x₁ == x₂ and y₁ == y₂
```

In the recursive calls of `_==_` the code argument is inferred automatically. The same happens when we use our equality function on concrete examples:

```
example₁ : isTrue (2 + 2 == 4)
example₁ = _

example₂ : isTrue (not (true :: false :: [] == true :: true :: []))
example₂ = _
```

In summary, universe constructions allows us to define functions by pattern matching on (codes for) types. We have seen a few simple examples, but there are a lot of other interesting possibilities. For example

– XML schemas as codes for the types of well-formed XML documents,
– a universe of tables in a relational database, allowing us to make queries which are guaranteed to be well-typed,

4 Conclusions

In these notes I have tried to give a reasonably complete introduction to Agda and dependently typed programming. Even so, there are a lot of things missing. To fill the gaps please visit the Agda Wiki and sign up to the Agda mailing list:

- **Wiki:** `http://www.cs.chalmers.se/~ulfn/Agda`
- **List:** `https://lists.chalmers.se/mailman/listinfo/agda`

The wiki is at the moment very much under construction. Feel free to add any information that you think is missing.

Libraries for Generic Programming in Haskell

Johan Jeuring, Sean Leather, José Pedro Magalhães, and
Alexey Rodriguez Yakushev

Utrecht University, The Netherlands

Abstract.

1 Introduction

Software development often consists of designing a datatype to which functionality is added. Some functionality is datatype-specific. Other functionality is defined on almost all datatypes, and only depends on the structure of the datatype; this is called datatype-generic functionality. Examples of such functionality include comparing two values for equality, searching a value for occurrences of a particular string or other value, editing a value, and pretty-printing a value. Larger examples include XML tools, testing frameworks, debuggers, and data conversion tools.

Until recently, an instance of a datatype-generic program on a particular datatype was obtained by implementing the instance by hand. This is an error-prone task: it is often boring and reduces the productivity of programmers. Some programming languages provide standard implementations of basic datatype-generic programs such as equality of two values and printing a value. In this case, the programs are integrated into the language, and cannot be extended or adapted. So, how can we define datatype-generic programs ourselves?

More than a decade ago the first programming languages appeared that support the definition of datatype-generic programs. Using these programming languages it is possible to define a generic program which can then be used on a particular datatype without further work. Although these languages allow us to define our own generic programs, they have never grown out of the research prototype phase, and most cannot be used anymore.

The rich type system of Haskell allows us to write a number of datatype-generic programs in the language itself. The power of classes, constructor classes, functional dependencies, generalized algebraic data types, and other advanced language constructs of Haskell is impressive, and since 2001 we have seen at least 10 proposals for generic programming libraries in Haskell using one or more of these advanced constructs. Using a library instead of a separate programming language for generic programming has many advantages. The main advantages are that a user does not need a separate compiler for generic programs and that generic programs can be used out of the box. Furthermore, a library is much easier to ship, support, and maintain than a programming language, which makes the risk of using generic programs smaller. The loss of expressiveness compared with a generic programming language such as Generic Haskell is limited.

These lecture notes introduce generic programming in Haskell using libraries. We will introduce several characteristic generic programming libraries, and we will show how to use them to use and write generic programs. Furthermore, we will introduce a number of medium-sized applications which support a student with solving mathematical exercises. These 'exercise assistants' use a number of components that are instances of generic programs. In the lab exercise for these lectures, we ask you to take one of the three libraries introduced here, and use that library to turn the exercise assistants into a generic program.

These notes are organised as follows. Section 2 puts generic programming in context and introduces the various kinds of datatypes supported by Haskell and common extensions. Section 3 introduces a large example in which generic programming plays a rôle in several components. Section 4 introduces libraries for generic programming, and briefly discusses the criteria we used to select the libraries for these notes. Section 5 discusses the library known as a 'Lightweight Implementation of Generics'. Section 6 discusses the library Generics for the Masses, and Section 7 discusses Scrap Your Boilerplate. Section 8 compares the different libraries for generic programming in Haskell. Section 9 shows how you can implement generic datatypes using associated datatypes in GHC, and Section 10 concludes.

We have not been able to completely finish these lecture notes before the school. We will add Sections 8 and 9 to the final version of these notes.

2 Introduction to generic programming

Generic programming has developed as a technique for increasing the amount and scale of reuse in code while still preserving type safety. The term "generic" is highly overloaded in computer science; however, broadly speaking, most uses involve some sort of parametrisation. A generic program abstracts over the differences in separate but similar programs. In order to arrive at specific programs, one instantiates the parameter in various ways. It is the type of the parameter that distinguishes how the concept of generic programming is interpreted.

In this section, we describe the concept of generic programming and put it into context. Section 2.1 introduces a number of variations on the theme of generic programming as well as demonstrates how they may be used in Haskell. Section 2.2 reveals our focus on datatype-generic programming by discussing the world of datatypes. In our discussions, we attempt to provide understandable examples and, where possible, to use running examples in which we implement equality and/or logic. This is in keeping with the background of the exercise assistant introduced in Section 3.

2.1 Types of generic programming

There are many different variations of generic programming. Gibbons [Gibbons, 2007] lists 7 broad categories which we briefly review. In each of the following sections (titled according to the type of the parameter of the generic abstraction), we give a general description of the relevant form of generics and also provide an example of how to apply the technique in Haskell.

Value

The most basic form of generic programming is to parametrise a computation by values. The idea goes by various names in programming languages: procedure, subroutine, and function, and it is a fundamental element in mathematics. While parametrisation by value is not generally considered under the definition of "generic," it is perfectly reasonable to model other forms of genericity as functions. In this case, the function $g(x)$ represents a generic component g that is parametrised by an entity x. Instantiation of the generic component is then analogous to application of a function.

In Haskell, functions come naturally. For example, here is function that takes two Boolean values as arguments and determines their basic equality.

$$\begin{aligned}&\mathit{eq}_{\mathit{Bool}} :: \mathsf{Bool} \rightarrow \mathsf{Bool} \rightarrow \mathsf{Bool}\\ &\mathit{eq}_{\mathit{Bool}}\ a\ b = (\mathit{not}\ a \wedge \mathit{not}\ b) \vee (a \wedge b)\end{aligned}$$

Type

Commonly known as polymorphism, genericity by type refers to both type abstractions (types parametrised by other types) and polymorphic functions (functions with polymorphic types).

Haskell has excellent support for polymorphic datatypes and functions. The canonical examples are the List datatype and the *length* function. We provide the datatype for a list of Boolean values.

$$\mathbf{data}\ \mathsf{List} = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{Bool}\ \mathsf{List}$$

We then offer a simple function for calculating the length of the list.

$$\begin{aligned}&\mathit{length} :: \mathsf{List} \rightarrow \mathsf{Int}\\ &\mathit{length}\ \mathit{Nil} &&= 0\\ &\mathit{length}\ (\mathit{Cons}\ x\ xs) &&= 1 + \mathit{length}\ xs\end{aligned}$$

It is clear from observation that the List datatype need not contain only Boolean values. Notice also that *length* never makes use of the actual elements of the list, rather it only counts the quantity. We can therefore abstract over the type of the element and preserve the structure by redefining the List datatype and the *length* function:

```
data List a = Nil | Cons a (List a)

length :: List a → Int
length Nil          = 0
length (Cons x xs) = 1 + length xs
```

List a is a datatype parametrised by another type, and *length* is now a polymorphic function that can be applied to a value of type List a for any a. Note how the implementation of *length* remains the same; only the type has changed. We must only apply the constructors and function to arrive at an instance for a particular concrete type[1].

```
blist :: List Bool
blist = Cons True (Cons False (Cons True Nil))


length blist ⇝ 3
```

Function

If a function is a first-class citizen in a programming language, parametrisation of one function by another function is exactly the same as parametrisation by value. However, we explicitly mention this category because it enables abstraction over control structure. The full power of functions parametrised by functions can be seen in the higher-order functions of functional programming languages such as Haskell and ML.

Suppose we have a list of Bool values and we want to determine both the logical conjunction and logical disjunction of the list. We can define these functions as follows.

```
and :: List Bool → Bool
and Nil          = True
and (Cons p ps) = p ∧ and ps


or :: List Bool → Bool
or Nil          = False
or (Cons p ps) = p ∨ or ps
```

Now that we can see the control structure of these functions, it is evident that the same pattern of recursive operator application appears in both. To abstract from this pattern, we look at the differences between *and* and *or*, and abstract over those components.

```
foldr :: (a → b → b) → b → List a → b
```

[1] We use the notation $a \rightsquigarrow b$ to mean "in GHCi, expression a evaluates to b".

$$
\begin{aligned}
&\mathit{foldr}\ f\ n\ \mathit{Nil} &&= n\\
&\mathit{foldr}\ f\ n\ (\mathit{Cons}\ x\ \mathit{xs}) &&= f\ x\ (\mathit{foldr}\ f\ n\ \mathit{xs})
\end{aligned}
$$

The pattern that is extracted is known in the Haskell standard library as *foldr* ("fold from the right"). It captures the essence of the recursion in the functions *and* and *or*, and accepts parameters for the binary logical operator and the value for the *Nil* case. Now, we redefine *and* and *or* in simpler terms.

$$
\begin{aligned}
&\mathit{and} &&= \mathit{foldr}\ (\wedge)\ \mathit{True}\\
&\mathit{or} &&= \mathit{foldr}\ (\vee)\ \mathit{False}
\end{aligned}
$$

Interface

A generic program may also abstract over a given set of requirements, an interface of sorts. In this case, a specific program can only be instantiated by parameters that conform to this interface and the generic program will remain unaware of any unspecified aspects of the parameters. Gibbons calls the set of required operations the "structure" of the parameter; however, we think this may easily be confused with the shape of a datatype discussed later. We propose that this be called an *interface*.

This idea has been popularised by the C++ template system, in which a *concept* embodies an implicit set of operations necessary for a template to be instantiated. A concept is an implicit interface that specifies the operations necessary for instantiation. Consider this trivial C++ template function for equality:

```
template ⟨typename A⟩
bool equals (A a1, A a2){
  return a1 == a2;
}
```

The interface of the type parameter A is the implicit requirement that a value supports *operator==*(). In specific terms, this means that when the template for *equals*⟨A⟩() is expanded at the call site given a concrete type C (i.e. with the constraint that $A \equiv C$), the compiler tries to find a declaration for *operator==*(c_1, c_2). If such a declaration is not found, the template cannot be instantiated. This is different from parametric polymorphism, because the template function performs an operation on values of type A, whereas a polymorphic function needs no knowledge of A values.

Unlike the implicit concept in C++ templates, Haskell supports explicit specification of interfaces using type class constraints [Wadler and Blott, 1989], which we illustrate with a function for determining whether or not two lists are equal. Equality is of course not restricted to a single type. In fact, many different datatypes support equality. But unlike *length*, equality must be implemented differently for each type a in List a, because the equality operation requires inspection of the elements in the list. The code below defines the class of types that support the equality ((==)) and inequality (($\neq$)) operations.

```
class Eq a where
  (==), (≠) :: a → a → Bool
  a == b = not (a ≠ b)
  a ≠ b = not (a == b)
```

The type class definition includes the types of the interface operations and default implementations. For a datatype to support the operations in the class *Eq*, we create an instance of it. In our case, we will use Boolean equality as we have defined it above.

```
instance Eq Bool where
  (==) = eq_Bool
```

Now we implement our list equality function in a generic manner and the instance *Eq* Bool ensures that it will work at least with Boolean values.

```
instance Eq a ⇒ Eq (List a) where
  Nil         == Nil         = True
  (Cons x xs) == (Cons y ys) = x == y ∧ xs == ys
  _           == _           = False
```

So List a is an instance of the *Eq* type class with the provided implementation of $(==)$. The *Eq* a $\Rightarrow$ part of the instance declaration is the *type-class context*. It imposes the constraint that the element type a supports the equality operation.

Property

Gibbons expands the concept of generic programming to include properties or specifications of programs or program components. These are "generic" in the sense that a specification may hold for multiple implementations. These properties may be informally or formally defined, and, depending on the language or tool support, they may be encoded into a program, used as part of the testing process, or just appear as text.

A simple example of a property is found once again in the implementation of the equality function [Hudak et al., 1999]. A programmer using an instance of the *Eq* type class above would likely expect that $x == y \equiv \mathit{not}\ (x \neq y)$ for any values of some type a such that $x, y :: \mathit{Eq}\ \mathsf{a} \Rightarrow \mathsf{a}$. However, there is no guarantee of this. Both $(==)$ and $(\neq)$ are provided as separate methods, and the compiler cannot verify such a relationship. This informal law relies on programmers implementing the operations as expected for all instances of *Eq*.

Another example where properties are useful in the specification of multiple instances occurs in the concept of a monad. The concept comes from category theory, and it was first applied to functional programming in order to structure "impure" features such as state, exceptions, and continuations [Wadler, 1990]. Since then, uses for the monad have expanded to include general input and output [Jones and Wadler, 1993], state threads [Launchbury and Jones, 1994], and parser combinators [Hutton and Meijer, 1998], among others.

In Haskell, a particular monad is defined as an instance of the following *Monad* type class:

```
class Monad m where
  (>>=) :: m a → (a → m b) → m b
  return :: a → m a
```

The instance must provide implementations for *return* and (≫=) (pronounced "bind"), the two primary operators used to structure monadic code. We use the Maybe datatype as an example:

```
data Maybe a = Nothing | Just a
instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing >>= _ = Nothing
  return        = Just
```

The Maybe datatype serves as a simple failure-tracking mechanism. Combined with the monadic structure, we can use it to string together multiple computations that might fail:

```
computeSomething :: Int → Maybe Int
computeSomething i = do j ← computeOrFail₁ i
                        k ← do x ← computeOrFail₂ j
                               y ← computeOrFail₃ x
                               return y
                        return k
```

If any of the $computeOrFail_n$ :: Int → Maybe Int functions fails with *Nothing*, the statements following that function are not executed and *computeSomething* fails. Note that this example uses Haskell's **do**-notation which is a convenient syntactic sugar for repeated applications of (≫=) [Peyton Jones et al, 2003].

The concept of the monad is not only specified by the type class *Monad*; it also requires that the functions satisfy a number of laws:

```
return a >>= k              ≡ k a                 -- left unit
m >>= return                ≡ m                   -- right unit
m >>= (λx → k x >>= h)      ≡ (m >>= k) >>= h     -- associative
```

These three laws ensure that composition of monadic binds are associative and that *return* is both a left unit and a right unit of (≫=) [Wadler, 1992]. All instances of the *Monad* class should obey these laws. While this property cannot be directly verified by the compiler, it is important for allowing support of the **do**-notation, especially nested **do** blocks as shown in *computeSomething* above.

Program Representation

There exist many techniques in which one program is parametrised by the representation of another program (or its own). This area includes such techniques as the following:

- Code generation, such as the generation of parsers and lexical analysers. Happy [Marlow and Gill, 1997] and Alex [Dornan et al., 2003] are commonly used in Haskell for parser generation and lexical analysis, respectively.
- Reflection or the ability of a program to observe and modify its own structure and behavior. Reflection has been popularized by programming languages that support some dynamic type checking such as Java [Forman and Danforth, 1999], but some attempts have also been made in Haskell [Lämmel and Peyton Jones, 2004].
- Multi-stage programming for separating computation into stages [Taha, 1999].

Gibbons describes these ideas as genericity by stage; however, some techniques such as reflection do not immediately lend themselves to being staged. We think that this category of is better described as *metaprogramming* or generic programming in which the parameter is some form of program representation.

Perhaps the best known form of generic programming in this category is using C++ templates. The C++ template facility has led to a number of advanced libraries such as Boost [Gurtovoy and Abrahams, 2002] as well as other unexpected uses [Alexandrescu, 2001]. Partly inspired by C++ templates and the multi-stage programming language MetaML [Sheard, 1999], Template Haskell provides a metaprogramming extension to Haskell98 [Sheard and Peyton Jones, 2002].

We introduce the concept by example. An annoying itch in Haskell is the need to explicitly write selection functions for tuples of different arities. The standard library provides $\mathit{fst} :: (\mathsf{a}, \mathsf{b}) \to \mathsf{a}$ and $\mathit{snd} :: (\mathsf{a}, \mathsf{b}) \to \mathsf{b}$, because pairs are the most common form of tuples. But what about triples, quadruples, etc.? We can use Template Haskell to scratch that itch.

We want to automatically generate functions such as these:

```
fst3  = λ(x, _, _)    → x
snd4 = λ(_, x, _, _) → x
```

Using Template Haskell, we can write:

```
fst3  = $ (sel 1 3)
snd4 = $ (sel 2 4)
```

This demonstrates the use of the "splice" syntax, $(...), to evaluate the enclosed "..." at compile time. Each call to $(*sel i n*) is expanded to a function that selects the *i*-th component of a *n*-tuple. Consider the following implementation [2]:

```
sel :: Int → Int → ExpQ
sel i n = lamE pats body
```

[2] The code for *sel* is derived from the original example [Sheard and Peyton Jones, 2002] with modifications to simplify it and to conform to the *Language.Haskell.TH* library included with GHC 6.8.2.

```
where pats  = [tupP (map varP as)]
      body = varE (as !! (i − 1))
      as    = [mkName ("a" ++ show j) | j ← [1 . . n]]
```

In order to define *sel*, we need to create an abstract syntax recipe of the form $\lambda(a_1, a_2, ..., ai, ..., an) \rightarrow ai$. We pull each of our ingredients from the Template Haskell libraries. First, we need a lambda expression (*lamE*) to create a function. A lambda expression requires a list of patterns on the left of the → and an expression on the right. In our pattern, we use a tuple (*tupP*) and a list of *n* variables (*varP*) whose names are generated (using *mkName*) based on their position in the tuple. In the body of the lambda abstraction, we need an expression containing a variable (*varE*) that represents the *i*-th component. Finally, our function creates an expression of type ExpQ that can subsequently be evaluated within a splice.

Note that Template Haskell is type-safe and that a well-typed program cannot "go wrong" at run-time [Milner, 1978]. In the first stage of Template Haskell compilation, the expression inside the $(...) is type-checked. Next, this expression is compiled and executed, and the resulting piece of syntax is spliced into place where the call is made. Finally, the resulting Haskell program is completely type-checked as if this were again the initial stage of compilation. To put this into the context of our example above, compilation could fail for several reasons, namely the function *sel* does not type-check or a call to $(*sel i n*) generates code that does not type-check.

Template Haskell has also been explored for other uses in generic programming. Most notably, it is possible to prototype datatype-generic extensions to Haskell with Template Haskell [Norell and Jansson, 2004a].

Shape

Recall the implementation of equality as defined by the type class *Eq*. Note that we gave only two instances, one for Bool and one for List a. In order for *Eq* to be useful, we need to provide a large number of instances: one for every primitive type and one for every algebraic datatype. While we cannot escape specialized definitions for basic types such as Int and Char, we should be able to abstract over the similarities of many algebraic datatypes. Intuitively, we can visualize these similarities simply by reviewing the syntax for datatypes in Haskell:

```
data List a = Nil | Cons a (List a)
```

From our understanding of the **data** declaration, we know that List a is constructed by either a *Nil* or a *Cons*. This choice may be called a *sum* and denoted by the + symbol. We see that the *Cons* takes two arguments, a and another List a. We refer to this pair-like constructor as a *product* and use the × symbol. A constructor that takes no arguments (such as *Nil*) also has the special designator of *unit*, often represented by **1** or $\mathbb{1}$. We can now reconstruct the List datatype in the new formulation:

$$\textbf{data}\ \mathsf{List\ a} = \mathbb{1} + (\mathsf{a} \times \mathsf{List\ a})$$

The *sum of products* view allows us to abstract over the *shape* or *structure* of a datatype. Each regular datatype can be formulated as an equation of sums, products, and units. As another, slightly more complicated, example, take the following Tree3 datatype with its sum of products view:

$$\begin{array}{l}
\textbf{data}\ \mathsf{Tree3\ a} = \mathit{Twig} \mid \mathit{Leaf}\ \mathsf{a} \mid \mathit{Branch}\ (\mathsf{Tree3\ a})\ \mathsf{a}\ (\mathsf{Tree3\ a}) \\
\textbf{data}\ \mathsf{Tree3\ a} = \mathbb{1} + (\mathsf{a} + (\mathsf{Tree3\ a} \times (\mathsf{a} \times \mathsf{Tree3\ a})))
\end{array}$$

Tree3 has three alternatives: *Twig*, *Leaf*, and *Branch*. To simulate this in our binary + representation, we use right-associative nested sums. Additionally, we use right-associative nested products to simulate the *Branch* constructor with more than two arguments. Overall, these approaches simplify the view, since the representation of any datatype only need the nullary unit and the binary sum and product.

The sum of products view is used in projects such as Generic Haskell [Löh, 2004] to achieve datatype-generic programming. In fact, Generic Haskell provides us with a solution to the problem with our *Eq* type class mentioned above. Consider this implementation of datatype-generic equality:

$$\begin{array}{lllll}
\multicolumn{5}{l}{eq\{\!|\mathsf{a} :: \star|\!\} :: (eq\{\!|\mathsf{a}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}} \\
eq\{\!|\mathsf{Unit}|\!\} & \_ & \_ & = & \mathit{True} \\
eq\{\!|\mathsf{Int}|\!\} & x & y & = & eq_{\mathit{Int}}\ x\ y \\
eq\{\!|\mathsf{Char}|\!\} & x & y & = & eq_{\mathit{Char}}\ x\ y \\
eq\{\!|\mathsf{a} + \mathsf{b}|\!\} & (\mathit{Inl}\ x) & (\mathit{Inl}\ y) & = & eq\{\!|\mathsf{a}|\!\}\ x\ y \\
eq\{\!|\mathsf{a} + \mathsf{b}|\!\} & (\mathit{Inl}\ x) & (\mathit{Inr}\ y) & = & \mathit{False} \\
eq\{\!|\mathsf{a} + \mathsf{b}|\!\} & (\mathit{Inr}\ x) & (\mathit{Inl}\ y) & = & \mathit{False} \\
eq\{\!|\mathsf{a} + \mathsf{b}|\!\} & (\mathit{Inr}\ x) & (\mathit{Inr}\ y) & = & eq\{\!|\mathsf{b}|\!\}\ x\ y \\
eq\{\!|\mathsf{a} \times \mathsf{b}|\!\} & (x_1 \times y_1) & (x_2 \times y_2) & = & eq\{\!|\mathsf{a}|\!\}\ x_1\ x_2 \wedge eq\{\!|\mathsf{b}|\!\}\ y_1\ y_2
\end{array}$$

The $eq\{\!|\mathsf{a} :: \star|\!\}$ function uses pattern matching on types to perform case analysis. Thus, each of the primitive types are specifically defined and the algebraic datatypes are supported with the generic sum of products view. We will not explain this function in detail. We use Generic Haskell as an example of a datatype-generic extension to Haskell, because it has a minimal syntax that is relatively easy to understand. However, Generic Haskell is a language extension of Haskell for which we need a separate compiler/preprocessor to perform specialization for functions such as $eq\{\!|\mathsf{a} :: \star|\!\}$ on the datatypes on which it is needed. These lecture notes focus on library support for generic programming instead.

There are generic views other than the sum of products. For example, we may regard a datatype as a fixed-point, allowing us to make all recursion in the datatype explicit. Another example is the spine view which we discuss in a varient of the Scrap Your Boilerplate library in Section 7. For a more in-depth study of generic views, refer to [Holdermans et al., 2006].

2.2 The world of Haskell datatypes

Datatypes play a central rôle in programming in Haskell. Solving a problem often consists of designing a datatype, and defining functionality on that datatype. Haskell offers a powerful construct for defining datatypes: **data**. Haskell also offers two other constructs: **type** to introduce type synonyms and **newtype**, a restricted version of **data**. We will mainly use the **data** construct in these notes.

Datatypes come in many variations: we have finite, regular, nested, and many more kinds of datatypes. This subsection introduces many of these variations of datatypes by example. Not all datatypes are pure Haskell 98, some require extensions to Haskell. Many of these extensions are supported by most Haskell compilers, some only by GHC. On the way, we will explain kinds and show how they are used to classify types. In addition to the datatype definitions, we will also define an equality function on the datatypes we introduce. As we will see, the definitions of equality on the different datatypes follow the same pattern. This pattern will be used to define a generic program for equality later in these notes.

Monomorphic datatypes

We start our journey through datatypes with lists containing values of a particular type. For example, in the previous subsection we have defined the datatype of lists of booleans:

$$\textbf{data}\ \mathsf{List} = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{Bool}\ \mathsf{List}$$

We define a new datatype, called List, which has two kinds of values: an empty list (represented by the constructor *Nil*), or a list consisting of a boolean value in front of another List. This datatype is the same as Haskell's predefined list datatype containing booleans, with $[\,]$ and $(:)$ as constructors. Since the datatype List does not take any type parameters, it has base kind $\star$. Other examples of datatypes of kind $\star$ are Int, Char, $[\mathsf{Maybe}\ ()]$, etc.

Here is the equality function on this datatype:

$$\begin{array}{llll}
\mathit{eq}_{\mathit{List}_{\mathit{Bool}}} :: \mathsf{List} \to \mathsf{List} \to \mathsf{Bool} \\
\mathit{eq}_{\mathit{List}_{\mathit{Bool}}}\ \mathit{Nil} & \mathit{Nil} & = \mathit{True} \\
\mathit{eq}_{\mathit{List}_{\mathit{Bool}}}\ (\mathit{Cons}\ b_1\ l_1) & (\mathit{Cons}\ b_2\ l_2) & = \mathit{eq}_{\mathit{Bool}}\ b_1\ b_2 \wedge \mathit{eq}_{\mathit{List}_{\mathit{Bool}}}\ l_1\ l_2 \\
\mathit{eq}_{\mathit{List}_{\mathit{Bool}}}\ \_ & \_ & = \mathit{False}
\end{array}$$

Two empty lists are equal, and two nonempty lists are equal if their head elements are the same (which we check using equality on Bool) and their tails are equal. An empty list and a nonempty list are unequal.

Polymorphic datatypes

We abstract from the datatype of booleans in the type List to obtain polymorphic lists.

$$\textbf{data}\ \mathsf{List\ a} = \mathit{Nil}_{\mathit{List}} \mid \mathit{Cons}_{\mathit{List}}\ \mathsf{a}\ (\mathsf{List\ a})$$

Compared with List, the List a datatype has a different functional structure: its kind is $\star \to \star$. Kinds represent the structure of datatypes, and are either $\star$ (base kind) or $\kappa \to \nu$, where κ and ν are kinds. Since List takes a base type as argument, it has the functional kind $\star \to \star$. Actually, nothing is known about the type variable a, and in such a case, Haskell defaults its kind to $\star$.

Equality on polymorphic lists is almost the same as equality on lists of booleans:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{eq}_{\mathit{List}} :: (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to \mathsf{List\ a} \to \mathsf{List\ a} \to \mathsf{Bool}} \\
\mathit{eq}_{\mathit{List}}\ \mathit{eq}_a\ \mathit{Nil}_{\mathit{List}} & \mathit{Nil}_{\mathit{List}} & = \mathit{True} \\
\mathit{eq}_{\mathit{List}}\ \mathit{eq}_a\ (\mathit{Cons}_{\mathit{List}}\ x_1\ l_1) & (\mathit{Cons}_{\mathit{List}}\ x_2\ l_2) & = \mathit{eq}_a\ x_1\ x_2 \land \mathit{eq}_{\mathit{List}}\ \mathit{eq}_a\ l_1\ l_2 \\
\mathit{eq}_{\mathit{List}}\ \_\quad \_ & \_ & = \mathit{False}
\end{array}$$

The only difference with equality on List is that we need to have some means of determining equality of the elements of the list, so we need an additional equality function of type $(\mathsf{a} \to \mathsf{a} \to \mathsf{Bool})$ as parameter[3].

Families and mutually-recursive datatypes

A family of datatypes is a set of datatypes that may use each other. Below we present a non-recursive family of datatypes, which represents a simplified representation of a system of linear expressions. A system of linear equations is a list of equations, in which an equation consists of two linear expressions. Here is an example:

$$\begin{array}{llll}
2\,x & +3\,y & +z & = 5 \\
x & -y & & = 1 \\
x & +y & +z & = 7
\end{array}$$

For simplicity, we assume linear expressions are values of type Expr a, a datatype for arithmetic expressions. An arithmetic expression abstracts over the type of constants, typically an instance of the *Num* class, and is a variable, a literal, or the addition, subtraction, multiplication, or division of two arithmetic expressions. Of course, the domain of expressions also contains non-linear expressions, in which a variable is multiplied with itself or another variable, so this description is not type-safe.

$$\begin{array}{l}
\textbf{type}\ \mathsf{LinearSystem} = \mathsf{List\ LinearExpr} \\
\textbf{data}\ \mathsf{LinearExpr} \quad = \mathit{Equation}\ (\mathsf{Expr\ Int})\ (\mathsf{Expr\ Int}) \\
\textbf{infixl}\ 6\ \times, \div \\
\textbf{infixl}\ 5\ +, - \\
\textbf{data}\ \mathsf{Expr\ a} = \mathit{ExprVar}\ \mathsf{String}
\end{array}$$

[3] Using Haskell's type classes, this would correspond to replacing the type of the first argument in the type of $\mathit{eq}_{\mathit{List}}$ by an *Eq* a $\Rightarrow$ constraint.

$|\ \textit{Lit}\ \mathsf{a}$
$|\ \mathsf{Expr\ a} + \mathsf{Expr\ a}$
$|\ \mathsf{Expr\ a} - \mathsf{Expr\ a}$
$|\ \mathsf{Expr\ a} \times \mathsf{Expr\ a}$
$|\ \mathsf{Expr\ a} \div \mathsf{Expr\ a}$

Even though these datatypes are defined in terms of each other, there is no mutual recursion: Expr a is at the end of the hierarchy and is defined only in terms of itself. Defining equality for LinearSystem is trivial and results in non-mutually recursive equality functions.

However, datatypes in Haskell can also be mutually recursive, as can be seen in the following example. A forest is either empty or a tree followed by a forest, and a tree is either empty or a node of a forest:

```
data Tree a   = Empty | Node  a        (Forest a)
data Forest a = Nil   | Const (Tree a) (Forest a)
```

Defining the equality function for these datatypes amounts to defining the equality function for each datatype separately. The result is a set of mutually recursive functions:

$eq_{Tree} :: (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to \mathsf{Tree\ a} \to \mathsf{Tree\ a} \to \mathsf{Bool}$
$eq_{Tree}\ eq_a\ Empty\quad Empty \quad = True$
$eq_{Tree}\ eq_a\ (Node\ a_1\ f_1)\ (Node\ a_2\ f_2) = eq_a\ a_1\ a_2 \wedge eq_{Forest}\ eq_a\ f_1\ f_2$
$eq_{Tree}\ \_\quad \_ \quad \_ \quad = False$

$eq_{Forest} :: (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to \mathsf{Forest\ a} \to \mathsf{Forest\ a} \to \mathsf{Bool}$
$eq_{Forest}\ eq_a\ Nil\quad Nil \quad = True$
$eq_{Forest}\ eq_a\ (Const\ t_1\ f_1)\ (Const\ t_2\ f_2) = eq_{Tree}\ eq_a\ t_1\ t_2 \wedge eq_{Forest}\ eq_a\ f_1\ f_2$
$eq_{Forest}\ \_\quad \_ \quad \_ \quad = False$

Higher-order kinded datatypes

Consider the following minimal datatype for logic expressions, together with an equality function:

$\textbf{data}\ \mathsf{Logic_s} = \textit{Lit}\ \mathsf{Bool}$
$\quad |\ \textit{Not}\ \mathsf{Logic_s}$
$\quad |\ \textit{Or}\ \mathsf{Logic_s}\ \mathsf{Logic_s}$

$eq_{Logic_s} :: \mathsf{Logic_s} \to \mathsf{Logic_s} \to \mathsf{Bool}$
$eq_{Logic_s}\ (Lit\ x_1)\quad (Lit\ x_2) \quad = eq_{Bool}\ x_1\ x_2$
$eq_{Logic_s}\ (Not\ e_1)\quad (Not\ e_1') \quad = eq_{Logic_s}\ e_1\ e_1'$
$eq_{Logic_s}\ (Or\ e_1\ e_2)\ (Or\ e_1'\ e_2') = eq_{Logic_s}\ e_1\ e_1' \wedge eq_{Logic_s}\ e_2\ e_2'$
$eq_{Logic_s}\ \_ \quad \_ \quad = False$

Again, the equality function follows the pattern: for each pair of constructors with the same name, either recursively compare their arguments pairwise, or, if they do not take arguments, return *True*. Comparing different constructors results in *False*.

But suppose we now want to use the fact that disjunction is associative, and represent our $\mathsf{Logic_s}$ datatype as:

$$\begin{aligned}
&\textbf{data}\ \mathsf{Logic_L} = \mathit{Lit}\ \ \mathsf{Bool}\\
&\qquad\qquad\quad \mid \mathit{Not}\ \mathsf{Logic_L}\\
&\qquad\qquad\quad \mid \mathit{Or}\ \ (\mathsf{List}\ \mathsf{Logic_L})
\end{aligned}$$

We can abstract from the "container type" List, which contains the subexpressions, by introducing a type argument for it.

$$\begin{aligned}
&\textbf{data}\ \mathsf{Logic_F}\ \mathsf{f} = \mathit{Lit}\ \ \mathsf{Bool}\\
&\qquad\qquad\quad\ \ \mid \mathit{Not}\ (\mathsf{Logic_F}\ \mathsf{f})\\
&\qquad\qquad\quad\ \ \mid \mathit{Or}\ \ (\mathsf{f}\ (\mathsf{Logic_F}\ \mathsf{f}))
\end{aligned}$$

We have introduced a type variable, and so $\mathsf{Logic_F}$ does not have kind $\star$ as its predecessors. However, its kind is also not $\star \to \star$, as we have seen previously in the List datatype, because the type argument that $\mathsf{Logic_F}$ expects is not a base type, but a "type transformer": we can see in the application $\mathsf{f}\ (\mathsf{Logic_F}\ \mathsf{f})$ that f is applied to an argument. The kind of $\mathsf{Logic_F}$ is thus: $(\star \to \star) \to \star$. This datatype is a higher-order kinded datatype.

To better understand abstraction over container types, consider the following type:

$$\textbf{type}\ \mathsf{Logic'_L} = \mathsf{Logic_F}\ \mathsf{List}$$

Modulo undefined values, $\mathsf{Logic'_L}$ is isomorphic to $\mathsf{Logic_L}$. The type argument of $\mathsf{Logic_F}$ describes what "container" will be used for the elements of the *Or* case.

Defining equality for the $\mathsf{Logic'_L}$ datatype is simple:

$$\begin{aligned}
&\mathit{eq}_{\mathit{Logic'_L}} :: \mathsf{Logic'_L} \to \mathsf{Logic'_L} \to \mathsf{Bool}\\
&\mathit{eq}_{\mathit{Logic'_L}}\ (\mathit{Lit}\ x_1)\ \ (\mathit{Lit}\ x_2)\ \ = \mathit{eq}_{\mathit{Bool}}\ x_1\ x_2\\
&\mathit{eq}_{\mathit{Logic'_L}}\ (\mathit{Not}\ x_1)\ (\mathit{Not}\ x_2) = \mathit{eq}_{\mathit{Logic'_L}}\ x_1\ x_2\\
&\mathit{eq}_{\mathit{Logic'_L}}\ (\mathit{Or}\ l_1)\ \ \ (\mathit{Or}\ l_2)\ \ \ =\\
&\quad \mathit{length}\ l_1 \equiv \mathit{length}\ l_2 \wedge \mathit{and}\ (\mathit{zipWith}\ \mathit{eq}_{\mathit{Logic'_L}}\ l_1\ l_2)\\
&\mathit{eq}_{\mathit{Logic'_L}}\ \_\qquad\quad\ \_\qquad\quad\ \ = \mathit{False}
\end{aligned}$$

Note that we use the $\mathit{zipWith} :: (\mathsf{a} \to \mathsf{b} \to \mathsf{c}) \to \mathsf{List}\ \mathsf{a} \to \mathsf{List}\ \mathsf{b} \to \mathsf{List}\ \mathsf{c}$ function because we know the container is the list type.

We can also provide the equality function for the $\mathsf{Logic_F}$ type:

$$\begin{aligned}
&\mathit{eq}_{\mathit{Logic_F}} :: (\forall \mathsf{a}\,.\,(\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to \mathsf{f}\ \mathsf{a} \to \mathsf{f}\ \mathsf{a} \to \mathsf{Bool}) \to\\
&\qquad\qquad\quad \mathsf{Logic_F}\ \mathsf{f} \to \mathsf{Logic_F}\ \mathsf{f} \to \mathsf{Bool}
\end{aligned}$$

$$\begin{aligned}
&eq_{Logic_F}\ eq_f\ (Lit\ x_1)\ \ (Lit\ x_2)\ &&= eq_{Bool}\ x_1\ x_2\\
&eq_{Logic_F}\ eq_f\ (Not\ x_1)\ (Not\ x_2) &&= eq_{Logic_F}\ eq_f\ x_1\ x_2\\
&eq_{Logic_F}\ eq_f\ (Or\ x_1)\ \ (Or\ x_2) &&= eq_f\ (eq_{Logic_F}\ eq_f)\ x_1\ x_2\\
&eq_{Logic_F}\ \_\ \ \_\ \ \ \ \ \ \ \ \_ &&= False
\end{aligned}$$

This definition is considerably more complex than the previous one. The added complexity is caused by the fact that we do not know what the "container" type is, and therefore we have to abstract from an equality function on such a type. However, since that type is just a container, its equality function also needs to abstract from an equality function on the contained type! This requires the use of rank-2 polymorphism, a common extension to Haskell, and is indicated by the presence of the $\forall a$ in the type signature of eq_{Logic_F}. When invoking this function, the user supplies the first argument (the equality function on the f-type).

Nested datatypes

A *regular* data type is a recursive, parametrised type whose definition does not involve a change of the type parameter(s). All the datatypes we have introduced so far are regular. However, it is also possible to define so-called nested datatypes [Bird and Meertens, 1998]. In a nested datatype, recursive occurrences of the datatype have other type arguments than the datatype being defined. Perfect binary trees are an example of such a datatype:

$$\textbf{data}\ \mathsf{Perfect\ a} = Leaf\ \mathsf{a} \mid Node\ (\mathsf{Perfect}\ (\mathsf{a},\mathsf{a}))$$

Any value of this datatype is a full binary tree in which all leaves are at the same depth. This is attained by using the pair constructor in the recursive call for the *Node* constructor. An example of such tree is:

$$perfect = Node\ (Node\ (Node\ (Leaf\ (((1,2),(3,4)),((5,6),(7,8))))))$$

Here is the equality function on Perfect:

$$\begin{aligned}
&eq_{Perfect} :: (\forall \mathsf{a}\,.\,\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to \mathsf{Perfect\ b} \to \mathsf{Perfect\ b} \to \mathsf{Bool}\\
&eq_{Perfect}\ eq_a\ (Leaf\ x_1)\ \ (Leaf\ x_2)\ &&= eq_a\ x_1\ x_2\\
&eq_{Perfect}\ eq_a\ (Node\ x_1)\ (Node\ x_2) &&= eq_{Perfect}\ eq_a\ x_1\ x_2\\
&eq_{Perfect}\ \_\ \ \_\ \ \ \ \ \ \ \ \_ &&= False
\end{aligned}$$

This definition is again very similar to the equality on datatypes we have introduced before. An interesting aspect of this definition is that it needs rank-2 polymorphism (as in eq_{Logic_F}), since the type of the elements at the leaves depends on how deep the tree is.

Existentially quantified datatypes

Many of the datatypes we have seen take arguments, and in the type of the constructors of these datatypes, these type argument are universally quantified. For example, the constructor $Cons_{List}$ of the datatype List a has type a $\rightarrow$ List a $\rightarrow$ List a for all types a. However, we can also use existential types, which "hide" a type variable that only occurs under a constructor. Consider the following example:

```
data Dynamic = ∀a . Dyn (Rep a) a
```

The type Dynamic encapsulates a type a and its representation, a value of type Rep a. We will encounter this datatype later in these lecture notes (Section 5), where it is used to check and convert between types at run-time in a type-safe fashion. Despite the use of the $\forall$ symbol, the type variable a is said to be existentially quantified because it is only available inside the constructor — Dynamic has kind $\star$. Existential datatypes are typically used to encapsulate some type with its corresponding actions: in the above example, the only thing we can do with a Dynamic is inspect its representation. Other important applications of existentially quantified datatypes include the implementation of abstract datatypes, which encapsulate a type together with a set of operations. Existential datatypes are not part of the Haskell 98 standard, but they are a fairly common extension.

The definition of equality on an existentially quantified datatype may be problematic. We can only compare two values if the operations provided by the constructor allow us to compare two values. For example, if the only operation provided by the constructor is a string representation of the value, we can only compare the string representation of two values. In Section 5 we will show how to compare two dynamic values.

Generalized algebraic data types

Another powerful extension to the Haskell 98 standard are generalized algebraic data types (GADTs). A GADT is a datatype in which different constructors may have related but different result types. Consider the following example, where we combine the datatypes $\mathsf{Logic_s}$ and $\mathsf{Expr_s}$ shown before in a datatype for statements:

```
data Stat a where
  Val   :: Expr Int    → Stat (Expr Int)
  Term  :: Logics      → Stat Logics
  If    :: Stat Logics → Stat a → Stat a → Stat a
  Write :: Stat a      → Stat ()
  Seq   :: Stat a      → Stat b → Stat b
```

The new aspect here is the ability to give each constructor a different result type Stat.... This has the advantage that we can describe the type of the different constructors more precisely. For example, the type of the *If* constructor now says that the first argument of the *If* should return a logic expression, and the expressions returned in the then and else branches may be of any type, as long as they have the same type.

Defining equality of two statements is still a matter of repeating similar code:

$$
\begin{array}{llll}
\multicolumn{4}{l}{eq_{Stat} :: \mathsf{Stat\ a} \rightarrow \mathsf{Stat\ b} \rightarrow \mathsf{Bool}} \\
eq_{Stat}\ (Val\ x_1) & (Val\ x_2) & = & eq\_Expr\ (==)\ x_1\ x_2 \\
eq_{Stat}\ (Term\ x_1) & (Term\ x_2) & = & eq_{Logic_s}\ x_1\ x_2 \\
eq_{Stat}\ (If\ x_1\ x_2\ x_3) & (If\ x'_1\ x'_2\ x'_3) & = & eq_{Stat}\ x_1\ x'_1 \wedge eq_{Stat}\ x_2\ x'_2 \wedge eq_{Stat}\ x_3\ x'_3 \\
eq_{Stat}\ (Write\ x_1) & (Write\ x_2) & = & eq_{Stat}\ x_1\ x_2 \\
eq_{Stat}\ (Seq\ x_1\ x_2) & (Seq\ x'_1\ x'_2) & = & eq_{Stat}\ x_1\ x'_1 \wedge eq_{Stat}\ x_2\ x'_2 \\
eq_{Stat}\ \_ & \_ & = & False
\end{array}
$$

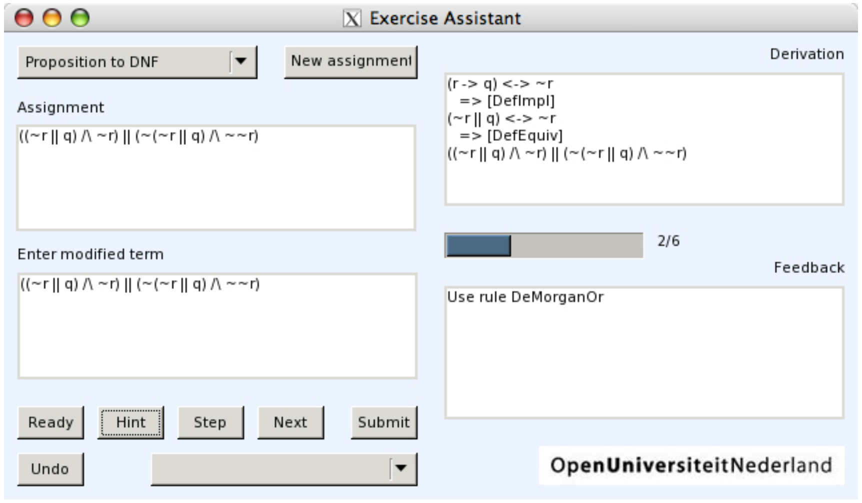
We have shown many kinds of datatypes and an example of a function that offers functionality needed for many datatypes. We have seen that we can define these functions ourselves, but the code quickly becomes repetitive and tedious. Furthermore, if the datatypes change, the definitions will have to be changed accordingly. This is not only inefficient and time-consuming but also error-prone. In the next chapter we will introduce a larger example in which many different datatypes naturally arise, on which we have to implement the same functionality. This example will further illustrate the need for generic programming.

3 Lab assignment: Exercise Assistants

At the Open University NL and Utrecht University, we are developing several exercise assistants. We have developed an exercise assistant that supports interactively solving a system of linear equations [Passier and Jeuring, 2006], an assistant that supports calculating a disjunctive normal form (DNF) of a logical expression [Lodder et al., 2006], and an assistant that supports several kinds of exercises within linear algebra. A screenshot of the assistant that supports calculating a DNF of a logical expression is shown in Figure 1.

The different exercise assistants can be viewed as instances of a generic program, but they have not been implemented using generic programming techniques. As a consequence, each time we get a request for an exercise assistant on a new domain, which happens not infrequently, we have to reimplement quite a lot of boilerplate code for the new domain. We have done so with the domains of arithmetics expressions, used in exercises about simplifying fractions, and relational algebra, used in exercises for normalizing relational algebra expressions. We would like to simplify creating an instance of the exercise assistant for a new domain, by making the exercise assistants instances of a generic program.

In this lab assignment it is your task to take any of the libraries discussed in the following sections, and to use the library to implement (parts of the) exercise assistants as a generic program.



```
((~r || q) /\ ~r) || (~(~r || q) /\ ~~r)
```

```
(r -> q) <-> ~r
  => [DefImpl]
(~r || q) <-> ~r
  => [DefEquiv]
((~r || q) /\ ~r) || (~(~r || q) /\ ~~r)
```

```
((~r || q) /\ ~r) || (~(~r || q) /\ ~~r)
```

```
Use rule DeMorganOr
```

Fig. 1. The Exercise Assistant

This section introduces exercise assistants. We first give some general background behind these assistants. Then we discuss the generic components which are needed to turn the instances of a generic program into a generic program. Divide the lab into three parts:

- analyse which generic components can be implemented using the library you have selected;
- implement these generic components as stand-alone generic programs;
- replace the uses of datatype-specific code in the exercise assistants with instances of the generic programs.

We think this lab exercise is rather ambitious, but it should not be too hard to make some progress in the first part.

In this lab assignment you can use the following materials:

- modules that implement the different generic programming libraries for Haskell;
- modules that implement the various exercise assistants.

For the first part of the lab assignment you only need one of the library modules, for the second part you'll need the code for the exercise assistants.

3.1 An introduction to exercise assistants

This subsection introduces our exercise assistants. We describe the characteristics of exercise assistants, their user interface, and their main components. We will use our exercise assistant that supports solving exercises about calculating a DNF of a logical expression as the running example.

Characteristics. The exercise assistants we are building have the following characteristics:

- An assistant is interactive, so a student solves an exercise step by step and receives semantically rich feedback after each erroneous step. Using this feedback, a student can correct his or her mistake or adjust the solving strategy.
- A student enters and rewrites expressions in the working area. This approach mimics the pen-and paper situation as close as possible; a student can make syntactical as well as semantical mistakes.
- Feedback is produced automatically. For example, the exercise assistant that supports calculating the DNF of a logical expression, gives feedback about any errors made in any of the exercises. The feedback need not be specified with every exercise, but is automatically calculated based on the exercise, the available rewrite rules, possibly the known "buggy" rewrite rules [Brown and VanLehn, 1980], the strategy for solving the exercise, and the expression entered by the student. The feedback is not hard coded, but is generated algorithmically on the level of rewrite steps and of the strategy for solving the exercise.

The distinguishing feature of our assistants is that they give very good feedback. Although giving good feedback is generally acknowledged to be vital for learning [Mory, 2003], current e-learning systems that support incrementally solving exercises lack sophisticated techniques for giving feedback. We hope to improve upon this situation.

The user interface and feedback. The user interface of the exercise assistant, shown in Figure 1, consists of four text fields and nine buttons. The current exercise assistant combines all the kinds of exercises we support in a single application. The top-left bottom is set to 'Proposition to DNF', other kinds of exercises that can be selected here are 'Simplifying fractions', 'Gaussian elimination', etc. Figure 1 shows the exercise assistant halfway solving an exercise, just after a user has clicked the Hint button.

The text fields. The top-left text field shows the current term, in this case the logical formula that has to be transformed to DNF. The current formula to be rewritten is `((~r || q) /\ ~r) || (~(~r || q) /\ ~~r)`, where `~` denotes logical negation $\neg$, and `||` denotes logical or $\vee$. The second text field is the working area in which the student stepwise edits the logical formula into a DNF. The third top-right text field displays the derivation that the student has performed until now, with the justification of the steps (the name of the rewrite rules that have been applied) in between the terms. Finally, the fourth text-field displays feedback to the student. Since the student has asked for a hint, it tells the student to use the rule 'DeMorganOr'. If a student applies a rule correctly, it tells the student which rule has been applied. If a student applies a rule that does not fit the strategy for the exercise it tells the student to step back and try again. We have not yet added detailed feedback for when a buggy rule is applied.

The buttons. A student selects the particular kind of exercises to practice with the top-left button. The top-right button is then used to generate a new exercise, which is presented in the top-left text field and the working area. After editing the formula, a student clicks the Submit button to obtain feedback, and the Undo button to undo the last step (or any amount of steps). If a student wants a hint, he or she clicks the Hint or Step button. Clicking the Hint button gives a suggestion about how to proceed, whereas clicking the Step button gives a detailed next step. If for example the current formula is $\neg q \vee \neg(t \wedge t)$, clicking the Hint button gives the message: *"You can apply the De Morgan rule"*, and clicking the Step button gives the message *"Apply the De Morgan rule on $\neg(t \wedge t)$, giving $\neg t \wedge \neg t$"*. The Next button performs the next step in the calculation. This button can be used to play the stepwise solution of an exercise for a student. The Ready button is clicked when a student thinks the exercise is correctly solved. If the formula is not yet in DNF, the exercise assistant gives the massage: *"Sorry, you have not yet reached a solution"*, and otherwise *"Congratulations: you have reached a solution!"*. If a student has solved an exercise, but does not click the Ready button, the exercise assistant gives the feedback *"No more steps left to do"*.

Feedback messages. A student can make different kinds of mistakes when solving an exercise in our exercise assistants:

- syntactical mistakes, for example when a student writes $\neg(q \wedge (t \wedge))$ instead of $\neg(q \wedge (t \wedge t))$ (the and operator $\wedge$ needs two arguments) or a formula with a missing parenthesis;
- rewrite step mistakes, such as forgetting to change a disjunction into a conjunction when applying the De Morgan rule. For example $\neg(t \wedge t)$ is rewritten to $\neg t \wedge \neg t$ instead of $\neg t \vee \neg t$;
- strategic mistakes, such pushing the operator or to top level before all negations have been pushed in front of constants or variables. This is not a real mistake, but it is not a clever way to solve the exercise, since after pushing the negations down, the operator or has to be pushed to top level again.

The exercise assistant reports the syntactical mistakes using Swierstra's error-correcting parsers [Swierstra and Duponcheel, 1996]. At the moment the error message is rather basic, but this can easily be improved. We can either use the generic feedback provided by the error-correcting parsers, or we can give domain-specific syntactic error messages [Horacek and Wolska, 2006]. When a student performs an erroneous rewrite step, this is reported. We have developed techniques for explaining erroneous rewrite steps, but they have not yet been included in our exercise assistant. Finally, when a student makes a strategic mistake, the assistant tells the student about it, and asks to try again. At the moment the message does not contain any detail, but pressing undo and asking for a hint will tell the student what the assistant expects at this point in the exercise.

3.2 The main components

To model intelligence in a computer program, Bundy [Bundy, 1983] identifies three important, basic needs:

– The need to have knowledge about the domain.
– The need to reason with that knowledge.
– The need for knowledge about how to direct or guide that reasoning.

Our exercise assistants take instantiations of these needs as input. Each exercise assistant needs a domain description (for example: systems of linear equations, or logical expressions), together with a concrete representation of the domain (how are the expressions presented to the student); rules for reasoning about the domain (for example: multiplication distributes over addition, de Morgan for logical expressions); and one or more strategies for solving exercises in the domain (for example: first move occurrences of $\neg$ in front of logical variables, and then distribute $\wedge$ over $\vee$ to obtain an expression in DNF).

This subsection briefly illustrates these three needs for the domain of logic expressions. Then we discuss the components of the exercise assistants that only depend on the structure of the domain for which the exercise assistant has been developed. These components should be implemented as generic programs.

The domain of logic expressions

The domain. A logic expression is a value of a datatype that is an extension of the datatype *logicS* defined in Section 2.2. It is a logic variable, a constant *true* or *false* (written *True* and *False*), the negation of a logic expression, or the disjunction, implication, or equivalence of two logic expressions. Note that we use *True*, *False*, $\wedge$, and $\vee$ as constructor names of Logic expressions in this section, instead of functions on the basic Haskell datatype Bool. In the other sections of these notes, these names denote the standard functions on Bool, unless explicitly stated otherwise.

```
infixr 1 ↔
infixr 2 →
infixr 3 ∨
infixr 4 ∧
data Logic = LogicVar String
           | True
           | False
           | ¬Logic
           | Logic → Logic
           | Logic ↔ Logic
           | Logic ∧ Logic
           | Logic ∨ Logic
```

If necessary, we write parentheses to resolve ambiguities. Examples of valid expressions are $\neg(p \vee (q \wedge r))$ and $\neg(\neg p \leftrightarrow p)$.

Basic Rules:	
Constants:	AndTrue : $p \wedge True = p$ OrTrue : $p \vee True = True$ NotTrue : $\neg True = False$ AndFalse : $p \wedge False = False$ OrFalse : $p \vee False = p$ NotFalse : $\neg False = True$
Definitions:	ImplDef : $p \rightarrow q = \neg p \vee q$ EquivDef : $p \leftrightarrow q = (p \wedge q) \vee (\neg p \wedge \neg q)$
Negations:	DeMorganAnd : $\neg(p \wedge q) = \neg p \vee \neg q$ NotNot : $\neg\neg p = p$ DeMorganOr : $\neg(p \vee q) = \neg p \wedge \neg q$
Distribution:	AndOverOr : $p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$
Additional Rules:	
Tautologies:	OrTaut : $p \vee \neg p = Ty$ EquivTaut : $p \leftrightarrow p = True$ ImplTaut : $p \rightarrow p = True$
Contradictions:	AndContr : $p \wedge \neg p = False$ EquivContr : $p \leftrightarrow \neg p = False$

Fig. 2. Rules for logic expressions

The rules. Logic expressions form a boolean algebra, and hence a number of rules for logic expressions can be formulated. Figure 2 presents a small collection of basic rules and some tautologies and contradictions. All variables in these rules are meta-variables and range over arbitrary logic expressions. The rules are expressed as equivalences, but are only applied from left to right. For most rules we assume to have a commutative variant, for instance, $True \wedge p = p$ for rule AndTrue. With these implicit rules, we can bring every logic expression to disjunctive normal form.

Every serious exercise assistant for this domain has to be aware of a much richer set of rules. In particular, we have not given rules for commutativity and associativity, several plausible rules for implications and equivalences are omitted, and the list of tautologies and contradictions is far from complete.

Strategy 1: apply rules exhaustively. The first strategy applies the basic rules from Figure 2 exhaustively: we proceed as long as we can apply *some* rule *somewhere*, and we will end up with a logic expression in disjunctive normal form. This is a special property of the chosen collection of basic rules, and this is not the case for a rule set in general. The strategy is very liberal, and approves every sequence of rules.

Strategy 2: four steps. Strategy 1 accepts sequences that are not very attractive, and that no expert would ever consider. We give two examples:

$$\neg\neg(p \vee q) \xRightarrow{\textsc{DeMorganOr}} \neg(\neg p \wedge \neg q) \qquad True \vee (\neg\neg p) \xRightarrow{\textsc{NotNot}} True \vee p$$

In both cases, it is more appealing to select a different rule (NOTNOT and ORTRUE, respectively). We define a new strategy that proceeds in four steps, and such that the above sequences are not permitted.

- **Step 1:** Remove constants from the logic expression with the rules for "constants" (see Figure 2), supplemented with constant rules for implications and equivalences. Apply the rules *top-down*, that is, at the highest possible position in the abstract syntax tree. After this step, all occurrences of *True* and *False* are removed.
- **Step 2:** Use IMPLDEF and EQUIVDEF to rewrite implications and equivalences in the formula. Proceed in a *bottom-up* order.
- **Step 3:** Push negations inside the expression using the rules for "negations", and do so in a *top-down* fashion. After this step, all negations appear directly in front of a logic variable.
- **Step 4:** Use the distribution rule (ANDOVEROR) to move disjunctions to top-level. The order is irrelevant.

Strategy 3: tautologies and contradictions. Suppose that we want to extend Strategy 2, and use rules expressing tautologies and contradictions (for example, the additional rules in Figure 2). These rules introduce constants. Our last strategy is as follows:

- Follow the four steps of Strategy 2, however:
- *Whenever* possible, use the rules for tautologies and contradictions (*top-down*), *and*
- clean up the constants afterwards (step 1). Then continue with Strategy 2.

Buggy rules. In addition to the collection of rules and a strategy, we can formulate *buggy rules*. These rules capture mistakes that are often made, such as the following unsound variations on the two De Morgan rules:

$$\text{BUGGYDM1}: \neg(p \wedge q) \neq \neg p \wedge \neg q \qquad \text{BUGGYDM2}: \neg(p \vee q) \neq \neg p \vee \neg q$$

The advantage of formulating buggy rules is that specialized feedback can be presented if the system detects that such a rule was applied. Note that these rules should not appear in strategies, since that would invalidate the strategy.

The idea of formulating buggy rules can easily be extended to buggy strategies. Such a strategy helps to recognize common procedural mistakes, in which case we can report a detailed message.

Other domains in the exercise assistants

Besides exercises about rewriting logic expressions, our exercise assistants offer exercises about simplifying arithmetic expressions, in particular fractions. The datatype for arithmetic expressions has been defined in Section 2.2. We omit the rewriting rules and normalisation strategies for arithmetic expressions.

Other exercises in the exercise assistant deal with solving a system of linear equations, which is a value of type LinearSystem a, also introduced in Section 2.2. Since the datatype LinearSystem a also contains non-linear systems, we use a slightly more restricted variant of this datatype, which prohibits the use of non-liner expressions, in the code for the exercise assistant.

Finally, the exercise assistants supports several kinds of exercises on matrices, such as Gaussian elimination. We view a matrix as a list of rows of expressions. All rows have to be equally long.

```
newtype Matrix a = M [[Expr a]]
```

The rewrite rules on matrices consist, amongst others, of the elementary row operations for switching rows, multiplying a row, and adding rows.

Generic components

Adapting exercise assistants. For an e-learning tool to be successful, users have to be able to adapt the presentation of domains, the solving strategy of exercises, the kinds of exercises, and even the domains themselves. Since we have started working on the exercise assistants, we frequently get requests for instantiations of the exercise assistant on other domains. The code for the different domains is very similar, and we do not want to repeat implementing boilerplate code.

Furthermore, when improving our tools in a non-domain-specific way, we do not want to repeat the same improvements for all the tools we have built. To support the possibility to adapt the several components, to avoid code duplication and maximize code reuse, we want to develop generic programs that implement part of the functionality of exercise assistants.

Generic programs. We can distinguish several functions in the exercise assistants that can be implemented as generic programs. These functions deal with:

- rewriting expressions, including matching,
- determining whether or not two expressions are equal, determining the distance between two expressions, possibly after normalisation,
- generating exercises,
- traversing expressions,
- selecting within expressions,
- serialization.

We introduce each of these components below.

Rewriting. The basic concept behind the exercise assistant is that a student *rewrites* an expression, until a solution to the exercise is reached. So for each domain, we need to specify rewrite rules, and we have to apply rewrite rules to expressions. To specify rewrite rules, we have to add (meta-)variables to the domain. Logic expression may contain logic variables, but the variables used in the rewrite rules for logic expressions in Figure 2 are meta-variables. It follows that we either have to adapt the domain with an extra constructor for meta-variables, or that we can distinguish meta-variables from variables in the variable constructor in some way. Adding a variable to a datatype can be done generically, but it is not a generic *program*, but a generic (or type-indexed) *datatype* [Hinze et al., 2002]. In Section 9 we will show how to use associated datatypes [Chakravarty et al., 2005a] to implement such type-indexed datatypes.

The first step in rewriting an expression consists of matching the left-hand side of a rewrite rule with the current expression. The result of matching a left-hand side of a rewrite rule with an expression is either nothing, or a substitution, binding meta-variables to expressions. If matching succeeds, we replace the current term with the right-hand side of the rewrite rule, to which we apply the substitution.

Equality and minimum edit distance. Equality is used in all domains, amongst others to verify that a student has reached a solution. All domains also have a definition of equivalence, which implements semantic equality. The definition of equivalence varies for the different domains, but we expect it can be defined generically by applying the strategy to both terms, and using standard equality on the result. At the moment equivalence is a domain-specific function, but it often uses the fold on the domain.

When a student makes a rewrite step mistake, the exercise assistant reports this. We would like to replace this message by a more informative message, which also tries to explain a likely cause of the error. One way to find out a likely cause for the error is to rewrite the previous term in all possible, correct, ways, and to determine the difference between the terms obtained, and the term submitted by the student. The difference can be calculated using a generic program for determining the minimum edit distance between two trees.

Generating exercises. Each strategy determines a class of exercises which can be solved using the strategy. We want to generate random instances, within some constraints, of such exercises for students to practice. For example, for Gaussian elimination we want to generate random matrices, but we want to make sure that the solution is not too complicated. Preferably, the solution is a matrix consisting of entries that are relatively small integers. We use QuickCheck to generate exercises. The generators on different domains are very similar, and we want to have a generic program that abstracts from these instances.

Term traversals. The second strategy for rewriting a logic expression to DNF given in the previous subsection bottom-up eliminates all $\vee$'s that appear below top level, using the rule that says that $\wedge$ distributes over $\vee$, in its last step. To perform a rewrite rule bottom-up, or top-down, we first have to traverse to the lowest level in the term, try the rewrite rule, and then move up. It follows that we need functions for traversing terms. Here we use the usual strategic combinators for term transformations [Visser, 2005]. In the exercise assistants, all of these combinators are implemented in terms of the traversal function *once*, which takes a rewrite rule as argument and applies it to one of the children of the top-level constructor. This function is currently implemented using the Uniplate [Mitchell and Runciman, 2007] library.

Selections. One of the extensions to the exercise assistants that we are implementing is that students may select a subexpression, and ask for possible rewrite rules for that subexpression. Before we can present the possible rewrite rules, we want to check if a selected subexpression is *valid*. A subexpression is valid if it can be assigned a type. Checking whether or not a subexpression is valid or not can partially be built into the parser for the domain, and depends on the concrete syntax of the domain, but given the right parser combinators, we expect this functionality can be defined generically.

Serialization. We offer services that provide feedback given a term and a location in a strategy. These feedback services communicate with outside parties to exchange terms, strategies, and feedback messages. For communication purposes, we have to exchange information. At the moment, we exchange information via an XML format. This implies that we have to serialize our data to XML, and read XML into our services. Serializing and deserializing can partially be implemented as a generic program.

3.3 Implementing generic components

Study the Haskell libraries for generic programming presented in the following sections, pick your choice, analyse which generic components can be implemented using the library, and try to implement some of the above generic programs in this library. If you accomplish this in a reasonable amount of time, try to replace the domain-specific functionality in the exercise assistants by your generic programs.

4 Libraries for generic programming

Datatype-generic programming has been around for more than 10 years now. The first approaches to datatype-generic programming used programming language extensions to describe generic functions. Since the beginning of this decennium quite a number of libraries for generic programming in Haskell have been developed. The rationale for developing a library for generic programming instead of a language extension is that Haskell is powerful enough to support most generic programming concepts by means of a library. Furthermore, compared with a language extension, a library is much easier to ship, support, and maintain. A library might be accompanied by tools that depend on non-standard language extensions, for example for generating embedding-projection pairs, but the core is Haskell.

The libraries for generic programming have different characteristics. Recently, an extensive comparison of generic programming libraries (and their characteristics) has been performed [Rodriguez et al., 2008]. In these notes we will discuss three of those libraries: a Lightweight Implementation of Generics and Dynamics, Extensible and Modular Generics for the Masses, and Scrap Your Boilerplate.

We focus on these three ibraries for a number of reasons. First, we think these libraries are representative examples: one library explicitly passes a type representation as argument to a generic function, another relies on the type class mechanism, and the third is traversal- and combinator-based. Furthermore, all three have been used for a number of generic functions, and are relatively easy to use for parts of the lab exercise given in these notes. Finally, all three of them can express many generic functions; the Uniplate library [Mitchell and Runciman, 2007] is also representative and eays to use, but Scrap Your Boilerplate is more powerful.

The example libraries show different ways to implement the essential ingredients of generic programming libraries. Support for generic programming consists of three essential ingredients: a run-time type representation, a generic view on data, and support for overloading.

A *type-indexed function* (TIF) is a function that is defined on every type of a family of types. We say that the types in this family index the TIF, and we call the type family a universe. The run-time representation of types determines the universe on which we can pattern match in a type-indexed function. The larger this universe, the more types the function can be applied to.

A type-indexed function only works on the universe on which it is defined. If a new datatype is defined, the type-indexed function cannot be used on this new datatype. There are two ways to make it work on the new datatype. A non-generic extension of the universe of a TIF requires a type-specific, ad-hoc case for the new datatype. A generic extension (or a generic view) of a universe of a TIF requires to express the new datatype in terms of the universe of the TIF so that the TIF can be used on the new datatype without a type-specific case. A TIF combined with a generic extension is called a *generic function*.

An overloaded function is a function that analyses types to exhibit type-specific behavior. Type-indexed and generic functions are special cases of overloaded functions. Many generic functions even have type-specific behavior: lists are printed in a non-generic way by the generic pretty-printer defined by **deriving** *Show* in Haskell.

In the next sections we will see how to encode these basic ingredients in the three libraries we chose. For each library, we present its run-time type representation, the generic view on data and how overloading is achieved.

5 Lightweight Implementation of Generics and Dynamics

This section introduces generic programming in a simplified form of the datatype-generic programming library Lightweight Implementation of Generics and Dynamics (LIGD, [Cheney and Hinze, 2002]).

5.1 An example function

In polymorphic lambda calculus it is impossible to write a single parametrically polymorphic equality function that works on all datatypes [Wadler, 1989].

That is why the definition of equality in Haskell uses type classes, and ML uses equality types. The *Eq* type class provides the equality operator ==, which is overloaded for a family of types. To add a newly defined datatype to this family, the programmer defines an instance of equality for it. Thus, the programmer manually writes definitions of equality for every new datatype that is defined, as we did in Section 2. For equality, this process could be automated by using the type class deriving mechanism. However, this mechanism can only be used with a small number of type classes, because it is hardwired into the language, making it closed and impossible to extend or change by the programmer. In this subsection we show how equality is defined once and for all datatypes in LIGD.

The equality function *geq* takes three arguments: the two values of type a to compare, and a representation of the type of these values Rep a.

```
geq :: Rep a → a → a → Bool
```

Function *geq* is defined by pattern matching on the representation type Rep. The representation type contains a constructor for the unit type, which represents types with a single value, for the sum type, which represents a choice between two types, and for the product type, which represents a pair of types. We will introduce the type Rep and its constructors in the following subsection. Function *geq* is an example of a TIF.

```
geq (RUnit)       Unit          Unit          = True
geq (RInt   )     i             j             = i == j
geq (RChar)       c             d             = c == d
geq (RSum ra rb)  (Inl a1)      (Inl a2)      = geq ra a1 a2
geq (RSum ra rb)  (Inr b1)      (Inr b2)      = geq rb b1 b2
geq (RSum ra rb)  _             _             = False
geq (RProd ra rb) (Prod a1 b1)  (Prod a2 b2)  = geq ra a1 a2 ∧ geq rb b1 b2
```

Note that the run-time type-representation argument of function *geq* of type Rep a represents the type of the following two arguments, namely a. The type Unit, represented by *RUnit*, contains a single value *Unit*. Since there is only a single value of type Unit, two values of the type are always equal. For the Int type and the Char type we use the equality functions that are available for these types. For the Sum type we first check if the two values have the same outermost constructor (*Inl* or *Inr*). If so, we recursively compare their arguments, using the type representation of the type of those arguments. For the product type Prod we pairwise compare the components of the product, using the representation types of the types of these components. Function *geq* is type-safe, in the sense that the the two arguments that we want to compare have the same type, given by the run-time type representation Rep a. If an argument would have another type, Haskell's type-checker would complain at compile-time.

5.2 Run-time type representation

In LIGD, the first argument of a type-indexed function is a run-time *type representation*, which describes the type of the the arguments, results, or both arguments and results, of the function. The type representation need not appear as the first argument, but this is standard practice. In the variant of LIGD we give in these notes, types are represented by a generalized algebraic data type. Using a GADT has the advantage that case analysis on types can be implemented by pattern matching, a familiar construct to functional programmers. The GADT represents the types of the universe consisting of units, sums and products, together with basic types such as integers and characters. Of course, there are many more basic types, such as floats, doubles, etc, but we only include Int and Char.

```
data Unit       = Unit
data Sum a b  = Inl a | Inr b
data Prod a b = Prod a b
```

Unit is an example of a type with a single constructor with no arguments. The Sum datatype is equal to the datatype Either in Haskell's prelude. The Prod equals the pair (tuple) type in Haskell. We have chosen to use new datatypes to represent type representations at run-time, to not mix the world of type representations and other types. The GADT Rep uses these datatypes to represent the structure of types.

```
data Rep t where
  RUnit ::                          Rep Unit
  RInt    ::                          Rep Int
  RChar ::                          Rep Char
  RSum  :: Rep a → Rep b → Rep (Sum a b)
  RProd :: Rep a → Rep b → Rep (Prod a b)
```

The original LIGD was developed before GADTs had been added to Haskell and used an existentially quantified datatype instead. GADTs make it easier to define these structure types.

5.3 Going generic: universe extension

If we define a datatype, how can we use our type-indexed function on this new datatype? In LIGD (and most other generic programming libraries), the introduction of a new datatype does not require redefinition or extension of all existing generic functions. We merely need to describe the new datatype to the library, and all existing and future generic functions will be able to handle it.

In LIGD, the structure of a datatype b is represented by the following Rep constructor.

```
RType :: Rep c → EP b c → Rep b
```

The type c is the *structure representation* type of b, where c is a type isomorphic to b. The isomorphism is witnessed by an embedding projection pair, which is a pair of functions that convert b values to c values and back.

```
data EP b c = EP{from :: (b → c), to :: (c → b)}
```

In LIGD, constructors are represented by nested sum types and constructor arguments are represented by nested product types. The structure representation type for lists is Sum Unit (Prod a (List a)), and the embedding projection for lists is as follows:

```
from_List :: List a → Sum Unit (Prod a (List a))
from_List Nil              = Inl Unit
from_List (Cons a as)      = Inr (Prod a as)
to_List :: Sum Unit (Prod a (List a)) → List a
to_List (Inl Unit)         = Nil
to_List (Inr (Prod a as))  = Cons a as
```

To extend the universe to lists, we write a type representation using *RType*:

```
r_List :: Rep a → Rep (List a)
r_List r_a = RType (RSum RUnit (RProd r_a (r_List r_a)))
                   (EP from_List to_List)
```

Note that the components of the pair are not embedded in the universe. The reason for this is that LIGD does not model recursion explicitly.

We defined a type-indexed function for equality above. This definition still misses a case to handle datatypes that are represented by *RType*. The definition of this case is given below. It takes two values, transforms them to their structure representations and recursively applies equality.

```
geq (RType r_a ep) t1 t2 = geq r_a (from ep t1) (from ep t2)
```

Adding this line to the definition of *geq* turns it into a generic function.

In summary, there are two ways to extend a universe to a type T. A non-generic extension requires type-specific, ad-hoc cases for T in type-indexed functions, and a generic-extension requires a structure representation of T but no additional function cases. This is the feature that distinguishes type-indexed functions and generic functions. The latter include a case for *RType*, which allows them to exploit the structure of a datatype in order to apply generic uniform behaviour to values of that datatype; while the former do not have a case for *RType*, and therefore rely exclusively on non-generic extension.

In LIGD, sums, products and units are used to represent the structure of a datatype. Certainly other choices are possible. For example, PolyLib includes the datatype Fix in the universe, in order to represent the recursive structure of datatypes. We refer to these representation choices as *generic views* [Holdermans et al., 2006]. Informally, a view consists of base (or view) types for the universe (for example Sum and Prod) and a convention to represent structure, for example, representing constructors by nested sums. Usually the choice of a view will have an impact on the expressiveness of a library, that is, which generic functions definitions are supported and what are the set of datatypes on which generic extension is possible.

Exercise 1. Give the representation of the datatypes Tree and Forest in terms of Rep.

5.4 Support for overloading

Function *geq* can be viewed as an implementation of **deriving** *Eq* in Haskell itself. Similarly, we can define define functions that implement the methods of the other classes that can be derived in Haskell: *Show*, *Read*, *Ord*, *Enum*, and *Bounded*. We will show a definition of a generic show function. Besides illustrating generic programming in the library in general, implementing a generic show function illustrates how a library deals with constructor names, and how a library deals with ad-hoc cases for particular datatypes. For example, we do not want a generic show function to show a string "abc" as a list with explicit occurrences of constructor names: *Cons* 'a' (*Cons* 'b' (*Cons* 'c' *Nil*)).

The type representations as discussed in the previous section do not contain any information about constructors, and hence it is impossible to define a generic show function using this representation. To deal with constructor names, we add an extra constructor to the structure type representation type.

```
RCon :: String → Rep a → Rep a
```

Using this extra constructor of Rep, the representation of the datatype List a becomes:

```
r_List :: Rep a → Rep (List a)
r_List r_a = RType (RSum (RCon "Nil" RUnit)
                         (RCon "Cons" (RProd r_a (r_List r_a))))
                   (EP from_List to_List)
```

Here is a first attempt at defining a generic show function:

```
gshow :: Rep t → t → ShowS
gshow RInt            t          = shows t
gshow RChar           t          = shows t
gshow RUnit           t          = showString ""
gshow (RSum r_a r_b)  (Inl a)    = gshow r_a a
gshow (RSum r_a r_b)  (Inr b)    = gshow r_b b
gshow (RProd r_a r_b) (Prod a b) = gshow r_a a . showString " " . gshow r_b b
gshow (RType r_a ep)  t          = gshow r_a (from ep t)
gshow (RCon s RUnit)  t          = showString s
gshow (RCon s r_a)    t          = showChar '('
                              . showString s . showChar ' ' . gshow r_a t
                              . showChar ')'
```

This definition works for all datatypes, but it shows strings and Haskell's lists in a uniform way, using constructor names. We want to extend this function so that it behaves in a special, non-generic way for the type of strings, and Haskell's list datatype [a].

For each type for which we want a generic function to behave in a special, non-generic, way, we have to extend the representation type. For example, to solve the problem with showing strings and lists, we add the following constructors to the representation type Rep:

```
RString ::          Rep String
RList   :: Rep a → Rep (List a)
```

Now we can add the following lines to the generic show function to obtain type-specific behavior for the type String and [a].

```
gshow (RList r_a) Nil         = showString "[]"
gshow (RList r_a) (Cons x xs) = gshow r_a x . showChar ':' . gshow (RList r_a) xs
gshow RString     s           = showString s
```

The resulting function does not implement all details of **deriving** *Show*, but it does provide the core functionality.

Note that we had to adapt the type representation type Rep to obtain type-specific behavior in the *gshow* function. It is undesirable to adapt a library for the purpose of obtaining special behavior of a single generic function on a particular datatype. Unfortunately, this is unavoidable in the LIGD library: for any generic function that needs special behavior on a particular datatype, we have to extend the type representation with that datatype. This implies that many users will construct their own variant of the LIGD library, making both the library and the generic functions written using it less portable and reusable. Löh and Hinze [2006] show how to add *open datatypes* to Haskell. A datatype is open if it can be extended in a different module. In a language with open datatypes, the above problem with LIGD would disappear.

5.5 Generic functions in LIGD

This section introduces some more generic functions in LIGD, in particular some functions for which we need different type representations. We start with a simple example of a generic program.

Empty

With every type we can associate an empty value. For example, the empty value of type Int is 0, and the empty value of type List a is [ ]. Function *gempty* is a generic function that returns the empty value for an arbitrary type. An interesting aspect of this function is that it *constructs* a value of a type, instead of consuming a value, as in *geq* and *gshow*.

```
gempty :: Rep a → a
gempty RUnit          = Unit
gempty RInt           = 0
gempty RChar          = '\NUL'
gempty (RSum r_a r_b)  = Inl (gempty r_a)
gempty (RProd r_a r_b) = Prod (gempty r_a) (gempty r_b)
gempty (RType r_a ep)  = to ep (gempty r_a)
gempty (RCon s r_a)    = gempty r_a
```

Exercise 2. Another generic function that constructs values of a datatype is the function *genum* :: Rep a → [a], which generates all values of a type. Many datatypes have infinitely many values, so it is important that function *genum* enumerates values fairly. Implement *genum* in LIGD.

Flatten

Many datatypes can be considered "container" datatypes: datatypes used to store and structure values. Examples are the datatypes List a, Tree a, Forest a, Expr a, all introduced in Section 2.2. One of the few datatypes introduced in Section 2.2 that is not a container datatype is the datatype $\mathsf{Logic_s}$. A common function on a container datatype is the function flatten, which takes a value of the datatype, and returns a list containing all values that it contains. For example, on the datatype Tree a function flatten would have type Tree a → [a]. This subsection explains how the generic flatten function *gflatten* is defined in LIGD.

To implement function *gflatten*, we have to solve a number of problems. A first problem is to describe its type. A first, incorrect, attempt would be the following:

```
gflatten :: Rep f → f a → [a]
```

where f abstracts over types of kind $\star \to \star$. Since Rep has kind $\star \to \star$, this gives a kind error. Replacing Rep f by Rep (f a) would solve the kinding problem, but introduces another: how do we split the representation of a container datatype into a representation for f and a representation for a? Type application is implicit in our type representation. We solve this problem by adapting the structure representation type for LIGD with an extra case *RVar1* which is used to define special functionality for occurrences of the type argument in constructors.

```
data Rep1 f a where
  RUnit1 ::                    Rep1 f Unit
```

$$\begin{array}{ll} \mathit{RSum1} & :: \mathsf{Rep1\ f\ a} \to \mathsf{Rep1\ f\ b} \to \mathsf{Rep1\ f\ (Sum\ a\ b)} \\ \ldots & \\ \mathit{RVar1} & :: \mathsf{f\ a} \to \qquad\qquad\qquad\quad \mathsf{Rep1\ f\ a} \end{array}$$

Except for its type, the representation of lists using this new representation type does not change.

$$\begin{array}{l} r_{List,1} :: \mathsf{Rep1\ f\ a} \to \mathsf{Rep1\ f\ (List\ a)} \\ r_{List,1}\ r_a = \mathit{RType1}\ (\mathit{RSum1}\ (\mathit{RCon1}\ \texttt{"Nil"}\ \mathit{RUnit1}) \\ \qquad\qquad\qquad\qquad\qquad (\mathit{RCon1}\ \texttt{"Cons"}\ (\mathit{RProd1}\ r_a\ (r_{List,1}\ r_a)))) \\ \qquad\qquad\qquad (\mathit{EP}\ \mathit{from}_{List}\ \mathit{to}_{List}) \end{array}$$

To obtain an instance of the generic function *gflatten* on the datatype List a we write:

$$\begin{array}{l} \mathit{flattenList} :: \mathsf{List\ a} \to [\mathsf{a}] \\ \mathit{flattenList} = \mathit{gflattenT}\ r_{List,1} \end{array}$$

To specify the type of *gflattenT* we introduce a **newtype** GFlatten, together with an abbreviation for a representation type Rep1 in which the action for type variables has been instantiated with GFlatten:

$$\begin{array}{lll} \mathbf{newtype} & \mathsf{GFlatten\ b\ a} & = \mathit{GFlatten}\{\mathit{gFlatten} :: \mathsf{a} \to \mathsf{b}\} \\ \mathbf{type} & \mathsf{a} :\Rightarrow \mathsf{b} & = \mathsf{Rep1\ (GFlatten\ b)\ a} \end{array}$$

which is used in specifying what to do with an occurrence of a value of the type variable. Function *mkFlatten* specifies what to do with such an occurrence: store it in a list by means of the function (:[ ]).

$$\begin{array}{l} \mathit{mkFlatten} :: \mathsf{a} :\Rightarrow [\mathsf{a}] \\ \mathit{mkFlatten} = \mathit{RVar1}\ (\mathit{GFlatten}\ (:[\,])) \end{array}$$

Function *gflatten* now takes a function which transforms a representation of the action on variables $\mathsf{a} :\Rightarrow [\mathsf{a}]$ to a representation of the argument datatype $\mathsf{b} :\Rightarrow [\mathsf{a}]$, and a value of the argument datatype b, and returns the list of occurrences of the values of the type variable [a].

$$\begin{array}{l} \mathit{gflattenT} :: ((\mathsf{a} :\Rightarrow [\mathsf{a}]) \to (\mathsf{b} :\Rightarrow [\mathsf{a}])) \to \mathsf{b} \to [\mathsf{a}] \\ \mathit{gflattenT}\ \mathit{repTransform} = \mathit{gflatten}\ (\mathit{repTransform}\ \mathit{mkFlatten}) \end{array}$$

where *gflatten* is the generic function that does the pattern matching on the structure representation type.

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{gflatten} :: \mathsf{b} :\Rightarrow [\mathsf{a}] \to \mathsf{b} \to [\mathsf{a}]} \\ \mathit{gflatten}\ \mathit{RUnit1} & \mathit{Unit} & = [\,] \\ \mathit{gflatten}\ (\mathit{RSum1}\ r_a\ r_b) & (\mathit{Inl}\ a) & = \mathit{gflatten}\ r_a\ a \\ \mathit{gflatten}\ (\mathit{RSum1}\ r_a\ r_b) & (\mathit{Inr}\ b) & = \mathit{gflatten}\ r_b\ b \end{array}$$

```
gflatten (RProd1 r_a r_b) (Prod a b) = gflatten r_a a ++ gflatten r_b b
gflatten RInt1            i          = []
gflatten RChar1           c          = []
gflatten (RCon1 _ r_a)    x          = gflatten r_a x
gflatten (RType1 r_a ep)  x          = gflatten r_a (from ep x)
gflatten (RVar1 f)        x          = gFlatten f x
```

Exercise 3. Many generic functions follow the pattern of the generic *flatten* function. Examples are a function that sums all the integers in a value of a datatype, and a function that takes the boolean or of all boolean values in a value of a datatype. We implement this pattern by function *gcrush*. Section 6 defines *crushRight*, which is a variant from the *gcrush* function.

Function *gcrush* abstracts from the functionality at occurrences of the type variable, and from the base case [ ] and the binary case ++ in the definition of flatten. Its type is as follows.

```
newtype Crush b a = Crush{crush :: a → b}
gcrush :: Rep1 (Crush b) a → (b → b → b) → b → a → b
```

Define function *gcrush*.

Instantiate *gcrush* with the addition operator, 0, and a value of a datatype containing integers to obtain a generic *sum* function, to test if your function implements the desired behavior.

Generalised map

A well-known generic function is the generic *gmap* function, which is a generalisation of the *map* function available on lists in Haskell. The *map* function in Haskell takes a function and a list as argument, and applies the function to all elements in the list. The generic *gmap* function takes a function of type a → b and a value of a datatype containing a's, and applies the function to all the a's in the value. Function *gmap* can be viewed as the implementation of **deriving** for the *Functor* type class in Haskell. Just as function *gflatten*, the generic map function needs to know where the occurrences of the type-argument of the datatype appear in a constructor. This implies that we have to be able to abstract over type constructors. If we use the representation type Rep1 to implement the generic map function, we can only define a map function in which the argument function returns a value of either a type that only depends on a, or a constant type. This is because the constructor *RVar1* has type f a → Rep1 f a. We use the *RVar1* constructor to specify the behavior at occurrences of values of the type variable, and to specify a function of type a → b there, we need an extra type variable. Since we want to be able to change the type of the values occurring in a datatype using the generic map function, we change the representation datatype once more.

```
data Rep2 f a b where
  RUnit2 ::                             Rep2 f Unit Unit
  RSum2 :: Rep2 f a b → Rep2 f c d → Rep2 f (Sum a c) (Sum b d)
  . . .
  RVar2  :: f a b →                     Rep2 f a b
```

The only difference with the representation type Rep1 is the occurrence of the extra type variable b in the representation type and all constructors. The representation of lists using this new representation type hardly changes: we get an extra embedding-projection pair for the extra type variable.

```
r_List,2 r_a = RType2 (RSum2 (RCon2 "Nil" RUnit2)
                             (RCon2 "Cons" (RProd2 r_a (r_List,2 r_a))))
                      (EP from_List to_List)
                      (EP from_List to_List)
```

Using $r_{List,2}$, function *map* on lists is obtained as follows:

```
mapList :: (a → b) → List a → List b
mapList = gmap r_List,2
```

where *gmap* is defined below. To define function *gmap* we follow the same approach as for defining *gflatten*. First we introduce the type GMap for functions, and the type a :→ b for a mapping in the representation type

```
newtype GMap a b = GMap{gMap :: a → b}
type (a :→ b)     = Rep2 GMap a b
```

Function *mkMap* specifies what to do with an occurrence of a value of the type variable, namely applying the argument function.

```
mkMap :: (a → b) → (a :→ b)
mkMap f = RVar2 (GMap f)
```

Function *gmap* now takes a function which transforms a representation of the action on variables *a* :→ *b*, a function, and a value of a datatype, and returns a value of the same shape, in which the argument function has been applied to all values appearing at the type variable position in the constructors.

```
gmap :: ((a :→ b) → (c :→ d)) → (a → b) → (c → d)
gmap repTransform f = gmapG (repTransform (mkMap f))
```

where *gmapG* is the generic function that does the pattern matching on the structure representation type.

Note that if we pass the representation transformer $r_{List,2}$ to *gmap*, the type of its first argument is unified to a :→ b:

```
r_List,2 :: (a :→ b) → (List a :→ List b)
```

Applying *gmap* to $r_{List,2}$ results in a function of the desired type $(\mathsf{a} \to \mathsf{b}) \to \mathsf{List\ a} \to \mathsf{List\ b}$.

```
gmapG :: (a :→ b) → a → b
gmapG RUnit2               Unit       = Unit
gmapG (RSum2 r_a r_b)      (Inl a)    = Inl (gmapG r_a a)
gmapG (RSum2 r_a r_b)      (Inr b)    = Inr (gmapG r_b b)
gmapG (RProd2 r_a r_b)     (Prod a b) = Prod (gmapG r_a a) (gmapG r_b b)
gmapG RInt2                i          = i
gmapG RChar2               c          = c
gmapG (RCon2 _nm r_a)      x          = gmapG r_a x
gmapG (RType2 r_a ep1 ep2) x          = (to ep2 . gmapG r_a . from ep1) x
gmapG (RVar2 f)            x          = gMap f x
```

Since the last two generic functions we have introduced both require a new structure representation type, one might wonder if this would happen for many generic functions. As far as we are aware, these extensions stop at level 3. We could use the datatype Rep3 for all of our generic functions, but that would introduce many type variables which are never used, so we prefer to use the representation type that is most suitable for the generic function at hand.

Exercise 4. Define the generalised version of function *zipWith* :: $(\mathsf{a} \to \mathsf{b} \to \mathsf{c}) \to [\mathsf{a}] \to [\mathsf{b}] \to [\mathsf{c}]$ in LIGD. You may adapt the structure representation type for this purpose.

6 Generics for the Masses

This section introduces generic programming as described by "Generics for the Masses" [Hinze, 2006], specifically the extensible and modular variation ("EMGM") [d. S. Oliveira et al., 2006].

6.1 An example function

Defining a generic function in the EMGM library involves several steps. First, we declare the type signature of a function in a **newtype** declaration.

```
newtype Geq a = Geq{geq' :: a → a → Bool}
```

Since *geq'* is a method in a record, we need to pass a record when using it, so the actual type of *geq'* is as follows:

```
geq' :: Geq a → a → a → Bool
```

An element of Geq a contains an instance of the equality function for a representation of a type a. In fact, we can use this function in a way very similar to *geq* in Section 5:

$$geq' \ (prod \ char \ int) \ (Prod \ \texttt{'Q'} \ 42) \ (Prod \ \texttt{'Q'} \ 42) \rightsquigarrow True$$

By passing the representation to the type-indexed function *geq′*, we specialize it for a given datatype. We could imagine the following functions as dispatchers for each specialization.

$$\begin{array}{llll}
geq_{\mathbb{1}} & Unit & Unit & = True \\
geq_{int} & i & j & = i == j \\
geq_{char} & c & d & = c == d \\
geq_{+} \ r_a \ r_b \ (Inl \ a_1) & & (Inl \ a_2) & = geq' \ r_a \ a_1 \ a_2 \\
geq_{+} \ r_a \ r_b \ (Inr \ b_1) & & (Inr \ b_2) & = geq' \ r_b \ b_1 \ b_2 \\
geq_{+} \ \_ \ \_ \ \_ & & \_ & = False \\
geq_{\times} \ r_a \ r_b \ (Prod \ a_1 \ b_1) & & (Prod \ a_2 \ b_2) & = geq' \ r_a \ a_1 \ a_2 \wedge geq' \ r_b \ b_1 \ b_2
\end{array}$$

We can read this in the same fashion as a type-indexed function in LIGD. Indeed, if we compare the collection of functions here with *geq* from Section 5, we notice a high degree of similarity. But instead of one function, we have three, each corresponding to the Unit, Sum, or Prod datatype. Another major difference with LIGD is that the the type representation parameters are explicit and not embedded in the Rep datatype. Specifically, each function takes the appropriate number of representations according to its arity; thus, $geq_{\mathbb{1}}$ has none and geq_{+} and $geq_{\times}$ each have two.

These functions are only part of the story, however. Notice that geq_{+} and $geq_{\times}$ each call the generic function *geq′*. We need to tie the recursive knot, so that *geq′* will now refer to each of these functions. We do this by creating an instance declaration:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance} \ Generic \ \mathsf{Geq} \ \textbf{where}} \\
\quad unit & & = Geq \ geq_{\mathbb{1}} \\
\quad int & & = Geq \ geq_{int} \\
\quad char & & = Geq \ geq_{char} \\
\quad plus \ r_a \ r_b & & = Geq \ (geq_{+} \ r_a \ r_b) \\
\quad prod \ r_a \ r_b & & = Geq \ (geq_{\times} \ r_a \ r_b)
\end{array}$$

The type class *Generic* has member functions corresponding to structure types. Each method binding defines the instance of the generic function for the associated type. Our dispatcher functions are now used to construct elements of Geq. Although the EMGM approach uses method overriding instead of the pattern matching of LIGD, it still effectively provides a case analysis on types.

We now have all of the necessary parts to use the type-index function *geq′*; however, we should not need to provide an explicit representation every time. So, we introduce a convenient wrapper that determines which type representation we need.

$$\begin{array}{l}
geq :: Rep \ \mathsf{a} \Rightarrow \mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Bool} \\
geq = geq' \ rep
\end{array}$$

We discuss the *Rep* type class in more detail in the next section. For now, we may think of the class context here simply as a requirement that type a be representable. The value *rep* is witness to that representation.

6.2 Run-time type representation

In contrast with LIGD's use of a generalized abstract datatype, EMGM makes extensive use of type classes for its run-time type representation. The primary classes are *Generic* and *Rep*, though others may be used to extend the basic concepts of EMGM as we will see later in Section 6.3.

The class *Generic* serves as the type case for generic functions.

```
class Generic g where
  unit :: g Unit
  int  :: g Int
  char :: g Char
  plus :: g a → g b → g (Sum a b)
  prod :: g a → g b → g (Prod a b)
```

The class is parametrised by the type constructor g with the intention that it represent the type of a generic function. As a result, the generic function has to be embedded in a datatype as is the function *geq*′.

Each method of the class represents a case of the generic function. The function supports the same universe of types as LIGD (e.g. Unit, Sum, Prod, and primitive types). Also like LIGD, the structural induction is implemented through recursive calls, but unlike LIGD, these are polymorphically recursive. Thus, each call to the generic function may have a different type.

The generic function as we have defined it to this point is a deconstructor for the type g. As such, it requires an instance of g, the type representation. In order to alleviate this requirement, we use another type class:

```
class Rep a where
  rep :: Generic g ⇒ g a
```

The class *Rep* allows us to replace any instance of the type g with *rep*. This is a simple but powerful concept. We use the type system to find the right representation for us. This representation is again built inductively using the methods of *Generic*:

```
instance Rep Unit where
  rep = unit
instance Rep Int where
  rep = int
instance Rep Char where
  rep = char
instance (Rep a, Rep b) ⇒ Rep (Sum a b) where
```

```
  rep = plus rep rep
instance (Rep a, Rep b) ⇒ Rep (Prod a b) where
  rep = prod rep rep
```

As simple as these instances of *Rep* are, they handle an important duty. In the function *geq*, we use *rep* to implicitly instantiate the structure of the arguments. Now, we may apply *geq* with the same ease of use as any ad-hoc polymorphic function, but it is actually datatype-generic.

6.3 Going generic: universe extension

Much like in LIGD, we need to extend our universe to include any new datatypes that we create. Just as with LIGD, we extend our type-indexed functions with a case to support arbitrary datatypes.

```
class Generic g where
  ...
  datatype :: EP b a → g a → g b
```

Our *datatype* function reuses the embedding-projection pair datatype EP mentioned earlier to witness the isomorphism between the structure representation and the datatype. Note the similarity with the *RType* constructor from LIGD.

To demonstrate the use of *datatype*, we will once again show how the List datatype may be represented generically. As mentioned before, we use the same universe as LIGD, so we can make use of the same pair of functions, $from_{List}$ and to_{List}, in the embedding projection for lists. Using this pair and an encoding of the list structure at the value level, we define a representation of lists:

```
r_List :: Generic g ⇒ g a → g (List a)
r_List r_a = datatype (EP from_List to_List) (plus unit (prod r_a (r_List r_a)))
```

It is now straightforward to apply a generic function to a list. To make it convenient, we create a new instance of *Rep* for List a with the constraint that the contained type a must also be representable:

```
instance Rep a ⇒ Rep (List a) where
  rep = r_List rep
```

At last, we can transform our type-indexed equality function into a true generic function. For this, we need to add another case for arbitrary datatypes.

```
geq_dt ep r_a a_1 a_2 = geq' r_a (from ep a_1) (from ep a_2)
```

```
instance Generic Geq where
  ...
  datatype ep r_a = Geq (geq_dt ep r_a)
```

The so-call "dispatcher" function geq_{dt} accepts any datatype for which an embedding-projection pair has been defined. It is very similar to the arm of the LIGD function *geq* for *RType*. The instance definition for *datatype* means that the recursive knot has been tied for *geq* which is now a generic function.

Exercise 5. Give representations for the datatypes $\mathsf{Logic}_\mathsf{L}$ and $\mathsf{Logic}_\mathsf{F}$ from Section 2.2. Follow the definition of r_{List}, and include the embedding-projection pairs. You may need to define representations of other datatypes in the process. Test your results using *geq*.

6.4 Support for overloading

In this section, we provide a definition for another generic function, while simultaneously demonstrating how the EMGM library can support constructor names and ad-hoc cases. Just as with LIGD, we illustrate support for overloading with *gshow* along with a non-generic definition for lists.

Currently, we cannot access information such as constructor names in the definition of a generic function. For this purpose, we add another "case" to our generic function declaration.

```
type Name = String
```

```
class Generic g where
  . . .
  constr :: Name → g a → g a
```

We use this class method to label other type components with a name. As an example of using *constr*, we extend the list type representation:

```
r_List :: Generic g ⇒ g a → g (List a)
r_List r_a = datatype (EP from_List to_List) (plus (constr "Nil" unit)
                                                   (constr "Cons" (prod r_a (r_List r_a))))
```

Using the capability to display constructor names, we can write a simplified generic show function:

```
newtype Gshow a = Gshow{gshow' :: a → ShowS}

gshow_1              Unit       = showString ""
gshow_int            i          = shows i
gshow_char           c          = showChar c
gshow_+        r_a r_b (Inl a)  = gshow' r_a a
gshow_+        r_a r_b (Inr b)  = gshow' r_b b
gshow_×        r_a r_b (Prod a b) = gshow' r_a a . showChar ' ' . gshow' r_b b
gshow_dt    ep r_a     a        = gshow' r_a (from ep a)
```

```
gshow_constr s  r_a   a          = showChar '(' .
                                   showString s . showChar ' ' . gshow' r_a a .
                                   showChar ')'
```

```
instance Generic Gshow where
  unit            = Gshow gshow_1
  int             = Gshow gshow_int
  char            = Gshow gshow_char
  plus     r_a r_b = Gshow (gshow_+ r_a r_b)
  prod     r_a r_b = Gshow (gshow_× r_a r_b)
  datatype ep r_a  = Gshow (gshow_dt ep r_a)
  constr   s  r_a  = Gshow (gshow_constr s r_a)
```

```
gshow :: Rep a ⇒ a → ShowS
gshow = gshow' rep
```

Applying this function to a list of integers gives us the expected result:

$(gshow\ (Cons\ 5\ (Cons\ 3\ Nil)))$ `""` $\rightsquigarrow$ `"(Cons 5 (Cons 3 (Nil )))"`

As mentioned in Section 5, we would prefer to see this list appear as it natively does in Haskell: `"[5,3]"`. We would also rather see the String type appear as a row of characters instead of a list with constructor names. To this end, just as we added constructors to the Rep GADT, we can add methods to the *Generic* type class.

```
class Generic g where
  ...
  list   :: g a → g (List a)
  string :: g String
```

It is then straightforward to define these new cases for the generic show function.

```
gshow_list r_a Nil         = showString "[]"
gshow_list r_a (Cons a as) = gshow' r_a a . showChar ':' . gshow' (list r_a) as
gshow_string s             = showString s
```

```
instance Generic Gshow where
  ...
  list r_a = Gshow (gshow_list r_a)
  string   = Gshow gshow_string
```

Our last step, as expected, is to make these types representable. We replace the previous instance of *Rep* for List a with one using the *list* method, and we add a new instance for String.

```
instance Rep a ⇒ Rep (List a) where
  rep = list rep
instance Rep String where
  rep = string
```

Now, when we revisit our example application of *gshow* to a list, we receive a more concise response: `"5:3:[]"`. We can further adapt the definition of $gshow_{list}$ to obtain $[5, 3]$ instead.

We have extended the representation for generic functions to support ad-hoc list and string cases. In order to do that, we modified the primary element: the *Generic* type class. This approach fails when the module containing *Generic* is distributed as a library. Users of the library either choose a different route for ad-hoc cases, or they duplicate the functionality. Fortunately, there is a solution to making a Generics for the Masses library extensible and modular.

6.5 Making generic functions extensible

We know that modifying the type class *Generic* should be off limits, so we might consider a hierarchy of classes. *Generic* would be the base class of all generic functions, and users would introduce subclasses for ad-hoc cases. Let us revisit the list example.

```
class Generic g ⇒ GenericList g where
  list :: g a → g (List a)
  list = r_List
```

This declaration introduces the subclass *GenericList* encoding a list representation. The default value of *list* is the same value that we determined previously, but it can be overridden in an instance declaration. For the ad-hoc case of the generic show function, we would use an instance with the same implementation as before:

```
instance GenericList Gshow where
  list r_a = Gshow (gshow_list r_a)
```

We have regained ground on our previous implementation of an ad-hoc case, yet at the same time, we have lost some. We can apply our generic function to a type representation and a value (e.g. $(gshow' \; (list \; int) \; (Cons \; 3 \; Nil))$ `""`), and it will evaluate as expected. However, we can no longer use the same means of dispatching the appropriate representation with ad-hoc cases. What happens if we attempt to write the following instance of *Rep*?

```
instance Rep a ⇒ Rep (List a) where
  rep = list rep
```

GHC gives us this error:

```
Could not deduce (GenericList g)
  from the context (Rep (List a), Rep a, Generic g)
  arising from a use of ‘list’ at ...
Possible fix:
  add (GenericList g) to the context of
    the type signature for ‘rep’ ...
```

We certainly do not want to follow GHC’s advise. Recall that the method *rep* of class *Rep* has the type $(Generic\ g, Rep\ a) \Rightarrow g\ a$. By adding *GenericList* g to its context, we would force all generic functions to support both *Generic* and *GenericList*, thereby ruling out any modularity. In order to use *Rep* as it is currently defined, we must use a type g that is an instance of *Generic*; instances of any subclasses are not valid.

Let us instead abstract over the type constructor g. We subsequently redefine *Rep* as a type class with two parameters.

```
class Rep g a where
  rep :: g a
```

A small matter of migrating the parametrisation of the type constructor to the class level and lifting the restriction of the *Generic* context opens the universe to a new round of representations. We now re-introduce the representative instances.

```
instance Generic g ⇒ Rep g Unit where
  rep = unit
instance Generic g ⇒ Rep g Int where
  rep = int
instance Generic g ⇒ Rep g Char where
  rep = char
instance (Generic g, Rep g a, Rep g b) ⇒ Rep g (Sum a b) where
  rep = plus rep rep
instance (Generic g, Rep g a, Rep g b) ⇒ Rep g (Prod a b) where
  rep = prod rep rep
instance (GenericList g, Rep g a) ⇒ Rep g (List a) where
  rep = list rep
```

The organization here is very regular. Every instance handled by a method of *Generic* is constrained by *Generic* in its context. For the ad-hoc list instance, we use *GenericList* instead. This is the key to true extensibility and modularity in the EMGM library.

Now, we rewrite our generic show function to use the new dispatcher by specialising the type constructor argument g to Gshow.

```
gshow :: Rep Gshow a ⇒ a → ShowS
gshow = gshow′ rep
```

Exercise 6. The standard *compare* function returns the ordered relationship ("less than," "equal to," or "greater than") between two instances of some type a.

```
data Ordering = LT | EQ | GT
compare :: (Eq a, Ord a) ⇒ a → a → Ordering
```

This function can be implemented by hand, but more often it is generated by the compiler using **deriving** *Ord*. The latter uses the syntactic ordering of constructors to determine the relationship. For example, the datatype Ordering derives *Ord* and its constructors have the relationship $LT < EQ < GT$.

Implement an extensible generic version of *compare* that behaves like **deriving** *Ord* and works with representable types. It should have a type signature similar to the following:

```
gcompare :: a → a → Ordering
```

6.6 Generic functions in EMGM

In this section, we discuss the implementations of different sorts of generic functions. Some require exploring alternative strategies from the approach described so far.

Empty

A simple example of a generic producer is one that generates an "empty" value for every possible type. We write the generic empty function in EMGM as follows:

```
newtype Gempty a = Gempty{gempty' :: a}

instance Generic Gempty where
  unit             = Gempty Unit
  int              = Gempty 0
  char             = Gempty '\NUL'
  plus        r_a r_b = Gempty (Inl (gempty' r_a))
  prod        r_a r_b = Gempty (Prod (gempty' r_a) (gempty' r_b))
  datatype ep r_a     = Gempty (to ep (gempty' r_a))
  constr    s r_a     = Gempty (gempty' r_a)

gempty :: Rep Gempty a ⇒ a
gempty = gempty' rep
```

There are a few notable differences here from previous examples. First, notice the use of *gempty′* with the type Gempty → a. The generic function simply takes a representation and produces a value, no other arguments are provided. In order to generate a value with a concrete type, we would evaluate it like this:

$$gempty :: \mathsf{Sum\ Int\ Char} \leadsto Inl\ 0$$

Second, we use the *to* destructor of EP instead of *from* in the *datatype* method of *Generic*. This is the signature of a producer function.

Crush and flatten

So far in this section, we have dealt with generic functions abstracted over finite and fully applied types (with kind $\star$). We should also investigate how to deal with type constructors (kind $\star \to \star$). There are many such "container" datatypes that use parametric polymorphism to store values of various types with a common structure.

We explore the realm of type constructors with the implementation of a generic crush function and a generic flatten operation (the latter is implemented using the former). *Crush* is a fold-like operation over a datatype [Meertens, 1996]. We want to end up with a function similar to this:

$$crushRight :: (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{f\ a} \to \mathsf{b}$$

Function *crushRight* takes three arguments: a "combining" operator that joins a-values with b-values to create new b-values, a "zero" value, and a container f parameterized by a. *crushRight* (sometimes called *reduce*) is a generalization of the standard Haskell right-fold, *foldr*. In *foldr*, f is specialized to [ ].

We split the implementation of *crushRight* into components, and we begin with the type signature for the combining function.

$$\textbf{newtype}\ \mathsf{Crush\ b\ a} = Crush\{crushRight' :: \mathsf{a} \to \mathsf{b} \to \mathsf{b}\}$$

This function extracts the container's element (the a type) and combines it in some way with a partial result (the b type) to produce a final result. The implementation is fairly straightforward[4]:

$$
\begin{array}{llll}
crushRight_{\mathbb{1}} & \_ & e = e \\
crushRight_{int} & \_ & e = e \\
crushRight_{char} & \_ & e = e \\
crushRight_{+} & r_a\ r_b\ (Inl\ a) & e = crushRight'\ r_a\ a\ e \\
crushRight_{+} & r_a\ r_b\ (Inr\ b) & e = crushRight'\ r_b\ b\ e \\
crushRight_{\times} & r_a\ r_b\ (Prod\ a\ b) & e = crushRight'\ r_a\ a\ (crushRight'\ r_b\ b\ e) \\
crushRight_{dt} & ep\ r_a\quad a & e = crushRight'\ r_a\ (from\ ep\ a)\ e \\
crushRight_{constr} & s\quad r_a\quad a & e = crushRight'\ r_a\ a\ e
\end{array}
$$

[4] For brevity, we elide most of the instance declaration, because it is as expected.

```
instance Generic (Crush b) where
  unit = Crush crushRight_1
  . . .
```

Note that *crushRight′* is only applied to the parametrised structural types: Sum, Prod, and the datatype and constructor cases; it is not applied to the primitive types. This operator is only useful for combining the elements of a polymorphic datatype and not for acting on non-parametrised types.

We have successfully made it this far, but we now run into a problem. We need a representation for a type of kind $\star \to \star$. The type for *rep* is *Rep* g a ⇒ g a, and type a is the representation type and of kind $\star$. To expand *rep* to support functional types, we add a parameter to the function (in the same way the type is now parametrised) to get:

```
class FRep g f where
  frep :: g a → g (f a)
```

In the class *FRep* (functional representation), the representation type argument a of *Rep* is replaced by a constructor type f. This is exactly what we need for container types such as Tree a or List a. The *FRep* instance for List is similar to the one for *Rep* except that references to the type argument (and the corresponding function argument) are removed.

```
instance Generic g ⇒ FRep g List where
  frep = r_List
```

The pieces are becoming clearer, but it is not yet obvious how to bring them together. The reason is that we fibbed a bit earlier regarding the *crushRight′* function. In its type, a → b → b, the a actually refers to the *representation* type, not the container type a that we want for our final *crushRight*. As a result, we need a way to convert the combining operator from using a container datatype to using its representation. The ingredients are available directly:

```
frep . Crush :: (FRep (Crush b) f) ⇒ (a → b → b) → Crush b (f a)
```

With this, we can pass a proper combining operator and operate on the structure of some container type f.

The last two components to *crushRight*, a zero value and a container, follow simply from the definition of *crushRight′*. The development of our function is now complete.

```
crushRight :: FRep (Crush b) f ⇒ (a → b → b) → b → f a → b
crushRight f z x = crushRight′ (frep (Crush f)) x z
```

To demonstrate the use of *crushRight*, we use the same generic flattening function introduced in Section 5. Recall that flattening involves translating all elements of a structure into a list. The definition of *gflatten* only requires the combining operator, (:), and the zero value, [ ].

```
gflatten :: FRep (Crush [a]) f ⇒ f a → [a]
gflatten = crushRight (:) []
```

Exercise 7. Define two functions using *crushRight*:

1. *showElements* takes a container with showable elements and returns a string with the elements printed in a comma-delimited fashion.
2. *sumElements* takes a container with numeric elements and returns the numeric sum of all elements.

Generalised map

The standard library *map* function applies a function on all of the elements of a list. A generic map operation generalises this to any container datatype. A datatype-specific map function may be implemented as an instance of the standard class *Functor*. This approach is error-prone and not generic.

As described in Section 5, a *gmap* function gives us the ability to modify the elements of any container type. We aim for a function with this type:

```
gmap :: (a → b) → f a → f b
```

As with *crushRight*, we first introduce the function that applies to each element.

```
newtype Gmap a b = Gmap{gmap′ :: a → b}
```

This function appears to be similar to *crushRight′*, and we might expect to implement it in the same way. The difference lies in the fact that *crushRight* is only a generic consumer of a datatype while *gmap* is both a consumer and a producer. We need to abstract over a type constructor with two different element types.

The implementation of the generic function follows:

```
gmap_1                x            = x
gmap_int              x            = x
gmap_char             x            = x
gmap_+          r_a r_b (Inl a)    = Inl (gmap′ r_a a)
gmap_+          r_a r_b (Inr b)    = Inr (gmap′ r_b b)
gmap_×          r_a r_b (Prod a b) = Prod (gmap′ r_a a) (gmap′ r_b b)
gmap_dt ep1 ep2 r_a   a            = (to ep2 . gmap′ r_a . from ep1) a
```

Since this function is also a generic producer, the definitions construct new values of the same representation. Also, $gmap_{dt}$ is interesting, because it uses both *from* and *to* for converting between datatypes and their representations. We need only one representation, r_a, to map over, but we need two different embedding-projection pairs, ep_1 and ep_2, to translate between the input type, the representation, and the output type.

In order to support $gmap_{dt}$ and abstraction over two types, we need a modified *Generic* class. One option is to add a type argument and reuse that type class for all previous implementations, ignoring the extra variable. Instead, we choose to create a new class to distinguish generic functions with arity 2.

```
class Generic2 g where
  unit2     :: g Unit Unit
  int2      :: g Int Int
  char2     :: g Char Char
  plus2     :: g aT1 aT2 → g bT1 bT2 → g (Sum aT1 bT1) (Sum aT2 bT2)
  prod2     :: g aT1 aT2 → g bT1 bT2 → g (Prod aT1 bT1) (Prod aT2 bT2)
  datatype2 :: EP aT2 aT1 → EP bT2 bT1 → g aT1 bT1 → g aT2 bT2
```

Again, it is a simple matter to make Gmap an instance of *Generic2*:

```
instance Generic2 Gmap where
  unit2               = Gmap gmap𝟙
  . . .
  datatype2 ep₁ ep₂ rₐ = Gmap (gmap_dt ep₁ ep₂ rₐ)
```

Unfortunately, now that *datatype2* differs from *datatype*, we need to recreate the representation of datatypes. Fortunately, the difference is the minor addition of another embedding-projection pair. We rewrite r_{List} as $r_{List,2}$:

```
r_List,2 :: Generic2 g ⇒ g a b → g (List a) (List b)
r_List,2 rₐ = datatype2 (EP from_List to_List) (EP from_List to_List)
                        (plus2 unit2 (prod2 rₐ (r_List,2 rₐ)))
```

We can immediately use the list representation to implement the standard *map* as *mapList*:

```
mapList :: (a → b) → List a → List b
mapList f = gmap′ (r_List,2 (Gmap f))
```

Of course, our goal is to generalise this, but we do not have an appropriate dispatcher class. *FRep* will not work, because it abstracts over only one type variable. We need to extend it just as we extended *Generic* to *Generic2*:

```
class FRep2 g f where
  frep2 :: g a b → g (f a) (f b)
instance Generic2 g ⇒ FRep2 g List where
  frep2 = r_List,2
```

The class *FRep2* uses a type constructor g that has kind $\star \to \star \to \star$ to support the input and output element types. Note, however, that we still expect unary datatypes: f has only one type variable.

Finally, we provide our definition of *gmap*.

$$\begin{aligned}&gmap :: FRep2\ \mathsf{Gmap}\ \mathsf{f} \Rightarrow (\mathsf{a} \to \mathsf{b}) \to \mathsf{f}\ \mathsf{a} \to \mathsf{f}\ \mathsf{b}\\ &gmap\ f = gmap'\ (frep2\ (Gmap\ f))\end{aligned}$$

Its definition follows as the expected generalisation of *mapList*. We use *frep2* . *Gmap* (as with *crushRight*) to translate between the function *f* that applies to datatypes and one that applies to datatypes with the representation.

Exercise 8. Extend *gmap* to support constructor labels. The *Generic2* class will likely need modification. Show how $r_{List,2}$ changes when constructor labels for *Nil* and *Cons* are added.

Exercise 9. In the standard Haskell libraries, there is a function called *transpose*:

$$transpose :: [[a]] \to [[a]]$$

When applied to a list of lists, it swaps the rows and columns of its argument. For example:

$$transpose\ [[1,2,3],[4,5,6]] \leadsto [[1,4],[2,5],[3,6]]$$

transpose can be generalised to a function that swaps container datatypes. Given the generic transpose function *gtranspose*, we can, for example, swap elements of Tree3 and List:

$$\begin{aligned}&gtranspose\ (Cons\ (Leaf\ 1)\ (Cons\ (Leaf\ 2)\ (Cons\ (Leaf\ 3)\ Nil)))\\ &\quad \leadsto Leaf\ (Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil)))\end{aligned}$$

Generic transpose has a type similar to this:

$$gtranspose :: \mathsf{f}\ (\mathsf{g}\ \mathsf{a}) \to \mathsf{g}\ (\mathsf{f}\ \mathsf{a})$$

Implement *gtranspose* in EMGM.

7 Scrap Your Boilerplate

In this section we describe the “Scrap Your Boilerplate” (SYB) approach to generic programming [Lämmel and Peyton Jones, 2003, 2004]. The novel concept behind this approach to generic programming is that contrary to the two approaches discussed previously in these notes, the structure of datatypes is not directly exposed to the programmer. In the SYB approach, generic functions are defined in terms of “primitive” generic combinators, which do not have to be given by the user because they are derivable using Haskell’s **deriving** mechanism for type classes.

7.1 An example function

Recall the Expr datatype from Section 3.2), and suppose we want to implement a function that increases the value of each literal by one. Here is a simple but incorrect solution:

```
inc :: Expr Int → Expr Int
inc (Lit x) = Lit (x + 1)
```

This solution is incorrect, because we also have to write the "boilerplate" code for traversing the entire expression tree, which just leaves the structure intact and recurses into the arguments. Using SYB, we do not have to do that anymore: we signal that the other cases are uninteresting by saying:

```
inc x = x
```

Now we have the complete definition of function *inc*: increment the literals and leave the rest untouched. To ensure that this function is applied everywhere in the expression we write:

```
increment :: Data a ⇒ a → a
increment = everywhere (mkT inc)
```

This is all we need: the *increment* function increases the value of each literal by one in any Expr. It even works for LinearExprs, or LinearSystems, with no added cost.

We now proceed to explore the internals of SYB to better understand the potential of this approach and the mechanisms that are involved behind a simple generic function such as *increment*.

7.2 Run-time type representation

Contrary to the approaches to generic programming discussed earlier, SYB does not provide the structure of datatypes to the programmer, but instead offers basic combinators for writing generic programs. At the basis of these combinators is the method *typeOf* of the type class *Typeable*. Instances of this class can be automatically derived by the GHC compiler, and implement a unique representation of a type, enabling run-time type comparison and type-safe casting.

```
class Typeable a where
  typeOf :: a → TypeRep
```

An instance of *Typeable* only provides a TypeRep (type representation) of itself. The automatically derived instances of this class in GHC are guaranteed to provide a unique representation for each type, which is a necessary condition for the type-safe cast, as we will see later. So, providing an instance is as easy as adding **deriving** *Typeable* at the end of a datatype declaration, as in:

```
data MyDatatype a = MyConstructor a deriving Typeable
```

We will not discuss the internal structure of TypeRep, since a programmer should not define instances manually. The built-in derivation of *Typeable* makes SYB somewhat less portable than the previous two libraries we have seen, and makes it impossible to adapt the type representation.

The *Typeable* class is the "back-end" of SYB. The class *Data* can be considered the "front-end". It is built on top of the *Typeable* class, and adds generic folding, unfolding and reflection capabilities[5].

```
class Typeable d ⇒ Data d where
  gfoldl     :: (∀a b . Data a ⇒ c (a → b) → a → c b)
             → (∀g . g → c g)
             → d
             → c d
  gunfold    :: (∀b r . Data b ⇒ c (b → r) → c r)
             → (∀r . r → c r)
             → Constr
             → c d
  toConstr   :: d → Constr
  dataTypeOf :: d → DataType
```

The combinator *gfoldl* is named after the function *foldl* on lists, as it can be considered a "left-associative fold operation for constructor applications", with *gunfold* being the corresponding unfold. The types of these combinators may be a bit intimidating, and they are probably better understood by looking at specific instances. We will give such instances in the next subsection, since giving an instance of *Data* for a datatype is the way generic functions become available on the datatype.

7.3 Going generic: universe extension

To use the SYB combinators on a particular datatype we have to supply the instances of the datatype for the *Typeable* and the *Data* class. A programmer should not define instances of *Typeable*, but instead rely on the automatically derived instances by the compiler. For *Data*, GHC can also automatically derive instances for a datatype, but we present an instance here to show how SYB works. For example, the instance of *Data* on the List datatype is as follows:

```
instance (Typeable a, Data a) ⇒ Data (List a) where
  gfoldl k z Nil_List        = z Nil_List
```

[5] The *Data* class has, in fact, many more methods, but they all have default definitions based on these four basic combinators. They are provided as instance methods so that a programmer can define more efficient versions, specialized to the datatype in question.

$$\begin{aligned}
&\mathit{gfoldl}\ k\ z\ (\mathit{Cons}_{List}\ h\ t) && = z\ \mathit{Cons}_{List}\ \text{`}k\text{`}\ h\ \text{`}k\text{`}\ t \\
&\mathit{gunfold}\ k\ z\ l && = \textbf{case}\ \mathit{constrIndex}\ l\ \textbf{of} \\
& && \quad 1 \rightarrow z\ \mathit{Nil}_{List} \\
& && \quad 2 \rightarrow k\ (k\ (z\ \mathit{Cons}_{List}))
\end{aligned}$$

Any instance of the *Data* class follows the regular pattern of the above instance: the first argument to *gfoldl* can be seen as an application combinator, and the second argument as the base case generator. Function *gfoldl* differs from the regular *foldl* in two ways: it is not recursive, and the base case takes a constructor as argument, instead of a base case for just the Nil_{List}. When we apply *gfoldl* to function application and the identity function, it becomes the identity function itself.

$$\mathit{gfoldl}\ (\$)\ \mathit{id}\ x = x$$

We further illustrate the *gfoldl* function with another example.

$$\begin{aligned}
&\mathit{gsize} :: \mathit{Data}\ a \Rightarrow a \rightarrow \mathsf{Int} \\
&\mathit{gsize} = \mathit{unBox} \mathbin{.} \mathit{gfoldl}\ k\ (\lambda\_ \rightarrow \mathit{IntBox}\ 1)\ \textbf{where} \\
&\quad k\ (\mathit{IntBox}\ h)\ t = \mathit{IntBox}\ (h + \mathit{gsize}\ t) \\
&\textbf{newtype}\ \mathsf{IntBox}\ x = \mathit{IntBox}\{\mathit{unBox} :: \mathsf{Int}\}
\end{aligned}$$

Function *gsize* returns the number of constructors that appear in a value of any datatype that is an instance of *Data*. For example, if it is applied to a list containing pairs, it will count both the constructors of the datatype List, and of the datatype for pairs. Given the general type of *gfoldl*, we have to use a container type for the result type Int and perform additional boxing and unboxing.

Function *gunfold* acts as the dual operation of the *gfoldl*: *gfoldl* is a generic consumer, which consumes a datatype value generically to produce some result, and *gunfold* is a generic producer, which consume a datatype value to produce a datatype value generically. Its definition relies on *constrIndex*, which returns the index of the constructor in the datatype of the argument.

The two other methods of class *Data* which we have not yet mentioned are *toConstr* and *dataTypeOf*. These functions return, as their names suggest, constructor and datatype representations of the term they are applied to. As an example, we provide the instance for the List datatype[6]:

$$\begin{aligned}
&\quad \mathit{toConstr}\ \ \mathit{Nil}_{List} && = \mathit{con}_1 \\
&\quad \mathit{toConstr}\ \ (\mathit{Cons}_{List}\ \_\ \_) && = \mathit{con}_2 \\
&\quad \mathit{dataTypeOf}\ \_ && = \mathit{ty} \\
&\mathit{con}_1 = \mathit{mkConstr}\ \mathit{ty}\ \texttt{"Empty\_List"}\ \ [\,]\ \mathit{Prefix} \\
&\mathit{con}_2 = \mathit{mkConstr}\ \mathit{ty}\ \texttt{"Cons\_List"}\ \ [\,]\ \mathit{Prefix} \\
&\mathit{ty}\ \ \ = \mathit{mkDataType}\ \texttt{"Module.Name"}\ [\mathit{con}_1, \mathit{con}_2]
\end{aligned}$$

[6] Instead of `"Module.Name"` one should supply the appropriate module name, which is used for unambiguous identification of a datatype.

The functions *mkConstr* and *mkDataType* are provided by the SYB library as means for building Constr and DataType, respectively. *mkConstr* build a constructor representation given the constructor's datatype representation, name, list of field labels and fixity. *mkDataType* builds a datatype representation given the datatype's name and list of constructor representations. These two methods together form the basis of SYB's type reflection mechanism, allowing the user to inspect and construct types at runtime.

Exercise 10. Write a suitable instance of the *Data* class for the Expr datatype from Section 2.2.

The basic combinators of SYB are mainly used to define other useful combinators. It is mainly these derived combinators that are used by a generic programmer. Functions like *gunfoldl* appear very infrequently in generic programs. In the next subsection we will show many of the derived combinators in SYB.

7.4 Generic functions in SYB

We now proceed to show a few generic functions in the SYB approach. In SYB, as in many other approaches, it is often useful to first identify the type of the generic function, before selecting the most appropriate combinators to implement it.

Types of SYB combinators

Transformations, queries, and builders are some of the important basic combinators of SYB. We discuss the type of each of these.

A transformation transforms an argument value in some way, and returns a value of the same type. It has the following type:

```
type GenericT = ∀a . Data a ⇒ a → a
```

There is also a monadic variant of transformations, which allows the use of a helper monad in the transformation:

```
type GenericM m = ∀a . Data a ⇒ a → m a
```

A query function processes an input value to collect information, possibly of another type.

```
type GenericQ r = ∀a . Data a ⇒ a → r
```

A builder produces a value of a particular type.

```
type GenericB = ∀a . Data a ⇒ a
```

A builder that has access to a monad is called a "reader":

```
type GenericR m = ∀a . Data a ⇒ m a
```

The type does not require m to be a monad.

Many functions in the SYB library are suffixed with one of the letters *T*, *M*, *Q*, *B*, or *R* to help identify their usage. Examples are the functions *mkT*, *mkQ*, *mkM*, *extB*, *extR*, *extQ*, *gmapT* and *gmapQ*, some of which are defined in the rest of this section.

Basic examples

Recall the *increment* function with which we started Section 7.1. Its definition uses the higher-order combinators *everywhere* and *mkT*. The former is a traversal pattern for transformations, applying its argument everywhere it can:

```
everywhere :: GenericT → GenericT
everywhere f = f . gmapT (everywhere f)
```

Function *gmapT* maps a function only to the immediate subterms of an expression. It is defined using *gfoldl* as follows:

```
gmapT :: Data a ⇒ (∀b . Data b ⇒ b → b) → a → a
gmapT f x = unID (gfoldl k ID x)
  where
    k (ID c) y = ID (c (f y))
newtype ID x = ID{unID :: x}
```

Exercise 11. Function *everywhere* traverses a (multiway) tree. Define

```
everywhere' :: GenericT → GenericT
```

as *everywhere* but traversing in the opposite direction.

Function *mkT* lifts a, usually type-specific, function to a function that can be applied to a value of any datatype.

```
mkT :: (Typeable a, Typeable b) ⇒ (b → b) → a → a
```

For example, *mkT* (*sin* :: Float → Float), applies function *sin* if the input value is of type Float, and the identity function to an input value of any other type. The combination of the two functions *everywhere* and *mkT* allows us to lift a type-specific function to a generic function and apply it everywhere in a value.

Proceeding from transformations to queries, we define a function that sums all the integers in an expression:

```
total :: GenericQ Int
total = everything (+) (0 ‘mkQ‘ lit) where
  lit :: Expr Int → Int
  lit (Lit x) = x
  lit x       = 0
```

Queries typically use the *everything* and *mkQ* combinators. Function *everything* applies its second argument everywhere in the argument, and combines results with its first argument. Function *mkQ* lifts a type-specific function of type $\mathsf{a} \rightarrow \mathsf{b}$, together with a default value of type b, to a generic a query function of type $\mathsf{GenericQ\ b}$. If the input value is of type a, then the type-specific function is applied to obtain a b value, otherwise it returns the default value. To sum the literals in an expression, function *total* combines subresults using the addition operator $(+)$, and it keeps occurrences of literals, whereas all other values are replaced by 0.

Generic maps

Functions such as *increment* and *total* are defined in terms of functions *everywhere*, *everything*, and *mkT*, which in turn are defined in terms of the basic combinators provided by the *Data* and *Typeable* classes. Many generic functions are defined in terms of combinators of the *Data* class directly, as in the examples below. We redefine function *gsize* defined in Section 7.3 using the combinator *gmapQ*, and we define a function *glength*, which determines the number of children of a constructor, also in terms of *gmapQ*.

$$
\begin{aligned}
&\mathit{gsize} :: \mathit{Data}\ \mathsf{a} \Rightarrow \mathsf{a} \rightarrow \mathsf{Int}\\
&\mathit{gsize}\ t = 1 + \mathit{sum}\ (\mathit{gmapQ}\ \mathit{gsize}\ t)\\
&\mathit{glength} :: \mathsf{GenericQ\ Int}\\
&\mathit{glength} = \mathit{length} \mathbin{.} \mathit{gmapQ}\ (\mathit{const}\ ())
\end{aligned}
$$

The combinator *gmapQ* is one of the mapping combinators in *Data* class of the SYB library, of type:

$$\mathit{gmapQ} :: (\forall \mathsf{a} \mathbin{.} \mathit{Data}\ \mathsf{a} \Rightarrow \mathsf{a} \rightarrow \mathsf{u}) \rightarrow \mathsf{a} \rightarrow [\mathsf{u}]$$

It is rather different from the regular list *map* function, in that works on any datatype that is an instance of *Data*, and that it only applies its argument function to the immediate children of the top-level constructor. So for lists, it only applies the argument function to the head of the list and the tail of the list, but it does not recurse into the list. This explains why *gsize* recursively calls itself in *gmapQ*, while *glength*, which only counts immediate children, does not use recursion.

Exercise 12. Define the function:

$$\mathit{gdepth} :: \mathsf{GenericQ\ Int}$$

which computes the depth of a value of any datatype using *gmapQ*. The depth of a value is the maximum number of constructors on any path to a leaf in the value. For example:

$$
\begin{gathered}
\mathit{gdepth}\ [1,2] \leadsto 3\\
\mathit{gdepth}\ ((\mathit{Lit}\ 1) + (\mathit{Lit}\ 2) + (\mathit{ExprVar}\ \texttt{"x"})) \leadsto 4
\end{gathered}
$$

Exercise 13. Define the function:

$gwidth :: \mathsf{GenericQ\ Int}$

which computes the width of a value of any datatype using *gmapQ*. The width of a value is the number of elements that appear at a leaf. For example:

$$
\begin{gathered}
gwidth\ () \leadsto 1 \\
gwidth\ (Just\ 1) \leadsto 1 \\
gwidth\ ((1,2),(1,2)) \leadsto 4 \\
gwidth\ (((1,2),2),(1,2)) \leadsto 5
\end{gathered}
$$

Equality

Defining the generic equality function is a relatively simple task in the libraries we have introduced previously. Defining equality in SYB is not that easy. The reason for this is that the structural representation of datatypes is not exposed directly — in SYB, generic functions are written using combinators like *gfoldl*. To define generic equality we need to generically traverse two values at the same time, and it is not immediately clear how we can do this if *gfoldl* is our basic traversal combinator.

To implement equality, we need a generic zip-like function that can be used to pair together the children of the two argument values. Recall the type of Haskell's *zipWith* function:

$zipWith :: (\mathsf{a} \to \mathsf{b} \to \mathsf{c}) \to [\mathsf{a}] \to [\mathsf{b}] \to [\mathsf{c}]$

However, we need a generic variant that works not only for lists but for any datatype. For this purpose, SYB provides the *gzipWithQ* combinator:

$gzipWithQ :: \mathsf{GenericQ}\ (\mathsf{GenericQ\ c}) \to \mathsf{GenericQ}\ (\mathsf{GenericQ}\ [\mathsf{c}])$

The type of *gzipWithQ* is rather intricate, but if we unfold the definition of GenericQ, and omit the occurrences of $\forall\cdot$ and *Data*, the argument of *gzipWithQ* has type $\mathsf{a} \to \mathsf{b} \to \mathsf{c}$. It would take too much space to explain the details of *gzipWithQ*. Defining equality using *gzipWithQ* is easy:

```
geq :: Data a ⇒ a → a → Bool
geq x y = geq′ x y
  where
    geq′ :: GenericQ (GenericQ Bool)
    geq′ x′ y′ = (toConstr x′ == toConstr y′) ∧ and (gzipWithQ geq′ x′ y′)
```

The outer function *geq* is used to constrain the type of the function to the type of equality. Function *geq′* has a more general type since it only uses *gzipWithQ*, besides some functions on Bool.

7.5 Support for overloading

Suppose we want to implement a function with the same functionality as Haskell's **deriving** *Show*. Here is a first attempt using the combinators we have introduced in the previous sections.

```
gshows :: Data a ⇒ a → String
gshows t = "("
           ++ showConstr (toConstr t)
           ++ concat (gmapQ ((++) " " . gshows) t)
           ++
           ")"
```

Function *showConstr* :: Constr → String is the only function we have not yet introduced. Its behavior follows immediately from its type: it returns the string representing the name of the constructor. Function $gshow_s$ returns the string representation of any input value. However, it does not implement **deriving** *Show* faithfully: it inserts too many parentheses, and, what's worse, it treats all types in a uniform way, so both lists and strings are shown using the names of he constructors *Cons* and *Nil*:

```
        gshows "abc" ⇝ "((:) (a) ((:) (b) ((:) (c) ([]))))"
```

The problem here is that $gshow_s$ is "too generic": we want its behavior to be non-generic for certain datatypes, such as String. To obtain special behavior for a particular type we use the *ext* combinators of the SYB library. Since function $gshow_s$ has the type of a generic query, we use the *extQ* combinator:

```
extQ :: (Typeable a, Typeable b) ⇒ (a → q) → (b → q) → a → q
```

This combinator takes an initial generic query and extends it with the type-specific case given in its second argument. Its implementation relies on type-safe cast:

```
extQ f g a = maybe (f a) g (cast a)
```

Function *cast* relies on the *typeOf* method of the *Typeable* class, the type of which we have introduced in Section 7.2), to guarantee type equality and ultimately uses *unsafeCoerce* to perform the cast.

Using *extQ*, we can now define a better pretty-printer:

```
gshow :: Data a ⇒ a → String
gshow = (λt →
           "("
           ++ showConstr (toConstr t)
           ++ concat (gmapQ ((++) " " . gshow) t)
           ++ ")"
        ) `extQ` (show :: String → String)
```

Summarizing, the *extQ* combinator (together with its companions *extT*, *extR*, ...) is the mechanism for overloading in the SYB approach.

Exercise 14.

1. Check the behavior of function *gshow* on a value of type Char, and redefine it to behave just like the standard Haskell *show*.
2. Check the behavior of *gshow* on standard Haskell lists, and redefine it to behave just like the standard Haskell *show*. Note: since the list datatype has kind $\star \to \star$, using *extQ* will give problems. This problem is solved in SYB by defining combinators for higher kinds. Have a look at the *ext1Q* combinator.
3. Check the behavior of *gshow* on standard Haskell pairs, and redefine it to behave just like the standard Haskell *show*. Note: now the datatype has kind $\star \to \star \to \star$, but *ext2Q* is not defined! Fortunately, you can define it yourself. . .

Exercise 15. Define function *gread* :: (*Data* a) ⇒ String → [(a, String)]. Decide for yourself how complete you want your solution to be regarding whitespace, infix operators, etc. Note: you don't have to use *gunfold* directly: *fromConstr*, which is itself defined using *gunfold*, can be used instead.

7.6 Making generic functions extensible

The SYB library as described above suffers from a serious drawback: after a generic function is defined, it cannot be extended to have special behavior on a new datatype. We can, as illustrated in Section 7.5 with function *gshow*, define a function with type-specific behavior. But after such function is defined, defining another function to extend the first one with more type-specific behavior is impossible. Suppose we want to extend the *gshow* function with special behavior for a new datatype:

```
data NewDatatype = One String | Two [Int] deriving (Typeable, Data)
gshow' :: Data a ⇒ a → String
gshow' = gshow `extQ` showNewDatatype where
  showNewDatatype :: NewDatatype → String
  showNewDatatype (One s) = "String: " ++ s
  showNewDatatype (Two l) = "List: " ++ gshow l
```

Now we have:

```
gshow' (One "a") ⇝ "String: a"
```

as we expected. However:

```
gshow' (One "a", One "b") ⇝ "((,) (One \"a\") (One \"b\"))"
```

This example illustrates the problem: as soon as *gshow'* calls *gshow*, the type-specific behavior we just defined is never again taken into account, since *gshow* has no knowledge of the existence of *gshow'*.

To make generic functions in SYB extensible, Lämmel and Peyton Jones [2005] extended the SYB library, lifting generic functions to Haskell's type class system. A generic function like *gsize* is now defined as follows:

```
class Size a where
  gsize :: a → Int
```

The default case is written as an instance of the form:

```
instance ... ⇒ Size a where ...
```

Ad-hoc cases are instances of the form (using lists as an example):

```
instance Size a ⇒ Size [a] where ...
```

This requires overlapping instances, since the default case is more general than any type-specific extension. Fortunately, GHC allows overlapping instances. A problem is that this approach also needs to lift generic combinators like *gmapQ* to a type class, which requires abstraction over type classes. Abstraction over type classes is not supported by GHC. The authors then proceed to describe how to circumvent this by encoding an abstraction using dictionaries. This requires the programmer to write the boilerplate code of the proxy for the dictionary type. We do not discuss this extension and refer the reader to [Lämmel and Peyton Jones, 2005] for further information.

7.7 Variants

Scrap Your Boilerplate Reloaded [Hinze et al., 2006] is a variant of SYB that replaces the combinator based approach of SYB by a tangible representation of the structure of values. The Spine datatype is used to encode the structure of datatypes.

```
data Spine :: ⋆ → ⋆ where
  Con :: a → Spine a
  (◇)  :: Spine (a → b) → Typed a → Spine b
```

where the Typed representation is given by:

```
data Typed a = (:̂){typeOf :: Type a, val :: a}
data Type :: ⋆ → ⋆ where
  Int  :: Type Int
  List :: Type a → Type [a]
  ...
```

This approach represents the structure of datatype values by making the application of a constructor to its arguments explicit. For example, the list $[1,2]$ can be represented by[7] *Con* (:) ◇ (*Int* :̂ 1) ◇ (*List Int* :̂ [2]). We can define the usual SYB combinators such as *gfoldl* on the Spine datatype. Function *gunfold* cannot be implemented in the approach. Scrap Your Boilerplate Revolutions [Hinze and Löh, 2006] solves this problem by introducing the "type spine" and "lifted

[7] Note the difference between the list constructor (:) and the Typed constructor (:̂).

spine" views. These views allow the definition of not only generic readers such as *gunfold*, but even functions that abstract over type constructors, such as *fmap*.

A disadvantage of these variants is that generic and non-generic universe extension require recompilation of type representations and generic functions. For this reason, these variants cannot be used as a library, and should be considered a design pattern rather than a library. It is possible to make the variants extensible by using a similar approach as discussed in Section 7.6: abstraction over type classes. We refer the reader to [Hinze et al., 2006, Hinze and Löh, 2006] for further information.

8 Comparing Libraries for Generic programming

In the previous sections we have introduced three libraries for generic programming in Haskell. We did not introduce any of the other libraries for generic programming, such as PolyLib [Norell and Jansson, 2004b], RepLib [Weirich, 2006], Smash your boilerplate [Kiselyov, 2006], and Uniplate [Mitchell and Runciman, 2007], for instance. The obvious question a Haskell programmer who wants to implement a generic program now asks is: which library should I use for my project? The answer to this question depends, of course. If you want to write generic programs with special behaviour on different datatypes, you shouldn't use LIGD, but using EMGM or SYB is fine. If you want abstraction over type constructors, necessary for *gflatten* and *gmap*, you shouldn't use SYB, but LIGD and EMGM are fine. If you want generation of type representations out of the box, you shouldn't use LIGD or EMGM, but SYB is fine.

Recently, an extensive comparison of generic programming libraries (and their characteristics) has been performed [Rodriguez et al., 2008]. In the final version of these lecture notes we will include a comprehensive comparison of the libraries introduced in these notes, and mention a few points about the libraries we did not introduce. For now we refer the readers to the technical report.

9 Type-indexed datatypes in GHC

So far we have been looking at libraries for generic programming and what functions can be expressed in these libraries. An related concept in generic programming is the concept of type-indexed datatypes [Hinze et al., 2002]. Such datatypes are constructed in a generic way from an argument datatype.

For example, a type-indexed datatype is needed when we want to implement rewriting on expressions (as discussed in Section 3.2). To specify rewrite rules, we have to add meta-variables to the domain on which we want to rewrite. Adapting the domain with an extra constructor for meta-variables requires changing the original datatype, which might even not be accessible (if it is taken from a library, for instance). The best solution is to define a new datatype which adds a new constructor for meta-variables. This can be done using a type-indexed datatype.

Type-indexed datatypes are available in Generic Haskell [Löh, 2004], an extension of Haskell with generic programming constructs. The generic programming libraries available for Haskell do not offer support for defining type-indexed datatypes. However, it is possible to implement type-indexed datatypes directly in Haskell using associated datatypes [Chakravarty et al., 2005b] , a recent extension implemented in GHC.

In the final version of these lecture notes we will show how to do this.

10 Conclusions

These notes discuss three libraries for generic programming in Haskell: LIGD, EMGM, and SYB. Each of these libraries has its strengths and weaknesses, and we discuss when a generic programmer should use which library. Furthermore, we discuss how to implement type-indexed datatypes using associated datatypes.

Acknowledgements. This work has been partially funded by the Netherlands Organisation for Scientific Research (NWO), via the Real-life Datatype-Generic programming project, project nr. 612.063.613, and by the Fundação para a Ciência e Tecnologia, via the SFRH/BD/35999/2007 grant.

Bibliography

Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-70431-5.

Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *MPC'98*, volume 1422 of *LNCS*, pages 52–67. Springer, 1998.

John Seely Brown and Kurt VanLehn. Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, 4:379–426, 1980.

Alan Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253, 2005a.

Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated Types with Class. *Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of Programming Languages*, pages 1–13, 2005b.

James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel Chakravarty, editor, *Haskell'02*, pages 90–104. ACM, 2002. doi: 10.1145/581690.581698.

Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. In Henrik Nilsson, editor, *Trends in Functional Programming*, pages 199–216, 2006.

Chris Dornan, Isaac Jones, and Simon Marlow. Alex User Guide, 2003. URL `http://www.haskell.org/alex`.

Ira R. Forman and Scott H. Danforth. *Putting metaclasses to work: a new dimension in object-oriented programming*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999. ISBN 0-201-43305-2.

Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719. Springer-Verlag, 2007.

Aleksey Gurtovoy and David Abrahams. The Boost C++ metaprogramming library, 2002. URL `http://www.cs.ualberta.ca/~graphics/software/boost/libs/mpl/doc/paper/mpl_paper.pdf`.

The Haskell Prime list. Haskell prime, 2006. Wiki page at `http://hackage.haskell.org/trac/haskell-prime`.

Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16:451–482, 2006.

Ralf Hinze and Andres Löh. "Scrap Your Boilerplate" revolutions. In Tarmo Uustalu, editor, *MPC'06*, volume 4014 of *LNCS*, pages 180–208. Springer, 2006.

Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *MPC '02: Proceedings of the 6th International Conference on Mathematics of Program*

Construction, pages 148–174, London, UK, 2002. Springer-Verlag. ISBN 3-540-43857-2.

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap Your Boilerplate" reloaded. In Philip Wadler and Masimi Hagiya, editors, *FLOPS'06*, volume 3945 of *LNCS*. Springer, 2006.

Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In Tarmo Uustalu, editor, *MPC'06*, volume 4014 of *LNCS*, pages 209–234. Springer, 2006.

Helmut Horacek and Magdalena Wolska. Handling errors in mathematical formulas. In Mitsuru Ikeda, Kevin D. Ashley, and Tak-Wai Chan, editors, *Intelligent Tutoring Systems*, volume 4053 of *Lecture Notes in Computer Science*, pages 339–348. Springer, 2006. ISBN 3-540-35159-0.

Paul Hudak, John Peterson, and Joseph Fasel. A Gentle Introduction to Haskell 98, 1999. URL `http://www.haskell.org/tutorial`.

Graham Hutton and Erik Meijer. Monadic parsing in haskell. *J. Funct. Program.*, 8(4):437–444, July 1998.

Simon P. Jones and Philip Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84, New York, NY, USA, 1993. ACM.

Oleg Kiselyov. Smash your boilerplate without class and typeable. `http://article.gmane.org/gmane.comp.lang.haskell.general/14086`, 2006.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003. TLDI'03.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP'04*, pages 244–255. ACM, 2004.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP'05*, pages 204–215, 2005.

John Launchbury and Simon P. Jones. Lazy functional state threads. *SIGPLAN Not.*, 29(6):24–35, June 1994.

Josje Lodder, Johan Jeuring, and Harrie Passier. An interactive tool for manipulating logical formulae. In M. Manzano, B. Pérez Lancho, and A. Gil, editors, *Proceedings of the Second International Congress on Tools for Teaching Logic*, 2006.

Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

Andres Löh and Ralf Hinze. Open data types and open functions. In Michael Maher, editor, *PPDP'06*, pages 133–144. ACM, 2006. doi: 10.1145/1140335.1140352.

Simon Marlow and Andy Gill. Happy User Guide, 1997. URL `http://www.haskell.org/happy`.

Lambert Meertens. Calculate polytypically! In *PLILP '96: Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 1–16, London, UK, 1996. Springer-Verlag.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Haskell'07*. ACM, 2007.

E. Mory. Feedback research revisited. In D.H. Jonassen, editor, *Handbook of research for educational communications and technology*, 2003.

Ulf Norell and Patrik Jansson. Prototyping generic programming in Template Haskell. In Dexter Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 314–333. Springer-Verlag, 2004a.

Ulf Norell and Patrik Jansson. Polytypic programming in Haskell. In G. Michaelson and P. Trinder, editors, *IFL'03*, volume 3145 of *LNCS*. Springer, 2004b.

Harrie Passier and Johan Jeuring. Feedback in an interactive equation solver. In M. Seppälä, S. Xambo, and O. Caprotti, editors, *Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006*, pages 53–68. Oy WebALT Inc., 2006.

Simon Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. A special issue of the Journal of Functional Programming.

Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. D. S. Oliveira. Comparing libraries for generic programming in haskell. Technical Report UU-CS-2008-010, Department of Information and Computing Sciences, Utrecht University, 2008.

Tim Sheard. Using MetaML: A Staged Programming Language. *Advanced Functional Programming: Third International School, Braga, Portugal, September 12-19, 1998, Revised Lectures*, 1999.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming*, pages 184–207, 1996.

Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, June 1999.

Thomas van Noort. Generic views for generic types. Master's thesis, Utrecht University, 2008.

Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.

Philip Wadler. Theorems for free! In *FPCA'89*, pages 347–359. ACM, 1989.

Philip Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, New York, NY, USA, 1990. ACM.

Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less adhoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

Stephanie Weirich. RepLib: a library for derivable type classes. In *Haskell'06*, pages 1–12. ACM, 2006. doi: 10.1145/1159842.1159844.

From reduction-based to reduction-free normalization (preliminary version)

Olivier Danvy

Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
danvy@daimi.au.dk
http://www.daimi.au.dk/~danvy

Abstract. We document a method to construct reduction-free normalization functions. Starting from a reduction-based normalization function, i.e., the iteration of a one-step reduction function, we successively subject it to refocusing (i.e., deforestation of the intermediate reduced terms in the reduction sequence), fusion of auxiliary functions, refunctionalization (i.e., the converse of defunctionalization), and direct-style transformation (i.e., the converse of the CPS transformation). We consider three simple examples and treat them in detail: for the first one, arithmetic expressions, we construct an evaluation function; for the second one, terms in the free monoid, we construct an accumulator-based flatten function; for the third one, applicative order lambda-terms with explicit substitutions and call/cc, and , we construct a call-by-value evaluator. The results are traditional reduction-free normalization functions. The overall method builds on previous work on a syntactic correspondence between reduction semantics and abstract machines and on a functional correspondence between evaluators and abstract machines.

1 Introduction

Normalization by evaluation is a 'reduction-free' approach to normalizing terms. Instead of repeatedly reducing a term towards its normal form, as in the traditional reduction-based approach, one uses an *extensional normalization function* that does not construct any intermediate term and directly yields a normal form, if there is any [30]. Normalization by evaluation has been developed in intuitionistic type theory [20, 49, 61], proof theory [10, 11], category theory [6, 22, 58], λ-definability [44], partial evaluation [24, 25, 38], and formal semantics [1, 41, 42]. The more complicated the terms and the notions of reduction, the more complicated the normalization functions.

Normalization by evaluation therefore requires one to extensionally define a reduction-free normalization function, which is non-trivial [7, 8]. Nevertheless, it is our contention that the computational content of a reduction-based normalization function—i.e., a function intensionally defined as the transitive closure of one-step reduction—can pave the way to constructing a reduction-free normalization function:

Our starting point: We start from a reduction semantics for a language of terms [40], i.e., an abstract syntax, a notion of reduction in the form of a collection of redexes and the corresponding contraction function, and a reduction strategy. The reduction strategy takes the form of a grammar of reduction contexts, its associated plug function, and a decomposition function mapping a term to a value or to a reduction context and a redex (we assume this decomposition to be unique). Thus equipped, we define a one-step reduction function as a function whose fixed points are values, and which otherwise decomposes a non-value term into a reduction context and a redex, contracts this redex, and plugs the contractum into the context:

$$\begin{tikzcd}
\text{non-value term} \arrow[r, "\text{decompose}"] \arrow[d, dashed, "\text{one-step reduction}"'] & \text{context} \times \text{redex} \arrow[d, "\text{context-(in)sensitive contraction}"] \\
\text{term} & \text{context} \times \text{contractum} \arrow[l, "\text{plug}"]
\end{tikzcd}$$

A reduction-based normalization function is defined as the iteration of this one-step reduction function.

A syntactic correspondence: On the way to reaching a normal form, the reduction-based normalization function repeatedly decomposes, contracts, and plugs. Observing that most of the time, the decomposition function is applied to the result of the plug function [37], Nielsen and the author have suggested to deforest the intermediate term by replacing the composition of the decomposition function and of the plug function by a *refocus* function that directly maps a reduction context and a contractum to the next reduction context and redex, if there are any. Such a refocused normalization function (i.e., a normalization function using a refocus function instead of a decomposition function and a plug function) takes the form of a small-step abstract machine.

A functional correspondence: A big-step abstract machine is often a defunctionalized continuation-passing program [3–5, 19, 28]. When this is the case, such abstract machines can be refunctionalized [33, 36] and transformed into direct style [23].

It is our experience that starting from a reduction semantics for a language of terms, we can refocus the corresponding reduction-based normalization function into a small-step abstract machine, and refunctionalize the corresponding big-step abstract machine into a reduction-free normalization function. The goal of this article is to illustrate it with three simple examples: arithmetic expressions, terms of the free monoid, and applicative-order lambda-terms with explicit substitutions and call/cc.

Overview: In Section 2, we implement a reduction semantics for arithmetic expressions in complete detail and in Standard ML, and we define the corresponding reduction-based normalization function. In Section 3, we refocus the reduction-based normalization function of Section 2 into a small-step abstract machine, and we present the corresponding reduction-free normalization function. In Sections 4 and 5, we go through the same motions for terms in the free monoid. In Section 2 and 7, we repeat the construction with lambda-terms.

Sections 2, 4 and 6 might appear as intimidating; however, except that they are expressed in ML, they describe straightforward reduction semantics as have been developed by Felleisen and his co-workers for the last two decades [39, 40, 62]. For this reason, these three sections have a parallel structure. Similarly, to emphasize that the construction of a reduction-free normalization function out of a reduction-based normalization function is systematic, we have also given Sections 3, 5 and 7 a parallel structure.

Prerequisites: The reader is expected to have some familiarity with the programming language Standard ML [53]; reduction semantics [37,40]; the CPS transformation [31, 60] and its left inverse, the direct-style transformation [23, 32]; and defunctionalization [36,57] and its left inverse, refunctionalization [33]. In particular, we build on evaluation contexts being defunctionalized continuations [27].

Contribution: This lecture note builds on work that was carried out at the University of Aarhus over the last 7 years and that gave rise to a number of doctoral theses [2,13,18,29,51,52,54]. The examples of arithmetic expressions and of the free monoid were presented at WRS'04 [26]. The example of lambda-terms originates in a joint work with Lasse R. Nielsen [37], Małgorzata Biernacka [16, 17], and Mads Sig Ager, Dariusz Biernacki, and Jan Midtgaard [3, 5]. Fusing the transition functions and the trampoline function of a small-step abstract machine to obtain a big-step one is a joint work with Kevin Millikin [34].

2 A reduction semantics for arithmetic expressions

To define a reduction semantics for simplified arithmetic expressions (integer literals and additions), we specify their abstract syntax, their notion of reduction (computing the sum of two integers), their reduction contexts and the corresponding plug function, and how to decompose them into a reduction context and the left-most inner-most redex, if there is one. We then define a one-step reduction function that decomposes a non-value term into a reduction context and a redex, contracts the redex, and plugs the contractum into the context. We can finally define a reduction-based normalization function that repeatedly applies the one-step reduction function until a value, i.e., a normal form, is reached.

2.1 Abstract syntax

An arithmetic expression is either a literal or the addition of two terms:

```
datatype term = LIT of int
              | ADD of term * term
```

2.2 Notion of contraction

A redex is the sum of two literals, and we implement contraction as computing this sum:

```
datatype redex = SUM of int * int

(*  contract : redex -> term  *)
fun contract (SUM (n1, n2))
    = LIT (n1 + n2)
```

The left-most inner-most reduction strategy converges and yields a literal.

2.3 Reduction contexts

We seek the left-most inner-most redex in a term. The grammar of reduction contexts and the corresponding plug function are as follows:

```
datatype context = C0
                 | C1 of term * context
                 | C2 of int * context

(*  plug : context * term -> term  *)
fun plug (C0, t)
    = t
  | plug (C1 (t', c), t)
    = plug (c, ADD (t, t'))
  | plug (C2 (n, c), t)
    = plug (c, ADD (LIT n, t))
```

2.4 Decomposition

A term is a value (i.e., it does not contain any redex) or it can be decomposed into a reduction context and a redex:

```
datatype value_or_decomposition = VAL of term
                                | DEC of context * redex
```

(No term is stuck.)

The decomposition function recursively searches for the left-most inner-most redex in a term. It is usually left unspecified in the literature [40]. We define it here it in a form we have found convenient in our previous study of reduction semantics [37], namely as a big-step abstract machine with two state-transition functions, `decompose'` and `decompose'_aux`: `decompose'` traverses a given term and accumulates the reduction context until it finds a value, and `decompose'_aux` dispatches on the accumulated context to decide whether the given term is a value, a redex has been found, or the search must continue:

```
(*  decompose' : term * context -> value_or_decomposition  *)
fun decompose' (LIT n, c)
    = decompose'_aux (c, n)
  | decompose' (ADD (t1, t2), c)
    = decompose' (t1, C1 (t2, c))
(*  decompose'_aux : context * int -> value_or_decomposition  *)
and decompose'_aux (C0, n)
    = VAL (LIT n)
  | decompose'_aux (C1 (t2, c), n)
    = decompose' (t2, C2 (n, c))
  | decompose'_aux (C2 (n', c), n)
    = DEC (c, SUM (n', n))

(*  decompose : term -> value_or_decomposition  *)
fun decompose t
    = decompose' (t, C0)
```

Lemma 1. *A term* `t` *is either a value or there exists a unique context* `c` *such that* `decompose t` *evaluates to* `DEC (c, r)`, *where* `r` *a redex.*

Proof. Immediate.

2.5 One-step reduction

We are now in position to define a one-step reduction function as a function that (1) maps a non-value term into a reduction context and a redex, (2) contracts the redex, and (3) plugs the contractum in the reduction context:

```
(*  reduce : term -> term  *)
fun reduce t
    = (case decompose t
         of (VAL t')
            => t'
          | (DEC (c, r))
            => plug (c, contract r))
```

2.6 Reduction-based normalization

A reduction-based normalization function is one that iterates the one-step reduction function until it yields a value (i.e., a fixed point):

```
(*  normalize : term -> term  *)
fun normalize t
    = (case reduce t
         of (LIT n)
            => LIT n
          | t'
            => normalize t')
```

In the following definition, we inline `reduce` in order to directly check whether `decompose` yields a value or a decomposition:

```
(*  iterate0 : value_or_decomposition -> term  *)
fun iterate0 (VAL t)
    = t
  | iterate0 (DEC (c, r))
    = iterate0 (decompose (plug (c, contract r)))

(*  normalize0 : term -> term  *)
fun normalize0 t
    = iterate0 (decompose t)
```

2.7 Reduction-based normalization, typefully

The type of `normalize0` is not informative. To make it appear more clearly that the normalization function yields normal forms, i.e., integers, we can refine the type of values to be that of integers, and adjust the first clause of `decompose'_aux` and the reduction function:

```
datatype value_or_decomposition = VAL of int (* was: term *)
                                | DEC of context * redex

    ...
and decompose'_aux (C0, n)
    = VAL n
  | ...

(*  reduce : term -> term  *)
fun reduce t
    = (case decompose t
         of (VAL n)
            => LIT n
          | (DEC (c, r))
            => plug (c, contract r))
```

The reduction-based normalization function can then return an integer rather than a literal:

```
(*  iterate1 : value_or_decomposition -> int  *)
fun iterate1 (VAL n)
    = n
  | iterate1 (DEC (c, r))
    = iterate1 (decompose (plug (c, contract r)))
```

```
(*  normalize1 : term -> int  *)
fun normalize1 t
    = iterate1 (decompose t)
```

The type of `normalize1` is more informative than that of `normalize0` since it makes it clear that applying `normalize1` to a term yields a value.

2.8 Summary and conclusion

We have implemented in ML, in complete detail, a reduction semantics for arithmetic expressions. Using this reduction semantics, we have presented two reduction-based normalization functions.

2.9 Exercises

1. Implement the reduction semantics above in the programming language of your choice.
2. Extend the source language with multiplication and adapt your implementation.

3 From reduction-based to reduction-free normalization

In this section, we transform the reduction-based normalization function of Section 2.7 into a reduction-free normalization function, i.e., one where no intermediate term is ever constructed. We first refocus the reduction-based normalization function [37] to deforest the intermediate terms, and we obtain a small-step abstract machine implementing the iteration of the refocus function. We then transform this small-step abstract machine into a big-step one [34]. This abstract machine is in defunctionalized form [36], and we refunctionalize it [33]. The result is in continuation-passing style and we re-express it in direct style [23]. The resulting direct-style function is a traditional evaluator for arithmetic expressions; in particular, it is reduction-free.

3.1 Plugging and decomposition

In the reduction-based normalization function of Section 2.7, `decompose` is always applied to the result of `plug` after the first decomposition. Let us add a vacuous initial call to `plug` so that in all cases, `decompose` is applied to the result of `plug`:

```
(*  normalize2 : term -> int  *)
fun normalize2 t
    = iterate1 (decompose (plug (C0, t)))
```

3.2 Refocusing

As investigated earlier by Nielsen and the author [37], the composition of `decompose` and `plug` can be deforested into one `refocus` function to avoid the construction of intermediate terms. In addition, this `refocus` function can be expressed very simply in terms of the decomposition functions of Section 2.4 (and this is the reason why we chose to specify them precisely like that):

```
(*  refocus : context * term -> value_or_decomposition  *)
fun refocus (c, t)
    = decompose' (t, c)
```

The refocused evaluation function therefore reads as follows:

```
(*  iterate3 : value_or_decomposition -> int  *)
fun iterate3 (VAL v)
    = v

  | iterate3 (DEC (c, r))
    = iterate3 (refocus (c, contract r))

(*  normalize3 : term -> int  *)
fun normalize3 t
    = iterate3 (refocus (C0, t))
```

The refocused normalization function is reduction-free because it is no longer based on a (one-step) reduction function. Instead, the refocus function directly maps a reduction context and a contractum to the next reduction context and redex, if there are any.

3.3 From refocused normalization function to staged abstract machine

The refocused normalization function is small-step -abstract machine in the sense that `decompose'` and `decompose'_aux` form a transition function and `iterate3` is a 'trampoline' [43], i.e., another transition function that keeps activating the two others until a value is obtained. Using Ohori and Sasano's 'lightweight fusion by fixed-point promotion' [55], let us fuse `iterate3` and `refocus` (i.e., `decompose'` and `decompose'_aux`, which we rename `refocus4` and `refocus4_aux` for the occasion) so that `iterate3` is directly applied to the result of `decompose'` and `decompose'_aux`. The result is a (tail-recursive) state-transition function, i.e., a big-step abstract machine [56]:

```
(*  iterate4 : value_or_decomposition -> int  *)
fun iterate4 (VAL v)
    = v
  | iterate4 (DEC (c, r))
    = refocus4 (contract r, c)
```

```
(*  refocus4 : term * context -> int  *)
and refocus4 (LIT n, c)
    = refocus4_aux (c, n)
  | refocus4 (ADD (t1, t2), c)
    = refocus4 (t1, C1 (t2, c))
(*  refocus4_aux : context * int -> int  *)
and refocus4_aux (C0, n)
    = iterate4 (VAL n)
  | refocus4_aux (C1 (t2, c), n)
    = refocus4 (t2, C2 (n, c))
  | refocus4_aux (C2 (n', c), n)
    = iterate4 (DEC (c, SUM (n', n)))

(*  normalize4 : term -> int  *)
fun normalize4 t
    = refocus4 (t, C0)
```

The structure of this abstract machine is remarkable because `iterate4` implements the contraction rules of the reduction semantics and `refocus4` and `refocus4_aux` implement its congruence rules—a distinction that usually requires a non-trivial analysis to establish for existing abstract machines [46].

3.4 Inlining and simplification

Since `iterate4` and `contract` are only pedagogical devices, let us inline them to streamline the abstract machine. Inlining `contract`, in the last clause of `refocus4_aux`, yields the following clause:

```
  | refocus4_aux (C2 (n', c), n)
    = refocus4 (LIT (n' + n), c)
```

Since `refocus4` is defined by cases on its first argument, this clause can be simplified as follows:

```
  | refocus4_aux (C2 (n', c), n)
    = refocus4_aux (c, n' + n)
```

The resulting simplified machine is an 'eval/apply' abstract machine [48].

3.5 Refunctionalization

Like many other abstract machines [3–5, 19, 28], the abstract machine of Section 3.4 is in defunctionalized form [36]: the reduction contexts, together with `refocus4_aux`, are the first-order counterpart of a function. The higher-order counterpart of the abstract machine reads as follows:

```
(*  refocus5 : term * (int -> int) -> int  *)
fun refocus5 (LIT n, c)
    = c n
```

```
  | refocus5 (ADD (t1, t2), c)
    = refocus5 (t1,
                fn n1 => refocus5 (t2,
                                   fn n2 => c (n1 + n2)))

(*  normalize5 : term -> int  *)
fun normalize5 t
    = refocus5 (t, fn n => n)
```

3.6 Back to direct style

The refunctionalized definition of Section 3.5 is in continuation-passing style since it has a functional accumulator and all of its calls are tail calls [23,31]. Its direct-style counterpart reads as follows:

```
(*  refocus6 : term -> int  *)
fun refocus6 (LIT n)
    = n
  | refocus6 (ADD (t1, t2))
    = (refocus6 t1) + (refocus6 t2)

(*  normalize6 : term -> int  *)
fun normalize6 t
    = refocus6 t
```

The resulting definition is that of the usual evaluation function for arithmetic expressions, i.e., a traditional reduction-free normalization function.

3.7 Summary and conclusion

We have refocused the reduction-based normalization function of Section 2 into a small-step abstract machine, and we have exhibited the corresponding reduction-free normalization function.

3.8 Exercises

1. Reproduce the construction above in the programming language of your choice.
2. Extend the source language with multiplication and adapt the construction.

4 A reduction semantics for terms in the free monoid

To define a reduction semantics for terms in the free monoid over a given carrier set, we specify their abstract syntax (a distinguished unit element, the other elements of the carrier set, and products of terms), their notion of reduction (oriented conversion rules), their reduction contexts and the corresponding plug function, and how to decompose them into a reduction context and the right-most inner-most redex, if there is one. We then define a one-step reduction function that decomposes a non-value term into a reduction context and a redex, contracts the redex, and plugs the contractum into the context. We can finally define a reduction-based normalization function that repeatedly applies the one-step reduction function until a value, i.e., a normal form, is reached.

4.1 Abstract syntax

Given a type `elem` of carrier-set elements, a term in the free monoid is either the unit element, an element of type `elem`, or the product of two terms:

```
datatype term = UNIT
              | ELEM of elem
              | PROD of term * term
```

Terms in the free monoid obey conversion rules: the unit element is neutral for the product (both on the left and on the right), and the product is associative.

4.2 Notion of contraction

We introduce a notion of reduction by orienting the conversion rules into reduction rules:

$$\begin{aligned} \texttt{PROD (UNIT, t)} &\longrightarrow \texttt{t} \\ \texttt{ELEM e} &\longrightarrow \texttt{PROD (ELEM e, UNIT)} \\ \texttt{PROD (PROD (t11, t12), t2)} &\longrightarrow \texttt{PROD (t11, PROD (t12, t2))} \end{aligned}$$

We represent redexes as a data type and implement their contraction with the corresponding reduction rules:

```
datatype redex = LEFT_UNIT of term
               | RIGHTMOST of elem
               | ASSOC of (term * term) * term

(*  contract : redex -> term  *)
fun contract (LEFT_UNIT t)
    = t
  | contract (RIGHTMOST e)
    = PROD (ELEM e, UNIT)
  | contract (ASSOC ((t11, t12), t2))
    = PROD (t11, PROD (t12, t2))
```

The right-most inner-most reduction strategy converges and yields a flat, list-like term in normal form.

4.3 Reduction contexts

We seek the right-most inner-most redex in a term. The grammar of reduction contexts and the corresponding plug function are as follows:

```
datatype context = C0
                 | C1 of term * context

(*  plug : context * term -> term  *)
fun plug (C0, t)
    = t
  | plug (C1 (t1, c), t2)
    = plug (c, PROD (t1, t2))
```

4.4 Decomposition

A term is a value (i.e., it does not contain any redex) or it can be decomposed into a reduction context and a redex:

```
datatype value_or_decomposition = VAL of term
                                | DEC of context * redex
```

(No term is stuck.)

The decomposition function recursively searches for the right-most inner-most redex in a term. As in Section 2.4, we define it with two auxiliary functions, `decompose'` and `decompose'_aux`: `decompose'` traverses a given term and accumulates the reduction context until it finds a redex or a value, and `decompose'_aux` dispatches on the accumulated context to decide whether the given term is a value, a redex has been found, or the search must continue:

```
(*  decompose' : term * context -> value_or_decomposition  *)
fun decompose' (UNIT, c)
    = decompose'_aux (c, UNIT)
  | decompose' (ELEM e, c)
    = DEC (c, RIGHTMOST e)
  | decompose' (PROD (t1, t2), c)
    = decompose' (t2, C1 (t1, c))

(*  decompose'_aux : context * term -> value_or_decomposition  *)
and decompose'_aux (C0, t)
    = VAL t
  | decompose'_aux (C1 (UNIT, c), t2)
    = DEC (c, LEFT_UNIT t2)
  | decompose'_aux (C1 (ELEM e, c), t2)
    = decompose'_aux (c, PROD (ELEM e, t2))
  | decompose'_aux (C1 (PROD (t11, t12), c), t2)
    = DEC (c, ASSOC ((t11, t12), t2))

(*  decompose : term -> value_or_decomposition  *)
fun decompose t
    = decompose' (t, C0)
```

Lemma 2. *A term* `t` *is either a value or there exists a unique context* `c` *such that* `decompose t` *evaluates to* `DEC (c, r)`*, where* `r` *a redex.*

Proof. Immediate.

4.5 One-step reduction

We are now in position to define a one-step reduction function as a function that (1) maps a non-value term into a reduction context and a redex, (2) contracts the redex, and (3) plugs the contractum in the reduction context:

```
(*  reduce : term -> term  *)
fun reduce t
    = (case decompose t
         of (VAL t')
            => t'
          | (DEC (c, r))
            => plug (c, contract r))
```

4.6 Reduction-based normalization

A reduction-based normalization function is one that iterates the one-step reduction function until it yields a value. In the following definition, and as in Section 2.6, we inline `reduce` and directly check whether `decompose` yields a value or a decomposition:

```
(*  iterate0 : value_or_decomposition -> term  *)
fun iterate0 (VAL t)
    = t
  | iterate0 (DEC (c, r))
    = iterate0 (decompose (plug (c, contract r)))

(*  normalize0 : term -> term  *)
fun normalize0 t
    = iterate0 (decompose t)
```

4.7 Reduction-based normalization, typefully

As in Section 2.7, the type of `normalize0` is not informative. To make it appear more clearly that the normalization function yields normal forms, let us introduce a data type of terms in normal form:

```
datatype term_nf = UNIT_nf
                 | PROD_nf of elem * term_nf
```

We can then refine the type of values by glueing it with the corresponding normal form:

```
datatype value_or_decomposition = VAL of term * term_nf
                                | DEC of context * redex
```

We must then adjust `decompose'_aux` to construct values both as regular terms and as terms in normal form:

```
(*  decompose' : term * context -> value_or_decomposition  *)
fun decompose' (UNIT, c)
    = decompose'_aux (c, UNIT, UNIT_nf)
  | decompose' (ELEM e, c)
    = DEC (c, RIGHTMOST e)
  | decompose' (PROD (t1, t2), c)
    = decompose' (t2, C1 (t1, c))
(*  decompose'_aux : context * term * term_nf
                     -> value_or_decomposition  *)
and decompose'_aux (C0, t, t_nf)
    = VAL (t, t_nf)
  | decompose'_aux (C1 (UNIT, c), t2, t2_nf)
    = DEC (c, LEFT_UNIT t2)
  | decompose'_aux (C1 (ELEM e, c), t2, t2_nf)
    = decompose'_aux (c, PROD (ELEM e, t2), PROD_nf (e, t2_nf))
  | decompose'_aux (C1 (PROD (t11, t12), c), t2, t2_nf)
    = DEC (c, ASSOC ((t11, t12), t2))

(*  decompose : term -> value_or_decomposition  *)
fun decompose t
    = decompose' (t, C0)
```

The reduction-based normalization function can then return the representation of the term in normal form:

```
(*  iterate1 : value_or_decomposition -> term_nf  *)
fun iterate1 (VAL (t, t_nf))
    = t_nf
  | iterate1 (DEC (c, r))
    = iterate1 (decompose (plug (c, contract r)))

(*  normalize1 : term -> term_nf  *)
fun normalize1 t
    = iterate1 (decompose t)
```

The type of `normalize1` is more informative than that of `normalize0` since it makes it clear that applying `normalize1` to a term yields a term in normal form.

4.8 Summary and conclusion

We have implemented in ML a reduction semantics for terms in the free monoid, given its carrier set. Using this reduction semantics, we have presented two reduction-based normalization functions.

4.9 Exercises

1. Implement the reduction semantics above in the programming language of your choice.

5 From reduction-based to reduction-free normalization

In this section, we transform the reduction-based normalization function of Section 4.7 into a reduction-free normalization function, i.e., one where no intermediate term is ever constructed. We first refocus the reduction-based normalization function and we obtain a small-step abstract machine. We then transform this small-step abstract machine into a big-step one. This abstract machine is in defunctionalized form, and we refunctionalize it. The result is in continuation-passing style and we re-express it in direct style. The resulting direct-style function is a traditional flatten function with an accumulator; in particular, it is reduction-free.

5.1 Plugging and decomposition

In the reduction-based normalization function of Section 4.7, `decompose` is always applied to the result of `plug` after the first decomposition. Let us add a vacuous initial call to `plug` so that in all cases, `decompose` is applied to the result of `plug`:

```
(*  normalize2 : term -> term_nf  *)
fun normalize2 t
    = iterate1 (decompose (plug (C0, t)))
```

5.2 Refocusing

As in Section 3.2, we now deforest the composition of `decompose` and `plug` into one `refocus` function:

```
(*  refocus : context * term -> value_or_decomposition  *)
fun refocus (c, t)
    = decompose' (t, c)
```

The refocused evaluation function therefore reads as follows:

```
(*  iterate3 : value_or_decomposition -> term_nf  *)
fun iterate3 (VAL (t, t_nf))
    = t_nf
  | iterate3 (DEC (c, r))
    = iterate3 (refocus (c, contract r))

(*  normalize3 : term -> term_nf  *)
fun normalize3 t
    = iterate3 (refocus (C0, t))
```

The refocused normalization function is reduction-free because it is no longer based on a reduction function and it no longer constructs intermediate terms.

5.3 From refocused evaluation function to staged abstract machine

Again, the refocused evaluation function is a small-step abstract machine in the sense that `decompose'` and `decompose'_aux` form a transition function and `iterate3` is a 'trampoline'. Let us fuse `iterate3` and `refocus` (i.e., `decompose'` and `decompose'_aux`, which we rename `refocus4` and `refocus4_aux` as in Section 3.3), so that `iterate3` is directly applied to the result of `decompose'` and `decompose'_aux`. The result is the following big-step abstract machine:

```
(*  iterate4 : value_or_decomposition -> term_nf  *)
fun iterate4 (VAL (t, t_nf))
    = t_nf
  | iterate4 (DEC (c, r))
    = refocus4 (contract r, c)
(*  refocus4 : term * context -> term_nf  *)
and refocus4 (UNIT, c)
    = refocus4_aux (c, UNIT, UNIT_nf)
  | refocus4 (ELEM e, c)
    = iterate4 (DEC (c, RIGHTMOST e))
  | refocus4 (PROD (t1, t2), c)
    = refocus4 (t2, C1 (t1, c))
(*  refocus4_aux : context * term * term_nf -> term_nf  *)
and refocus4_aux (C0, t, t_nf)
    = iterate4 (VAL (t, t_nf))
  | refocus4_aux (C1 (UNIT, c), t2, t2_nf)
    = iterate4 (DEC (c, LEFT_UNIT t2))
  | refocus4_aux (C1 (ELEM e, c), t2, t2_nf)
    = refocus4_aux (c, PROD (ELEM e, t2), PROD_nf (e, t2_nf))
  | refocus4_aux (C1 (PROD (t11, t12), c), t2, t2_nf)
    = iterate4 (DEC (c, ASSOC ((t11, t12), t2)))

(*  normalize4 : term -> term_nf  *)
fun normalize4 t
    = refocus4 (t, C0)
```

Again, this abstract machine is staged: `iterate4` implements the contraction rules of the reduction semantics and `refocus4` and `refocus4_aux` implement its congruence rules.

5.4 Inlining and simplification

As in Section 3.4, we inline `iterate4` and `contract` to streamline the abstract machine. Three cases occur:

1. The clause

   ```
   | refocus4 (ELEM e, c)
     = iterate4 (DEC (c, RIGHTMOST e))
   ```

   after inlining `iterate4` and `contract`, reads as follows:

```
| refocus4 (ELEM e, c)
  = refocus4 (PROD (ELEM e, UNIT), c)
```

Since `refocus4` is defined by cases on its first argument, this clause can be simplified as follows (skipping two steps):

```
| refocus4 (ELEM e, c)
  = refocus4_aux (c, PROD (ELEM e, UNIT), PROD_nf (e, UNIT_nf))
```

2. The clause

   ```
   | refocus4_aux (C1 (UNIT, c), t2, t2_nf)
      = iterate4 (DEC (c, LEFT_UNIT t2))
   ```

   after inlining `iterate4` and `contract`, reads as follows:

   ```
   | refocus4_aux (C1 (UNIT, c), t2, t2_nf)
     = refocus4 (t2, c)
   ```

   We know, however, that `t2` is in normal form, and therefore we can directly call `refocus4_aux` instead:

   ```
   | refocus4_aux (C1 (UNIT, c), t2, t2_nf)
     = refocus4_aux (c, t2, t2_nf)
   ```

3. The clause

   ```
   | refocus4_aux (C1 (PROD (t11, t12), c), t2, t2_nf)
     = iterate4 (DEC (c, ASSOC ((t11, t12), t2)))
   ```

   after inlining `iterate4` and `contract`, reads as follows:

   ```
   | refocus4_aux (C1 (PROD (t11, t12), c), t2, t2_nf)
     = refocus4 (PROD (t11, PROD (t12, t2)), c)
   ```

   Since `refocus4` is defined by cases on its first argument, this clause can be simplified as follows (skipping two steps):

   ```
   | refocus4_aux (C1 (PROD (t11, t12), c), t2, t2_nf)
     = refocus4 (t2, C1 (t12, C1 (t11, c)))
   ```

   We know, however, that `t2` is in normal form, and therefore we can directly call `refocus4_aux` instead:

   ```
       | refocus4_aux (C1 (PROD (t11, t12), c), t2, t2_nf)
         = refocus4_aux (C1 (t12, C1 (t11, c)), t2, t2_nf)
   ```

In the resulting definition of `refocus4_aux`, we observe that the second parameter is dead, i.e., that it is never used. Eliminating it (and renaming the last parameter to `a`) yields the following definition:

```
(*  refocus4 : term * context -> term_nf  *)
fun refocus4 (UNIT, c)
    = refocus4_aux (c, UNIT_nf)
  | refocus4 (ELEM e, c)
    = refocus4_aux (c, PROD_nf (e, UNIT_nf))
  | refocus4 (PROD (t1, t2), c)
    = refocus4 (t2, C1 (t1, c))

(*  refocus4_aux : context * term_nf -> term_nf  *)
and refocus4_aux (C0, a)
    = a
  | refocus4_aux (C1 (UNIT, c), a)
    = refocus4_aux (c, a)
  | refocus4_aux (C1 (ELEM e, c), a)
    = refocus4_aux (c, PROD_nf (e, a))
  | refocus4_aux (C1 (PROD (t11, t12), c), a)
    = refocus4_aux (C1 (t12, C1 (t11, c)), a)
```

5.5 Refunctionalization

The above definitions of `refocus4` and `refocus4_aux` are not in defunctionalized form because of the last clause of `refocus4_aux` [36]. To put them in defunctionalized form (eureka), we need to introduce one more auxiliary function:

```
(*  refocus4 : term * context -> term_nf  *)
fun refocus4 (UNIT, c)
    = refocus4_aux (c, UNIT_nf)
  | refocus4 (ELEM e, c)
    = refocus4_aux (c, PROD_nf (e, UNIT_nf))
  | refocus4 (PROD (t1, t2), c)
    = refocus4 (t2, C1 (t1, c))
(*  refocus4_aux : context * term_nf -> term_nf  *)
and refocus4_aux (C0, a)
    = a
  | refocus4_aux (C1 (t', c), a)
    = refocus4_aux' (t', c, a)
(*  refocus4_aux' : term * context * term_nf -> term_nf  *)
and refocus4_aux' (UNIT, c, a)
    = refocus4_aux (c, a)
  | refocus4_aux' (ELEM e, c, a)
    = refocus4_aux (c, PROD_nf (e, a))
  | refocus4_aux' (PROD (t11, t12), c, a)
    = refocus4_aux' (t12, C1 (t11, c), a)
```

Now the reduction contexts, together with `refocus4_aux`, are the first-order counterpart of a function. The higher-order counterpart of the normalization function reads as follows:

```
(*  refocus5 : term * (term_nf -> term_nf) -> term_nf  *)
```

```
fun refocus5 (UNIT, c)
    = c UNIT_nf
  | refocus5 (ELEM e, c)
    = c (PROD_nf (e, UNIT_nf))
  | refocus5 (PROD (t1, t2), c)
    = refocus5 (t2, fn t2'_nf => refocus5_aux' (t1, c, t2'_nf))

(*  refocus5_aux' : term * (term_nf -> term_nf) * term_nf -> term_nf  *)
and refocus5_aux' (UNIT, c, a)
    = c a
  | refocus5_aux' (ELEM e, c, a)
    = c (PROD_nf (e, a))
  | refocus5_aux' (PROD (t11, t12), c, a)
    = refocus5_aux' (t12, fn a' =>
        refocus5_aux' (t11, c, a'), a)

(*  normalize5 : term -> term_nf  *)
fun normalize5 t
    = refocus5 (t, fn a => a)
```

5.6 Back to direct style

The refunctionalized definition of Section 5.5 is in continuation-passing style since it has a functional accumulator and all of its calls are tail calls. Its direct-style counterpart reads as follows:

```
(*  refocus6 : term -> term_nf  *)
fun refocus6 UNIT
    = UNIT_nf
  | refocus6 (ELEM e)
    = PROD_nf (e, UNIT_nf)
  | refocus6 (PROD (t1, t2))
    = refocus6_aux' (t1, refocus6 t2)
(*  refocus6_aux : term * term_nf -> term_nf  *)
and refocus6_aux' (UNIT, a)
    = a
  | refocus6_aux' (ELEM e, a)
    = PROD_nf (e, a)
  | refocus6_aux' (PROD (t11, t12), a)
    = refocus6_aux' (t11, refocus6_aux' (t12, a))

(*  normalize6 : term -> term_nf  *)
fun normalize6 t
    = refocus6 t
```

The resulting definition is that of a flatten function with an accumulator, i.e., an uncurried version of the usual reduction-free normalization function for the free monoid [9, 12, 15, 47].

5.7 Summary and conclusion

We have refocused the reduction-based normalization function of Section 4 into a small-step abstract machine, and we have exhibited the corresponding reduction-free normalization function.

The resulting reduction-free normalization function could be streamlined by skipping `refocus6` as follows:

```
(*  normalize7 : term -> term_nf  *)
fun normalize7 t
    = refocus6_aux' (t, UNIT_nf)
```

This simplified reduction-free normalization function is the traditional flatten function with an accumulator. It, however, corresponds to another reduction-based normalization function and a slightly different reduction strategy—though one that yields the same normal forms.

5.8 Exercises

1. Reproduce the construction above in the programming language of your choice.

6 A reduction semantics for lambda-terms with explicit substitutions

See `http://www.brics.dk/~danvy/AFP08/`.

6.1 Abstract syntax

6.2 Notion of contraction

6.3 Reduction contexts

6.4 Decomposition

6.5 One-step reduction

6.6 Reduction-based normalization

6.7 Reduction-based normalization, typefully

6.8 Summary and conclusion

6.9 Exercises

7 From reduction-based to reduction-free normalization

See `http://www.brics.dk/~danvy/AFP08/`.

7.1 Plugging and decomposition

7.2 Refocusing

7.3 From refocused evaluation function to staged abstract machine

7.4 Inlining and simplification

7.5 Refunctionalization

7.6 Back to direct style

7.7 Summary and conclusion

7.8 Exercises

8 Conclusion

There is a general consensus that normalization by evaluation is an art because one must invent a non-standard, extensional evaluation function and its left inverse [1, 7, 8, 11, 14, 20, 22, 38, 44, 47, 49, 61].

In this article, we have built on the computational content of a reduction-based normalization function as provided by a reduction semantics, and we have presented a simple, derivational way to construct a reduction-free normalization function. We have illustrated the construction on two examples, arithmetic expressions and terms in a free monoid. Elsewhere, we have successfully constructed weak-head normalization functions for the lambda-calculus (a.k.a. evaluation functions) and, in a joint work with Kevin Millikin and Johan Munk, normalization functions for the lambda-calculus (yielding long beta-eta-normal forms, when they exist), thereby establishing a link between normalization by evaluation and abstract machines for strong reduction [21, 45, 50]. Elsewhere [35, 37] We have also constructed one-pass CPS transformations, which provide an early example of normalization by evaluation.

Acknowledgments: This lecture note is a revised version of an invited talk at WRS 2004, for which the author is grateful to Sergio Antoy and Yoshihito Toyama. Thanks are also due to Rinus Plasmeijer for the opportunity to present this material at AFP 2008.

References

1. K. Aehlig and F. Joachimski. Operational aspects of untyped normalization by evaluation. *Mathematical Structures in Computer Science*, 14:587–611, 2004.
2. M. S. Ager. *Partial Evaluation of String Matchers & Constructions of Abstract Machines.* PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, Jan. 2006.
3. M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In D. Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, Aug. 2003. ACM Press.
4. M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the research report BRICS RS-04-3.
5. M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as the research report BRICS RS-04-28.
6. T. Altenkirch, M. Hofmann, and T. Streicher. Categorical reconstruction of a reduction-free normalization proof. In D. H. Pitt, D. E. Rydeheard, and P. Johnstone, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, Cambridge, UK, Aug. 1995. Springer-Verlag.
7. T. Altenkirch and T. Uustalu. Normalization by evaluation for $\lambda^{\to 2}$. In Y. Kameyama and P. J. Stuckey, editors, *Functional and Logic Programming, 7th International Symposium, FLOPS 2004*, number 2998 in Lecture Notes in Computer Science, pages 260–275, Nara, Japan, Apr. 2004. Springer-Verlag.
8. V. Balat, R. D. Cosmo, and M. P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In X. Leroy, editor, *Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 39, No. 1, pages 64–76, Venice, Italy, Jan. 2004. ACM Press.
9. V. Balat and O. Danvy. Memoization in type-directed partial evaluation. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the 2002 ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GPCE 2002*, number 2487 in Lecture Notes in Computer Science, pages 78–92, Pittsburgh, Pennsylvania, Oct. 2002. Springer-Verlag.
10. U. Berger, M. Eberl, and H. Schwichtenberg. Normalization by evaluation. In B. Möller and J. V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137, Berlin, Germany, 1998. Springer-Verlag.
11. U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In G. Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.
12. I. Beylin and P. Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95*, number

1158 in Lecture Notes in Computer Science, pages 47–61, Torino, Italy, June 1995. Springer-Verlag.

13. M. Biernacka. *A Derivational Approach to the Operational Semantics of Functional Languages*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, Jan. 2006.
14. M. Biernacka, D. Biernacki, and O. Danvy. An operational foundation for delimited continuations in the CPS hierarchy. Technical Report BRICS RS-04-29, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Dec. 2004. A preliminary version was presented at the the Fourth ACM SIGPLAN Workshop on Continuations (CW'04).
15. M. Biernacka, D. Biernacki, and O. Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, Nov. 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW'04).
16. M. Biernacka and O. Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):1–30, 2007. Article #6. Extended version available as the research report BRICS RS-06-3.
17. M. Biernacka and O. Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007. Extended version available as the research report BRICS RS-06-18.
18. D. Biernacki. *The Theory and Practice of Programming Languages with Delimited Continuations*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, Dec. 2005.
19. D. Biernacki and O. Danvy. From interpreter to logic engine by defunctionalization. In M. Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, Aug. 2003. Springer-Verlag.
20. T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.
21. P. Crégut. An abstract machine for lambda-terms normalization. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 333–340, Nice, France, June 1990. ACM Press.
22. D. Čubrić, P. Dybjer, and P. J. Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998.
23. O. Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992).
24. O. Danvy. Type-directed partial evaluation. In G. L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, Jan. 1996. ACM Press.
25. O. Danvy. Type-directed partial evaluation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.
26. O. Danvy. From reduction-based to reduction-free normalization. In S. Antoy and Y. Toyama, editors, *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, volume 124(2) of *Electronic Notes in Theoretical Computer Science*, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk.

27. O. Danvy. On evaluation contexts, continuations, and the rest of the computation. In H. Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations (CW'04)*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, Jan. 2004. Invited talk.
28. O. Danvy. A rational deconstruction of Landin's SECD machine. In C. Grelck, F. Huch, G. J. Michaelson, and P. Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, number 3474 in Lecture Notes in Computer Science, pages 52–71, Lübeck, Germany, Sept. 2004. Springer-Verlag. Recipient of the 2004 Peter Landin prize. Extended version available as the research report BRICS RS-03-33.
29. O. Danvy. *An Analytical Approach to Program as Data Objects*. DSc thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Oct. 2006.
30. O. Danvy and P. Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation (NBE 1998)*, BRICS Note Series NS-98-8, Gothenburg, Sweden, May 1998. BRICS, Department of Computer Science, University of Aarhus. Available online at <http://www.brics.dk/ nbe98/programme.html>.
31. O. Danvy and A. Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
32. O. Danvy and J. L. Lawall. Back to direct style II: First-class continuations. In W. Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press.
33. O. Danvy and K. Millikin. Refunctionalization at work. *Science of Computer Programming*, 200? In press. A preliminary version is available as the research report BRICS RS-07-7.
34. O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008.
35. O. Danvy, K. Millikin, and L. R. Nielsen. On one-pass CPS transformations. *Journal of Functional Programming*, 17(6):793–812, 2007.
36. O. Danvy and L. R. Nielsen. Defunctionalization at work. In H. Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, Sept. 2001. ACM Press. Extended version available as the research report BRICS RS-01-23.
37. O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Nov. 2004. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.
38. P. Dybjer and A. Filinski. Normalization and partial evaluation. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, Sept. 2000. Springer-Verlag.
39. M. Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, Aug. 1987.
40. M. Felleisen and M. Flatt. Programming languages and lambda calculi. Unpublished lecture notes available at <http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html> and last accessed in April 2008, 1989-2001.

41. A. Filinski. A semantic account of type-directed partial evaluation. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, Sept. 1999. Springer-Verlag. Extended version available as the research report BRICS RS-99-17.
42. A. Filinski and H. K. Rohde. A denotational account of untyped normalization by evaluation. In I. Walukiewicz, editor, *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004*, number 2987 in Lecture Notes in Computer Science, pages 167–181, Barcelona, Spain, Apr. 2002. Springer-Verlag.
43. S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In P. Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 34, No. 9, pages 18–27, Paris, France, Sept. 1999. ACM Press.
44. M. Goldberg. Gödelization in the λ-calculus. *Information Processing Letters*, 75(1-2):13–16, 2000.
45. B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In S. Peyton Jones, editor, *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, SIGPLAN Notices, Vol. 37, No. 9, pages 235–246, Pittsburgh, Pennsylvania, Sept. 2002. ACM Press.
46. T. Hardin, L. Maranget, and B. Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.
47. Y. Kinoshita. A bicategorical analysis of E-categories. *Mathematica Japonica*, 47(1):157–169, 1998.
48. S. Marlow and S. L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In K. Fisher, editor, *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, SIGPLAN Notices, Vol. 39, No. 9, pages 4–15, Snowbird, Utah, Sept. 2004. ACM Press.
49. P. Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the Third Scandinavian Logic Symposium (1972)*, volume 82 of *Studies in Logic and the Foundation of Mathematics*, pages 81–109. North-Holland, 1975.
50. C. L. McGowan. The correctness of a modified SECD machine. In *Proceedings of the Second Annual ACM Symposium in the Theory of Computing*, pages 149–157, Northampton, Massachusetts, May 1970.
51. J. Midtgaard. *Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs.* PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, June 2007.
52. K. Millikin. *A Structured Approach to the Transformation, Normalization and Execution of Computer Programs.* PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, May 2007.
53. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.
54. L. R. Nielsen. *A study of defunctionalization and continuation-passing style.* PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-7.
55. A. Ohori and I. Sasano. Lightweight fusion by fixed point promotion. In M. Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 42, No. 1, pages 143–154, New York, NY, USA, Jan. 2007. ACM Press.

56. G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Sept. 1981.
57. J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [59].
58. J. C. Reynolds. Using functor categories to generate intermediate code. In P. Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 25–36, San Francisco, California, Jan. 1995. ACM Press.
59. J. C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.
60. G. L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.
61. R. Vestergaard. The simple type theory of normalisation by evaluation. In B. Gramlich and S. Lucas, editors, *Proceedings of the First International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, number 57 in Electronic Notes in Theoretical Computer Science, Utrecht, The Netherlands, May 2001. Elsevier Science.
62. Y. Xiao, A. Sabry, and Z. M. Ariola. From syntactic theories to interpreters: Automating proofs of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.

Self-Adjusting Computation

Umut A. Acar[1]

Toyota Technological Institute
Chicago, IL, USA
umut@tti-c.org

Abstract. In these notes, I present an overview of self-adjusting computation. These notes are a draft of more comprehensive notes to be distributed later. More information and software can be found at `http://ttic.uchicago.edu/~pl/sa-sml/`.

1 Motivation

The motivation behind self-adjusting computation is that observation that incremental change is pervasive in many applications.

What I mean by this is that many application interact with data that changes slowly and by small amounts, i.e., incrementally over time. Since incremental changes are small, they often cause small changes to the output. This enables updating the output much faster than recomputing it from scratch.

Applications that exhibit this phenomena abound. For example, applications that interact with or model the physical world (e.g., robots, traffic control systems, scheduling systems) must respond to changes in the environment quickly. Similarly in applications that interact with the user, application data change incrementally over time as a result of user commands. For example, in software development, we often invoke a compiler on programs that differ only by a few lines of how big the total code base may be—in a typical use, we make small changes to the code and compile the program. Other applications where such changes are common include databases, where the database itself changes incrementally over time.

Another entirely different class of applications are those where changes are inherent to the data itself, because of motion. For example, if we wish to simulate the flow of some fluid by modeling the particles that make up the fluid itself, then we will need to compute properties of moving objects. When a computation involves moving objects, it can often be performed accurately by updating the computation only when the relationships between objects change. Since motion is continuous such changes are incremental—often few relationships change at the same time and cause small changes to the output. This makes it possible to update the output much more efficiently than recomputing from scratch.

1.1 Discrete and Continuous Changes

I describe two kinds of changes that arise naturally in the applications mentioned above: discrete/dynamic, and continuous/kinetic.

Broadly, we can classify the kinds of changes into two

Discrete/Dynamic Changes: A discrete or dynamic change is a combinatorial change to the input of the program. It changes the set of objects represented by the input.

Continuous/Kinetic Changes: A continuous or kinetic change refers to a change that affects the relationship between the input objects but does not change the set itself. By a relationship we broadly mean any function that maps a number of objects into a function whose co-domain (range) is a discreet set. For example, comparisons between objects are relationships because the co-main contains `true` and `false`.

As a simple example, let's consider sorting a list of numbers and how discrete and continuous changes may arise in this application.

Suppose that we sort a list of numbers, e.g., [30,20,10,5], and then we add a new number to the list, e.g. [**7**,30,20,10,5]. Since the change modifies the set of objects itself, it is a discrete change. Changes such as an insertion/deletion into/from the input are discrete changes regardless of the application. Note that inserting/deleting an element from the input only changes the output by a small amount as well (by essentially inserting/deleting the new element into/from the output). This makes it possible to update the output much faster than recomputing from scratch. Section 1.2 describes how this can be achieved.

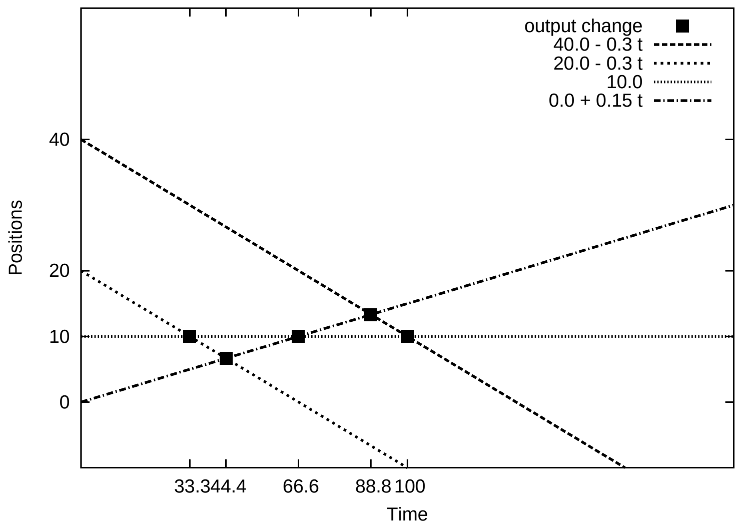


Fig. 1. Moving points.

Suppose now we want to sort a list of numbers that change continuously as a function of time. More precisely let $a(t) = 40.0 - 0.3t$, $b(t) = 10$, $c(t) = 20.0 - 0.3t$, and $d = 0.15t$. In other words, we want to start at time zero ($t = 0$), and then update the output whenever it "changes". Since the numbers are moving (i.e., changing continuously) the values in the output will change continuously but the output will only change discretely when the outcomes of the comparisons between the input objects change.

To see this, note that, at time 0, the numbers are sorted as $[a(t), b(t), c(t), d(t)]$. The output then remains the same until the time becomes is about 33.33, because only then the number $c(t)$ becomes less than $b(t)$ changing the output to $[a(t), c(t), b(t), d(t)]$. The output then remains the same until time (approximately) 44.44, when $c(t)$ falls below a, and the output becomes $[c(t), a(t), b(t), d(t)]$. The output then changes again at approximately 66.66 to $[c(t), b(t), a(t), d(t)]$, at approximately 88.88 to $[c(t), b(t), d(t), a(t)]$. The final change takes place at time 100.0, when the output becomes $[c(t), d(t), b(t), a(t)]$. Althought the output is changing continously, it is only at these time that we need to make any sort of updates to the sorting computation.

As the numbers increase and decrease over time the value of the output changes continuously but the output changes combinatorially only when the comparisons between the numbers change. Note that when the outcome of the comparison changes, the change in the output is very small—it is simply a swap of two adjacent numbers. These two properties make it possible to treat motion as a form of incremental change. In fact, in this example, by viewing the outcomes of comparisons as input to the computation, we can treat continuous changes as discrete changes to these comparisons. In general, if the computed property only depends on relationships between data whose values range over a discrete domain, then we can perform motion simulation by changing the values of these relationships as they take different values discretely.

1.2 Taking advantage of incrementality

The applications and examples described in the previous section suggest that it may be possible to update computation much faster than recomputing from scratch by taking advantage of the fact that incremental changes cause small changes to the output. How might we go about this?

One option is to customize our implementations by designing algorithms that can perform fast output updates. Another approach is to design languages for writing incremental programs in a more systematic, methodical way. Of course the second option would be nice. But neverthless let's go through an example to see the challenges with the first approach.

As a concrete example, let's look back to our sorting problem. Let's try to design a data structure that would update the sorted output efficiently under a single insertion. Considering single insertion seems to cause no loss of generality because we can express larger changes (e.g., two insertions) in terms of them. The interface for such data structure can expressed as follows

```
signature IncrementalSort =
  sig
    type 'a t
    type elt
    val insert: 'a t -> ('a * int) -> 'a t * 'a list
    val sort: elt list -> (elt * elt -> bool) -> (t * elt list)
end
```

The sort operation performs an "initial sort" of the input and generates the output list as well as a data structure (of abstract type `'a t`) that can be used to speedup the subsequent insertions. After `sort` is executed, we can change the input by inserting new elements using the `insert` operation. This operation takes the data structure and the element to insert and the position at which to insert the element and returns the updated data structure and the output list.

Now having designed the interface, let's try to give an implementation. There are several options that we can consider; these are described below from the simplest to the most sophisticated. These implementations differ in their choice of the auxiliary data structure used to speed up insertions. In the desription below, n denote the current input size.

1. The `sort` function returns the input as the auxiliary data structure. The `insert` operation simply changes the input and sorts from scratch to update the output. This is equivalent to a from-scratch execution.
2. The `sort` function returns both input and the output as the auxiliary data structure. The `insert` operation inserts the new element into the input and into the output. In this case, the `insert` operation can be performed in $O(n)$ time.
3. The `sort` function builds a binary-tree representation of the input list and returns it as the auxiliary data structure. Internally the binary tree also remembers in each node the sorted list of elements in that subtree. The `insert` operation performs an insertion into the binary search tree and constructs the output by using the sorted sublists stored in the nodes. These operations can be performed in $O(\log n)$ time by careful use of references and data structures.

Note that the first two approaches are not interesting. They only improve performance by a logarithmic factor, which is not worth the complexity—we might as well re-compute from scratch whenever there is a new input. The third approach is interesting but it does require quite a bit of algorithmic knowledge. Not only do we have to use binary search trees but we also need to construct the updated output. When the problem was more complex then sorting, designing these internal data structures become more challenging.

Perhaps one of the most important difficulties with this approach is that incremental structures are not composable. Let's try to use our sorting data structure in a larger application. Suppose that, given a list of numbers, we want to filter out those that are negative and sort the positive numbers. As the input is changed by an insertion, we want to update the output efficiently. If we did not have to worry about incremental changes, this is trivial code to write : `sort (filter (fn x => x > 0) l)`.

To develop an incremental solution to this problem, we also need an incremental filter, whose interface is given below:

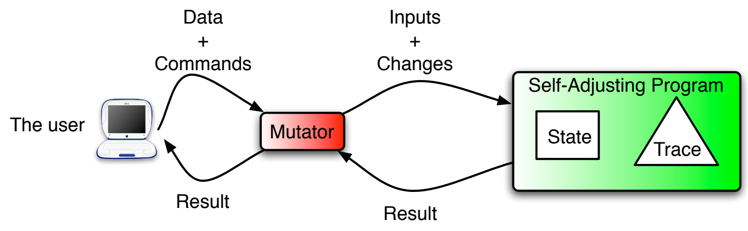


Fig. 2. Self-adjusting-computation model.

```
signature IncrementalFilter =
  sig
    type 'a t
    val filter: 'a list -> ('a -> bool) -> (t * 'a list)
    val insert: 'a list -> ('a * int) -> 'a t * 'a list
  end
```

Now we need to compose these data structures so that when we change the input to the `filter`, the change to the output of `filter` can be applied to the input to `sort`. But the `filter` function returns no information about how the output changes. Thus, we need to come up with a way to find the change in the output. This requires linear time in general. Furthermore, if the output is more complex than a list, it may not even be clear if the difference can be computed efficiently. For example, computing the difference between two graphs can be very difficult.

These examples motivate a more systematic approach to writing incremental programs; this is what self-adjusting computation offers.

2 Self-Adjusting Computation

Self-adjusting computation refers to a model of computation. In this model, programs are written using so called *modifiable references* or *modifiables*. The operations on modifiables reference are the same as those with ordinary references. In these lectures, however, we will consider a restricted interface that allows modifiables to be destructively updated only externally (not within a self-adjusting computation).

In self-adjusting computation, a program can be executed with an input (as in conventional models of computation) and, after such an execution, the contents of modifiables can be changed by destructive updates and the output can be updated by performing *change propagation*, which is provided by the model. Given a computation and a set of changed modifiables, change propagation updates the computation to incorporate the changed modifiables into the computation. Semantically, change propagation yields the same output as a from-scratch execution of the program with the change data.

In self-adjusting computation, a program typically consists of a self-adjusting function and a mutator program. Figure 2 illustrates the structure of a program in this model. The mutator program typically starts a computation by running the self-adjusting function with an input and returning the output for inspection to the user. The mutator then enters an interaction loop, where it receives commands from the user and based on these command issues changes to the data of the self-adjusting function. After the desired changes are performed, the mutator performs change propagation to update the computation, returns the updated output to the user, and jumps back to the start of the interaction loop.

This basic interaction loop of the mutator can be modified to perform more interesting tasks. For example the mutator can perform motion simulation. To simulate motion, the mutator maintains the simulation-time and performs a sweep of the time line as a part of this interaction loop. Whenever simulation-time is advanced to a time at which a continuous/kinetic change takes place, the mutator performs that change and updates the computation by running change propagation.

To respond to changes to its data efficiently, a self-adjusting program maintains a *trace*. This trace is first constructed when the self-adjusting function is executed with the user input and updated by change propagation when the data changes take place. At a high-level, the trace represents the execution of the self-adjusting function with the current data. More specifically, the trace records all modifiables and the operations on modifiables as well as the continuations at the time of dereference operations. The trace is internally represented by using *dynamic dependence graphs* or *DDGs* and function calls and continuations are memoized by using a particular form of memoization that remembers not just results of function calls also their traces. These memoized computations are re-used during change propagation by matching the arguments to memoized functions. Since modifiable references are mutable, we cannot re-use results of memoized functions directly (they may depend on mutated modifiables). Instead, we reuse the trace of a memoized computation after performing a change propagation on the trace.

As a concrete example, let's go back to our sorting example (Section 1.2. Suppose that instead of constructing an incremental sorting data structure, we write a self-adjusting version of the merge-sort algorithm, `msort`. We can write a mutator for `msort` that reads an input from the user, invokes `msort` with that input and returns the result to the user. The user can now work with this results, e.g., she can traverse the sorted output, print it. After this initial run, the mutator can enter the interaction loop, where it awaits the user to issue some changes to the input, performs such changes, and updates the output by running change propagation. Suppose again that we want to first filter the input with a provided predicate and sort the resulting list. Such a program can be trivially written by composing a self-adjusting function, `filter`, with `msort`, written `msort(filter (·))`. We can use the same mutator to enable interaction with the user. In this way, self-adjusting computation allows functions to be composed just as in standard models of computation.

Again going back to our sorting example, let's briefly discuss how we might compute the sorted output of continuously changing numbers (e.g., Figure 1).

To perform motion simulation with objects whose time-varying functions are known, we will need a priority queue for scheduling events at times at which the outcomes of comparisons change. We will start the simulation by reading the input from the user and running the `msort` function on this input. We would need to modify the `msort` function to work with a comparison function that instead of returning the outcome of the comparison directly, returns the outcome in a modifiable. In addition, the comparison function computes the time at which the value of the comparisons changes in the future, and inserts an event to the priority queue at that time. After the initial run of `msort` with the given input is complete, we can now perform motion simulation by scheduling the events in the queue. We remove the event with the earliest time, move the simulation time to that time, change the outcome of the comparison involved in the event by writing to the corresponding modifiable, and perform change propagation. We continue until no more events remain. For example, if the inputs are as shown in Figure 1, then we would perform motion simulation at the times marked with dark squares (labeled "output change").

3 Self-Adjusting SML

In this section, I describe the Self-Adjusting, adaptive ML, AML, language for writing self-adjusting program as an extension of the Standard ML language. In the rest of these notes, we will use this language for writing some examples.

In the AML language, we have two different function spaces: ordinary functions and *adaptive functions*. Adaptive functions can be thought as the functions that operate on changeable data by manipulating modifiables. For example, the primitive operations on modifiables will be adaptive functions.

Ordinary functions are declared using the conventional ML syntax for functions. Adaptive functions are declared with the `afun` and `mfun` keywords. The `afun` keyword declares an adaptive, non-memoizing function; the `mfun` keyword declares an adaptive, memoizing function. Adaptive functions have the adaptive-function type τ `-$>` τ. We use the infix operator `$` for applying adaptive (memoizing, or non-memoizing) functions. An adaptive function can perform all the operations that ordinary functions can. In addition, adaptive functions can call other adaptive functions.

We require an adaptive application to appear only in the body of an adaptive function—an adaptive application cannot appear in the body of a normal function. Because of this requirements, we can partition a self-adjusting program into a set of adaptive functions that call each other and other normal functions, and a set of normal functions that can only call other normal functions. This distinction helps us enforce the distinction between the mutator and a self-adjusting program. It also helps improve the efficiency of the compilation mechanisms and the compiled code, which we will not discuss here.

```
signature ADAPTIVE = sig
 type 'a box
 val put : 'a -$> 'a box
 val get : 'a box -$> 'a

 val putTh : (unit -$> 'a) -> 'a box
 val mkPut : unit -$> ('k * 'a -$> 'a box)

 (** Meta operations **)
 val new : 'a -> 'a box
 val deref : 'a box -> 'a
 val change : 'a box * 'a -> unit

 datatype 'a res = Value of 'a | Exn of exn
 val call : ('a -$> 'r) * 'a -> 'r res ref
 val propagate : unit -> unit
end
structure Adaptive :> ADAPTIVE = struct ... end
```

Fig. 3. Signature for the `Adaptive` library.

3.1 The Primitives

Except for `afun` and `mfun` keywords, all the rest of the self-adjusting-computation primitives are provided by a library. Figure 3 shows the interface to this library. The *box type* τ `box` is the type of a modifiable reference and serves as a container for changeable data. The `put:` α `-$>` α `box` primitive places a value into a box, while the `get:` α `box -$>` α primitive returns the contents of a box. Since the primitives have adaptive function types, they may only be used within an adaptive function.

The `put` and `get` function are all we need to operate on modifiable references. For the purposes of improving efficiency, however, the library provides two additional facilities: `putTh` and `mkPut`. The `putTh` operation receives as an argument a computation (a thunk), evaluates the computation, and places its value into a box. The `mkPut` operation returns a *putter* that can be used to perform allocations based on a key. The putter takes a key and a value to be boxed and returns a box holding that value.

When programming with these primitives, we will use `put` and `get` operations initially. We will then replace the put operations with `putTh` operations and calls to putters created by `mkPut` for improved performance.

To facilitate the mutator to drive self-adjusting computation and to operate on changeable data we provide number of *meta primitives*. These meta primitives are "impure" and should not be used within adaptive functions; doing otherwise can violate the correctness of change propagation.

The `new`, `change`, and `deref` operations are used by the mutator to create & modify inputs, and inspect outputs of a self-adjusting computation. The `new` operation places a value into box—it is the meta-version of the `put` operation. The `deref` operation returns the contents of a box—it is the meta-version of the `get` operation. The `change` operation takes a value and a box and changes the contents of the box to the given value by a destructive update.

The `call` and `propagate` primitives enable starting an adaptive computation and performing change propagation. At the meta-level the result of an adaptive computation is either an ordinary value or an exception. The result is an exception if the adaptive function raises an exception before terminating. More concretely, the `call` operation takes an adaptive function and an argument to that function, calls the function, and returns its results in an ordinary reference. Note, that the `call` operation is the *only* means of "applying" an adaptive function outside the body of another adaptive function. The result of the `call` operation is a mutable reference cell containing the output (distinguishing between normal and exceptional termination) of the self-adjusting computation. The `propagate` operation can be applied anytime. It incorporates the effects of the `change` operations executed since the beginning of the computation or the last call to `propagate`.

4 Adaptive Lists and Examples

In these notes, we will consider a number of examples with adaptive lists.

```
signature LIST_ADAPTIVE =
 sig
  datatype 'a u = NIL | CONS of 'a * 'a u box
  type 'a t = 'a u box

  val lengthLessThan: int -> 'a t -$> bool box
  val map: ('a -> 'b) -> 'a t -$> 'b t
  val partition: ('a -> bool) -> 'a t -$> 'a t * 'a t
  val merge: ('a * 'a -> bool) -> ('a t * 'a t) -$> 'a t
  val msort: ('a * 'a -> bool) -> 'a t -$> 'a t
 end

structure AdaptiveList:LIST_ADAPTIVE
```

Fig. 4. The interface for adaptive lists and the `AdaptiveList` structure.

Figure 4 shows the interface to our adaptive-list library. An adaptive list of type `'a t` is a recursive data type that is either empty (`NIL`) or is a cons cell containing an element of type `'a` and boxed lists. An adaptive list is defined in a way that is analogous linked lists: boxed tails can be viewed as "next" pointer in a linked list implementation. Boxing the tails allows us to insert/delete element into/from the list by changing the contents of boxed tails.

As examples, we will consider writing three simple primitives: `lengthLessThan`, `map`, and `reverse`. The `lengthLessThan` function takes an integer and returns a boxed boolean indicating whether the length of the list is less than the supplied integer. The `map` function takes a function and a list and maps the list to another by applying the function to its elements. The `reverse` function returns the reverse of a given list. Figure 5 shows the code for the ordinary and the adaptive versions of the list primitives. To obtain the adaptive versions from the ordinary versions, we insert the underlined pieces of syntax.

```
fun lengthLessThan
    (l: 'a list)
    : bool =
  let
    fun len (i,l) =
      if i >= n then
        false
      else
        case l of
          nil => true
        |  cons(h,t) => len (i+1,t)
  in
    len(0,l)
  end
```

```
afun lengthLessThan
    (l: 'a ListAdaptive.t)
    : bool box =
  let
    mfun len (i,l) =
      if i >= n then
        false
      else
        case get $ l of
          NIL => true
        |  CONS(h,t) => len $ (i+1,t)
  in
    put $ (len $ (0,l))
  end
```

```
fun map
    (f: 'a -> 'b)
    (l: 'a list)
    : 'b list =
  let
    fun m l =
      case l of
        nil => nil
      | cons (h,t) => cons(f h, m t)
  in m l end
```

```
afun map
    (f: 'a -> 'b)
    (l: 'a ListAdaptive.t)
    : 'b ListAdaptive.t=
  let
    mfun m l =
      case get $ l of
        NIL => put $ NIL
      | CONS(h,t) => put (CONS (f h, m t))
  in m l end
```

```
fun reverse
    (l: 'a list)
    : 'b list =
  let
    fun rev (l,acc) =
      case l of
        nil =>
          acc
      |  cons(h,t) =>
        rev (t,cons(h,acc))
  in
    rev (l,[])
  end
```

```
afun reverse
    (l: 'a AdaptiveList.t)
    : 'b AdaptiveList.t =
  let
    mfun rev (l,acc) =
      case get $ l of
        nil =>
          acc
      |  cons(h,t) =>
          rev (t, put $ (cons(h, acc)))
  in
    rev (l,put $ [])
  end
```

Fig. 5. Some list primitives, ordinary (left) and adaptive (right).

5 An Adaptive-List Mutator

In our examples, we are interested in changing the inputs with insertions and deletions. Figure 6 shows the code for such a mutator for adaptive-list applications. The `mutator` function takes a `toString` function for printing the input and the output lists, an adaptive function `f`, and an input list `l`.

The `mutator` function defines a few utility functions for converting between ordinary lists and adaptive lists, and deleting and inserting elements from/into adaptive lists.

The `mutator` starts by converting the input list to an adaptive list and calling the supplied function to the converted list. It then calls `checkAndPrintResult` on the result of the adaptive call. This function checks that the adaptive function returned a proper result (if an exception was raised than it is re-raised). If a non-exception value is returned, then the function prints the returned result. After checking the result from `mutator` deletes the first element in the lists, performs change propagation, and prints the result. It then inserts the element back into the list, performs a change propagation and prints the result.

```
structure ListAdaptive:LIST_ADAPTIVE =
   struct ... end

fun mutator
    (toString: 'a list -> string)
    (f: 'a alist -$> 'a alist)
    (l: 'a list) : unit =
let
   fun toList (l: 'a ListAdaptive.t) : 'a list =
      case (Adaptive.deref l) of
         NIL => nil
      |  CONS(h,t) => h::(toList t)

   fun fromList (l: 'a list) : 'a ListAdaptive.t =
      case l of
         nil => box NIL
      |  cons(h,t) => box (CONS(h, fromList t)

   fun checkAndprintResult r =
      case !r of
         Exn e => raise e
      |  Value v => let val s = toString (toList v)
                    in print (``Result ='' ^  s ^  ``\n'')

   fun delete cl =
      case (Adaptive.deref cl) of
         NIL => (NONE)
      |  CONS(h,t) => (Adaptive.change (cl, t); h)

   fun insert h cl =
      case h of
         NONE => ()
      |  SOME v => (Adaptive.change cl (CONS (v,box (Adaptive.deref cl))))

   val cl = fromList l
   val r = Adaptive.call (f, cl)
   val _ = checkAndPrintResult r

   val h = delete cl
   val _ = Adaptive.propagate ()
   val _ = checkAndPrintResult r

   val _ = insert h cl
   val _ = Adaptive.propagate ()
   val _ = checkAndPrintResult r
in () end
```

Fig. 6. An example mutator for inserting/deleting the first element of a list.

6 Performance of Change Propagation

I have described how to translate ordinary, purely functional programs into self-adjusting programs. The purpose of this conversion is to take advantage of the ability to update the output of computations much faster than recomputing from scratch by using change propagation. There is some more work that we need to do to ensure efficient change propagation. In this section, we identify two *stability-properties* and make our examples adhere to these properties. These properties ensure the stability of self-adjusting programs, i.e., that they propagate changes minimally.

Stability Property I: Result Stability. Let $f(L)$ be an application of a self-adjusting function f. We want $f(L)$ to return a modifiable, and always the same modifiable, even if its parts are re-executed during change propagation. The only exception to this property happens when the result of f is always placed into a modifiable by the caller.

Stability Property II: Output Stability. Consider some ordinary list function f and consider its execution with two lists L and L' where L' is obtained from L by inserting one new element. Define the *distance* between the outputs to be length of the output list $f(L')$ minus the longest common subsequence of $f(L)$ and $f(L')$. If we run the self-adjusting version f_a of f with the adaptive-list versions of L_a and L'_a, then we want the edit distance between $f_a(L_a)$ and $f_a(L'_a)$ to be d including the modifiable references in the outputs. In other words, we do not want the output of self-adjusting programs to change any more than the ordinary version even when including reference.

Ensuring the stability properties. To ensure these properties, we will rewrite our programs by replacing the `put` primitives with `putTh`'s and with putter functions. We will consider each function in turn. Figure 7 shows the for the modified list primitives.

For the discussions, I would like to note a few facts about change propagation. First note that change-propagation may re-execute a part of the computation starting at any read. Second, when the expression `e` in `put e`, i.e., the body of a `put`, is re-executed, the modifiable allocated by `put` may change. If we wish the modifiable remain the same (i.e., be pre-allocated before the expression is evaluated), then we will need to use `putTh`.

If the output of a function is another list, we will make sure that the modifiables in the output remain the same during change propagation by using a *putter* to allocate the modifiables. A putter takes two arguments. It allocates a modifiable and writes the value of the second argument to the modifiable. It names the modifiable with the first argument. If a putter is called with the same name, then it will allocate the same modifiable unless there is another modifiable with the same name. Throughout we will name the tail modifiables of cons cells by the heads of cons cells and assume that a lists contains no duplicates. This uniquely identifies the modifiables by the head item in the same cons cell: after change propagation, a cell that holds the same element of the input has the same modifiable as before the change propagation.

Consider the function `lengthLessThan`. The function returns a modifiable that holds the result. The first property requires that this modifiable be the same even if we start executing somewhere in the middle of the `len` function. To ensure this, we will use the function `putTh` to create the modifiable instead of `put`. Since this is the only modifiable allocated by `lengthLessThan`, the second property holds trivially.

Consider the function `map`. We will make sure that a modifiable tail of a cons cell holding a particular element in the output remains the same by naming the allocated tail by the element. To this end, we first create a `putter` by using the `mkPut` function. We then use `SOME h` as a key when allocating the tail of the cons cell holding `h`. This ensures the second property, if we insert a new element and perform change propagation, we will get only one new modifiable. How about the very first modifiable? To fix that modifiable, we simply use `NONE` as a name for it.

In function `reverse`, we use essentially the same idea as in `map`.

For all these functions, note that a putter is created every time we enter the function from the top level. It is only within a recursive chain of calls that the modifiables are being named. Two different calls to these functions will have different putters and will not share the output modifiables.

7 Merge Sort

Figure 8 shows the code for sorting a list using the merge-sort algorithm. The merge-sort algorithm sorts the input by partitioning the input list into two lists of equal size, recursively sorting them, and merging the sorted sublists. This partition operation, when performed deterministically, does not behave well under change propagation, because for example inserting a new element can change the partitioned lists dramatically. For example, if we first partition the list `[0,1,3,4,5]`, the two lists will be `[0,3,5]` and `[1,4]`. If we now insert `2` and partition `[0,1,2,3,4,5]`, then we will get `[0,2,4]` and `[1,3,5]`, both of which are now different than the two sorted lists before.

To address this problem, we will use randomization. Instead of performing the partition deterministically, we will flip a coin for each element and collect those elements that come up heads into one list and the other into another. Going back to the example, we may split `[0,1,3,4,5]`, into `[0,3,5]` and `[1,4]` by flipping, heads, tails, heads, tails, and heads. If we now insert `2` and partition, we will get `[0,3,5]` and `[1,2,4]`, if we flip a tail for `2`. For this to work, it will be important that we flip the same coins for the same elements whenever we repeat a partitioning. To do this, we will use random hash functions instead of random coins.

To satisfy the two stability properties, we use a putter for `merge` as with `map` and `reverse`. To make sure that `msort` always returns the same modifiable as the head of the list, we allocate its result with `putTh`.

8 Exercise

Write the code for `partition` function used in `msort`.

```
fun lengthLessThan
    (l: 'a list)
    : bool =
  let
    fun len (i,l) =
      if i >= n then
        false
      else
        case l of
          nil => true
        |  cons(h,t) => len (i+1,t)
  in
    len(0,l)
  end


fun map
    (f: 'a -> 'b)
    (l: 'a list)
    : 'b list =
  let
    fun m l =
      case l of
        nil => nil
      | cons (h,t) => cons(f h, m t)
  in m l end


fun reverse
    (l: 'a list)
    : 'b list =
  let
    fun rev (l,acc) =
      case l of
        nil =>
          acc
      |  cons(h,t) =>
        rev (t,cons(h,acc))
  in
    rev (l,[])
  end
```

```
afun lengthLessThan
    (l: 'a ListAdaptive.t)
    : bool box =
  let
    afun len (i,l) =
      if i >= n then
        false
      else
        case get $ l of
          NIL => true
        |  CONS(h,t) => len $ (i+1,t)
  in
    putTh $ (afn () => len $ (0,l))
  end


afun map
    (f: 'a -> 'b)
    (l: 'a ListAdaptive.t)
    : 'b ListAdaptive.t=
  let
    val putter = mkPut ()
    mfun m l =
      case get $ l of
        NIL => NIL
      | CONS(h,t) => putter $ (SOME h, CONS (f h, m t))
  in putter $ (NONE, m l) end


afun reverse
    (l: 'a AdaptiveList.t)
    : 'b AdaptiveList.t =
  let
    val putter = mkPut ()
    mfun rev (l,acc) =
      case get $ l of
        nil =>
          putter $ (NONE,acc)
      |  cons(h,t) =>
          rev (t, cons(h, putter $ (SOME h, acc)))
  in
    rev (l,nil))
  end
```

Fig. 7. Some list primitives, ordinary (left) and adaptive (right).

```
structure BinaryHashFamily =
  struct
    fun new () = ....
  end

afun split l =
  let
    val hashFun = BinaryHashFamily.new ()
    fun randFun x = hashFun (Adaptive.hash x) = 0
  in
    AdaptiveList.partition randFun $ l
  end

afun merge lt (a,b) =
  let
    val putter = mkPut ()
    mfun mmerge (a,b) =
      case (get $ a, get $ b) of
        (NIL,b) => b
      | (a,NIL) => a
      | (CONS(ha,ta),CONS(hb,tb)) =>
          if lt (ha,hb) then
            CONS(ha, putter $ (SOME ha, mmerge(ta,b)))
          else
            CONS(hb, putter $ (SOME hb, mmerge(a,tb)))
  in
    putter $ (NONE, mmerge (a,b))
  end

fun msort lt l =
  putTh (afn () =>
    if lengthLesThan $ 2 $ l then
      get $ l
    else
      let
        val (even,odd) = split $ l
        val evens = msort $ even
        val odds = msort $ odd
        val r = merge $ lt $ (evens,odds)
      in
        get $ r
      end))
```

Fig. 8. The adaptive merge-sort function.

A Tutorial on Parallel and Concurrent Programming in Haskell (DRAFT)

Simon Peyton Jones and Satnam Singh

Microsoft Research Cambridge
`simonpjmicrosoft.com satnamsmicrosoft.com`

Abstract. This tutorial introduces parallel programming in Haskell.

1 Introduction

The introduction of multi-core processors has renewed interest in parallel functional programming and there are now several interesting projects that explore the advantages of a functional language for writing parallel code or implicitly paralellizing code written in a pure functional language. These lecture notes present a variety of techniques for writing concurrent parallel programs which include existing techniques based on semi-implicit parallelism and explicit thread-based parallelism as well as more recent developments in the areas of software transactional memory and nested data parallelism.

We also use the terms *parallel* and *concurrent* with quite specific meanings. A parallel program is one which is written for performance reasons to exploit the potential of a real parallel computing resource like a multi-core processor. For a parallel program we have the expectation of some genuinely simultaneous execution. Concurrency is a software structuring technique that allows us to model computations as hypothetical independent activities (e.g. with their own program counters) chat can communicate and synchronize.

In these lecture notes we assume that the reader is familiar with the pure lazy functional programming language Haskell.

2 Applications of concurrency and parallelism

Writing concurrent and parallel programs is more challenging than the already difficult problem of writing sequential programs. However, there are some compelling reasons for writing concurrent and parallel programs:

Performance. We need to write parallel programs to achieve improving performance from each new generation of multi-core processors.

Hiding latency. Even on single-core processors we can exploit concurrent programs to hide the latency of slow I/O operations to disks and network devices.

Software structuring. Certain kinds of problems can be conveniently represented as multiple communicating threads which help to structure code in a more modular manner e.g. by modeling user interface components as separate threads.

Real world concurrency. In distributed and real-time systems we have to model and react to events in the real world e.g. handling multiple server requests in parallel.

All new mainstream microprocessors have two or more cores and relatively soon we can expect to see tens or hundreds of cores. We can not expect the performance of each individual core to improve much further. The only way to achieve increasing performance from each new generation of chips is by dividing the work of a program across multiple processing cores. One way to divide an application over multiple processing cores is to somehow automatically parallels the sequential code and this is an active area of research. Another approach is for the user to write a semi-explicit or explicitly parallel program which is then scheduled onto multiple cores by the operating systems and this is the approach we describe in these lectures.

3 Compiling Parallel Haskell Programs

To compile a parallel Haskell program you need to specify the `-threaded` extra flag. For example, to compile the parallel program contained in the file `Wombat.hs` issue the command:

```
ghc --make -threaded Wombat.hs
```

To execute the program you need to specify how many real threads are available to execute the logical threads in a Haskell program. This is done by specifying an argument to Haskell's run-time system at invocation time. For example, to use three real threads to execute the `Wombat` program issue the command:

```
Wombat +RTS -N3
```

In these lecture notes we use the term *thread* to describe a Haskell thread rather than a native operating system thread.

4 Semi-Explicit Parallelism

A pure Haskell program may appear to have abundant opportunities for automatic parallelization. Given the lack of side effects it may seem that we can productively evaluate every sub-expression of a Haskell program in parallel. In practice this does not work out well because it creates far too many small items of work which can not be efficiently scheduled and parallelism is limited by fundamental data dependencies in the source program.

Haskell provides a mechanism to allow the user to control the granularity of parallelism by indicating what computations may be usefully carried out in parallel. This is done by using functions from the Control.Parallel module. The interface for Control.Parallel is shown below:

```
par :: a -> b -> b
pseq :: a -> b -> b
```

The function par indicates to the Haskell run-time system that it may be beneficial to evaluate the first argument in parallel with the second argument. The par function returns as its result the value of the second argument. One can always eliminate par from a program by using the following identity without altering the semantics of the program:

```
par a b = b
```

A thread is not necessarily created to compute the value of the expression a. Instead, the Haskell run-time system creates a *spark* which has the potential to be executed on a different thread from the parent thread. A sparked computation expresses the possibility of performing some speculative evaluation. Since a thread is not necessarily created to compute the value of a this approach has some similarities with the notion of a *lazy future* [1].

Sometimes it is convenient to write a function with two arguments as an infix function and this is done in Haskell by writing quotes around the function:

```
a ‘par‘ b
```

An example of this expression executing in parallel is shown in Figure1.

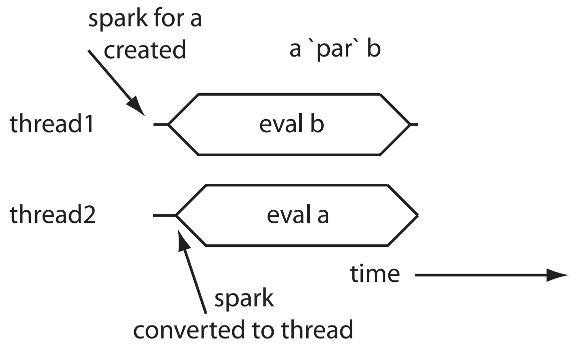


Fig. 1. Semi-explicit execution of a in parallel with the main thread b

We call such programs semi-explicitly parallel because the programmer has provided a hint about the appropriate level of granularity for parallel operations and the system implicitly creates threads to implement the concurrency. The user does not need to explicitly create any threads or write any code for inter-thread communication or synchronization.

To illustrate the use of par we present a program that performs two compute intensive functions in parallel. The first compute intensive function we use is the notorious Fibonacci function:

```haskell
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

The second compute intensive function we use is the sumEuler function taken from [2]:

```haskell
mkList :: Int -> [Int]
mkList n = [1..n-1]

relprime :: Int -> Int -> Bool
relprime x y = gcd x y == 1

euler :: Int -> Int
euler n = length (filter (relprime n) (mkList n))

sumEuler :: Int -> Int
sumEuler = sum . (map euler) . mkList
```

The function that we wish to parallelize adds the results of calling fib and sumEuler:

```haskell
sumFibEuler :: Int -> Int -> Int
sumFibEuler a b = fib a + sumEuler b
```

As a first attempt we can try to use par the speculatively spark off the computation of fib while the parent thread works on sumEuler:

```haskell
parSumFibEuler :: Int -> Int -> Int
parSumFibEuler a b
  = f `par` (f + e)
    where
    f = fib a
    e = sumEuler b
```

To help measure how long a particular computation is taking we use the Sytem.Time module and define a function that returns the difference between two time samples as a number of seconds:

```haskell
secDiff :: ClockTime -> ClockTime -> Float
secDiff (TOD secs1 psecs1) (TOD secs2 psecs2)
  = fromInteger (psecs2 - psecs1) / 1e12 + fromInteger (secs2 - secs1)
```

The main program calls the parSumFibEuler function with suitably large arguments and reports the value

```haskell
r1 :: Int
r1 = sumFibEuler 40 7450

```

```
main :: IO ()
main
  = do t0 <− getClockTime
       pseq r1 (return ())
       t1 <− getClockTime
       putStrLn ("sum:␣" ++ show r1)
       putStrLn ("time:␣" ++ show (secDiff t0 t1) ++ "␣seconds")
```

The calculations fib 40 and sumEuler 7450 have been chosen to have roughly the same execution time.

If we were to execute this code using just one thread we would observe the sequence of evaluations shown in Figure 2. Although a spark is created for the evaluation of f there is no other thread available to instantiate this spark so the program first computes f (assuming the + evaluates its left argument first) and then computes e and finally the addition is performed. Making an assumption about the evaluation order of the arguments of + is unsafe and another valid execution trace for this program would involve first evaluating e and then evaluating f.

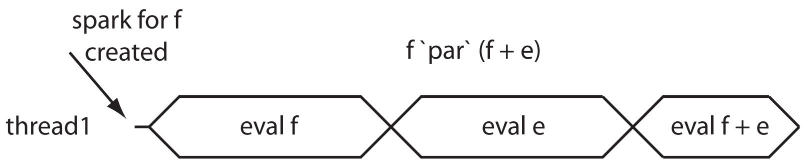


Fig. 2. Executing f ‘par‘ (e + f) on a single thread

The compiled program can now be run on a multi-core computer and we can see how it performs when it uses one and then two real operating system threads:

```
$ ParSumFibEuler +RTS -N1
sum: 119201850
time: 21.534 seconds

$ ParSumFibEuler +RTS -N2
sum: 119201850
time: 21.462 seconds
```

The output above shows that the version run with two cores did not perform any better than the sequential version. Why is this? The problem lies in line 3 of the parSumFibEuler function. Although the work of computing fib 40 is sparked off for speculative evaluation the parent thread also starts off by trying to compute fib 40 because this particular implementation of the program used a version of + that evaluates its left and side before it evaluates its right hand side. This causes the main thread to *demand* the evaluation of fib 40 so the spark never gets instantiated onto a thread. After the main thread evaluates fib 40 it goes onto evaluate sumEuler 7450 which results in a performance which is equivalent to the sequential program. A sample execution trace for this version of the program is shown in Figure 3.

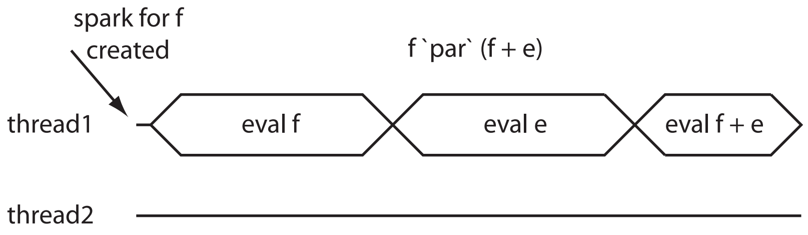


Fig. 3. A spark that does not get instantiated onto a thread

A tempting fix is to reverse the order of the arguments to +:

```
parSumFibEuler :: Int -> Int -> Int
parSumFibEuler a b
  = f `par` (e + f)
    where
    f = fib a
    e = sumEuler b
```

Here we are sparking off the computation of fib for speculative evaluation with respect to the parent thread. The parent thread starts off by computing sumEuler and hopefully the run-time will convert the spark for computing fib and execute it on a thread located on a different core in parallel with the parent thread. This does give a respectable speedup:

```
$ ParFibSumEuler +RTS -N1
sum: 119201850
time: 21.832 seconds

$ ParFibSumEuler +RTS -N2
sum: 119201850
time: 12.233 seconds
```

A sample execution trace for this version of the program is shown in Figure 4

However, it is a Very Bad Idea to rely on the evaluation order of + for the performance (or correctness) of a program. The Haskell language does not define the evaluation order of the left and right hand arguments of + and the compiler is free to transform a + b to b + a. What we really need to be able to specify what work the main thread should do first. We can use the pseq function from the Control.Monad module for this purpose. The expression a `pseq` b evaluates a and then returns b. We can use this function to specify what work the main

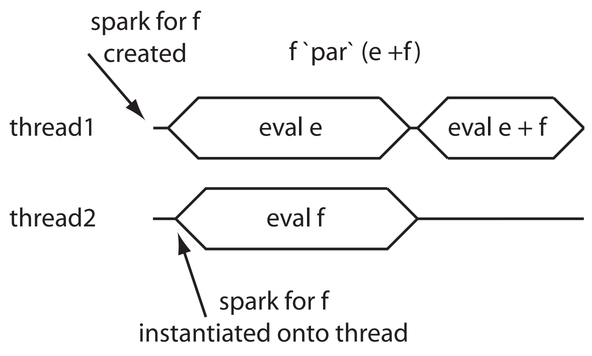


Fig. 4. A lucky parallelization (bad dependency on the evaluation order of +)

thread should do first (as the first argument of pseq) and we can then return the result of the overall computation in the second argument without worrying about things like the evaluation order of +. This is how we can re-write ParFibSumEuler with pseq:

```
parSumFibEuler :: Int -> Int -> Int
parSumFibEuler a b
  = f `par` (e `pseq` (e + f))
    where
    f = fib a
    e = sumEuler b
```

This program still gives a roughly 2X speedup as does the following version which has the arguments to + reversed but the use of pseq still ensures that the main thread works on sumEuler before it computes fib (which will hopefully have been computed by a speculatively created thread):

```
parSumFibEuler :: Int -> Int -> Int
parSumFibEuler a b
  = f `par` (e `pseq` (f + e))
    where
    f = fib a
    e = sumEuler b
```

An execution trace for this program is shown in Figure 5.

4.1 Weak Head Normal Form (WHNF)

The program below is a variant of the fib-Euler program in which each parallel workload involves mapping an operation over a list.

```
module Main
where
```

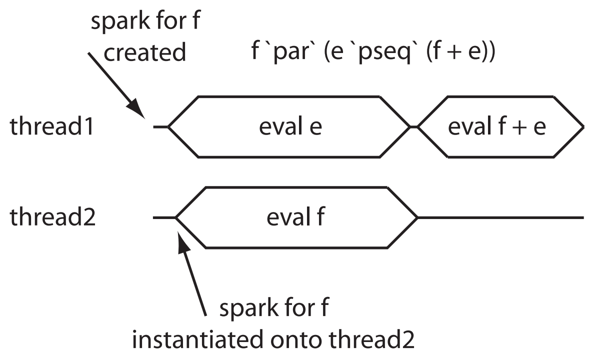

spark for f created
f `par` (e `pseq` (f + e))
thread1
eval e
eval f + e
thread2
eval f
spark for f instantiated onto thread2

Fig. 5. A correct parallelization which is not dependent on the evaluation order of +

```
import System.Time
import Control.Parallel

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

mapFib :: [Int]
mapFib = map fib [37, 38, 39, 40]

mkList :: Int -> [Int]
mkList n = [1..n-1]

relprime :: Int -> Int -> Bool
relprime x y = gcd x y == 1

euler :: Int -> Int
euler n = length (filter (relprime n) (mkList n))

sumEuler :: Int -> Int
sumEuler = sum . (map euler) . mkList

mapEuler :: [Int]
mapEuler = map sumEuler [7600, 7600]

parMapFibEuler :: Int
parMapFibEuler = mapFib `par`
                 (mapEuler `pseq` (sum mapFib + sum mapEuler))

main :: IO ()
```

```
main
  = putStrLn (show parMapFibEuler)
```

The intention here is that the computation that involves mapping the fib operation over a list occurs in parallel with the computation of in the main thread of the operation that maps sumEuler over a list and sums the two lists. We have chosen arguments which result in a similar run-time for mapFib and mapEuler.

However, when we run this program with one and then two cores we observe no speedup:

```
satnams@MSRC-LAGAVULIN ~/papers/afp2008/whnf
$ time WHNF2 +RTS -N1
263935901

real    0m48.086s
user    0m0.000s
sys     0m0.015s

satnams@MSRC-LAGAVULIN ~/papers/afp2008/whnf
$ time WHNF2 +RTS -N2
263935901

real    0m47.631s
user    0m0.000s
sys     0m0.015s
```

What went wrong? The problem is that the function mapFib does not return a list with four values each fully evaluated to a number. Instead, the expression is reduced to weak head normal form which only return the top level cons cell with the head and the tail elements unevaluated as shown in Figure 6. This means that almost no work is done in the parallel thread.

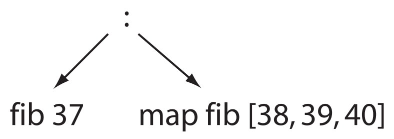


Fig. 6. parFib evaluated to weak head normal form (WHNF)

To fix this problem we need to somehow force the evaluation of the list. We can do this by defining a function that iterates over each element of the list and then uses each element as the first argument to pseq which will cause it to be evaluated to a number:

```
forceList :: [a] -> ()
forceList [] = ()
forceList (x:xs) = x `pseq` forceList xs
```

Using this function we can express our requirement to evaluate the the mapFib function fully to a list of numbers rather than to just weak head normal form:

```
module Main
where
import Control.Parallel

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

mapFib :: [Int]
mapFib = map fib [37, 38, 39, 40]

mkList :: Int -> [Int]
mkList n = [1..n-1]

relprime :: Int -> Int -> Bool
relprime x y = gcd x y == 1

euler :: Int -> Int
euler n = length (filter (relprime n) (mkList n))

sumEuler :: Int -> Int
sumEuler = sum . (map euler) . mkList

mapEuler :: [Int]
mapEuler = map sumEuler [7600, 7600]

parMapFibEuler :: Int
parMapFibEuler = (forceList mapFib) `par`
                 (forceList mapEuler `pseq` (sum mapFib + sum mapEuler))

forceList :: [a] -> ()
forceList [] = ()
forceList (x:xs) = x `pseq` forceList xs

main :: IO ()
main
  = putStrLn (show parMapFibEuler)
```

This gives the desired performance which shows the work of mapFib is done in parallel with the work of mapEuler:

```
satnams@MSRC-LAGAVULIN ~/papers/afp2008/whnf
$ time WHNF3 +RTS -N1
263935901

real    0m47.680s
```

```
user    0m0.015s
sys     0m0.000s

satnams@MSRC-LAGAVULIN ~/papers/afp2008/whnf
$ time WHNF3 +RTS -N2
263935901

real    0m28.143s
user    0m0.000s
sys     0m0.000s
```

Question. What would be the effect on performance if we omitted the call of forceList on mapEuler?

An important aspect of how pseq works is that it evaluates its first argument to weak head normal formal. This does not fully evaluate an expression e.g. for an expression that constructs a list out of a head and a tail expression (a CONS expression) pseq will not evaluate the head and tail sub-expressions.

Haskell also defines a function called seq but the compiler is free to swap the arguments of seq which means the user can not control evaluation order. The compiler has primitive support for pseq and ensures the arguments are never swapped and this function should always be preferred over seq for parallel programs.

4.2 Divide and conquer

Exercise 1: Parallel quicksort. The program below shows a sequential implementation of a quicksort algorithm. Use this program as a template to write a parallel quicksort function. The main body of the program generates a pseudo-random list of numbers and then uses measures the time taken to build the input list and then to perform the sort and then add up all the numbers in the list.

```haskell
module Main
where
import System.Time
import Control.Parallel
import System.Random

-- A sequential quicksort
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = losort ++ x : hisort
                   where
                   losort = quicksort [y | y <- xs, y < x]
                   hisort = quicksort [y | y <- xs, y >= x]

secDiff :: ClockTime -> ClockTime -> Float
secDiff (TOD secs1 psecs1) (TOD secs2 psecs2)
  = fromInteger (psecs2 - psecs1) / 1e12 + fromInteger (secs2 - secs1)
```

```

main :: IO ()
main
  = do t0 <- getClockTime
       let input = (take 20000 (randomRs (0,100) (mkStdGen 42)))::[Int]
       seq (forceList input) (return ())
       t1 <- getClockTime
       let r = sum (quicksortF input)
       seq r (return ()) -- Force evaluation of sum
       t2 <- getClockTime
       -- Write out the sum of the result.
       putStrLn (show r)
       -- Write out the time taken to build the input list.
       putStrLn (show (secDiff t0 t1))
       -- Write out the time taken to perform the sort.
       putStrLn (show (secDiff t1 t2))
```

The current version of GHC does not have a parallel garbage collector so some parallel programs have poor performance because all parallel execution is suspended during sequential garbage collection. This problem can be partly mitigated by running a Haskell program with a large heap to reduce the amount of garbage collection. The size of the heap is specified has an argument to the run-time system e.g. -H800M means use a 800MB heap.

```
satnams@msrc-bensley /cygdrive/l/papers/afp2008/quicksort
$ QuicksortD +RTS -N1 -H800M
Sum of sort: 50042651196
Time to sort: 4.593779

satnams@msrc-bensley /cygdrive/l/papers/afp2008/quicksort
$ QuicksortD +RTS -N2 -H800M
Sum of sort: 50042651196
Time to sort: 3.171895
```

You can find get some idea of how well a program has been parallelized and how much time is taken up with garbage collection by using the runtime -S flag to dump some statistics to a file:

```
QuicksortD.exe +RTS -N2 -H300M -Sn2.txt
```

After execution you can look at the end of the file n2.txt to see what happened:

```
1,488,789,968 bytes allocated in the heap
203,137,628 bytes copied during GC (scavenged)
 64,078,196 bytes copied during GC (not scavenged)
 66,813,952 bytes maximum residency (4 sample(s))

         12 collections in generation 0 (  0.73s)
          4 collections in generation 1 (  0.72s)
```

```
      325 Mb total memory in use

  Task  0 (worker) :  MUT time:   1.53s  (  3.03s elapsed)
                      GC  time:   0.41s  (  0.41s elapsed)

  Task  1 (worker) :  MUT time:   0.00s  (  3.04s elapsed)
                      GC  time:   0.00s  (  0.00s elapsed)

  Task  2 (worker) :  MUT time:   2.89s  (  3.04s elapsed)
                      GC  time:   1.05s  (  1.13s elapsed)

  Task  3 (worker) :  MUT time:   0.00s  (  3.04s elapsed)
                      GC  time:   0.00s  (  0.00s elapsed)

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    4.41s  (  3.04s elapsed)
  GC    time    1.45s  (  1.53s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    5.87s  (  4.57s elapsed)

  %GC time      24.7%  (33.5% elapsed)

  Alloc rate    337,224,832 bytes per MUT second

  Productivity  75.3% of total user, 96.5% of total elapsed
```

This execution of quicksort spent 24.7% of its time in garbage collection which is performed sequentially. The work of the sort was shared out amongst two threads (task 0 and task 2) although not evenly. The `MUT` time gives an indication of how much time was spent performing computation.

Writing semi-implicitly parallel programs can sometimes help to parallelize pure functional programs but it does not work when we want to parallelize stateful computations in the IO monad. For that we need to write explicitly threaded programs.

5 Explicit Concurrency

In this section we introduce Haskell's mechanisms for writing explicitly concurrent programs. Haskell presents explicit concurrency features to the programmer via a collection of library functions rather than adding special syntactic support for concurrency and all the functions presented in this section are exported by this module.

5.1 Creating Haskell Threads

The basic functions for writing explicitly concurrent programs are exported by the Control.Concurrent which defines an abstract type ThreadId to allow the identification of Haskell threads (which should not be confused with operating system threads). A new thread may be created for any computation in the IO monad which returns an IO unit result by calling the forkIO function:

```
forkIO :: IO () -> IO ThreadId
```

Why does the forkIO function take an expression in the IO monad rather than taking a pure functional expression as its argument? The reason for this is that most concurrent programs need to communicate with each other and this is done through shared synchronized state and these stateful operations have to be carried out in the IO monad.

One important thing to note about threads that are created by calling forkIO is that the main program (the parent thread) will not automatically wait for the child threads to terminate.

Sometimes it is necessary to use a real operating system thread and this can be achieved using the forkOS function:

```
forkOS :: IO () -> IO ThreadId
```

Threads created by this call are bound to a specific operating system thread and this capability is required to support certain kinds of foreign calls made by Haskell programs to external code.

5.2 MVars

To facilitate communication and synchronization between threads Haskell provides MVars which are exported by the module Control.Concurrent.MVar. Operations are provided the create an empty MVar, to create a new MVar with an initial value, to remove a value from an MVar, the observe the value in an MVar (plus non-blocking variants) as well as several other useful operations.

```
data MVar a

newEmptyMVar :: IO (MVar a)
newMVar :: a -> IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
readMVar :: MVar a -> IO a
tryTakeMVar :: MVar a -> IO (Maybe a)
tryPutMVar :: MVar a -> a -> IO Bool
isEmptyMVar :: MVar a -> IO Bool
-- Plus other functions
```

One can use a pair of MVars and the blocking operations putMVar and takeMVar to implement a *rendezvous* between two threads.

```haskell
module Main
where
import Control.Concurrent
import Control.Concurrent.MVar

threadA :: MVar Int -> MVar Float -> IO ()
threadA valueToSendMVar valueReceivedMVar
  = do -- some work
       -- new perform rendezvous by sending 72
       putMVar valueToSendMVar 72 -- send value
       v <- takeMVar valueToReadMVar
       putStrLn (show v)

threadB :: MVar Int -> MVar Float -> IO ()
threadB valueToReceiveMVar valueToSendMVar
  = do -- some work
       -- now perform rendezvous by waiting on value
       z <- takeMVar valueToReceiveMVar
       putMVar valueToSendMVar (1.2 * z)
       -- continue with other work

main :: IO ()
main
  = do aMVar <- newEmptyMVar
       bMVar <- newEmptyMVar
       forkIO (threadA aMVar bMVar)
       forkIO (threadB aMVar bMVar)
       threadDelay 1000 -- wait for threadA and threadB to finish (sleazy)
```

Exercise. Re-write this program to remove the use of threadDelay by using some other more robust mechanism to ensure the main thread does not complete until all the child threads have completed.

```haskell
module Main
where
import Control.Parallel
import Control.Concurrent
import Control.Concurrent.MVar

fib :: Int -> Int
-- As before

fibThread :: Int -> MVar Int -> IO ()
fibThread n resultMVar
  = putMVar resultMVar (fib n)

sumEuler :: Int -> Int
-- As before

s1 :: Int
s1 = sumEuler 7450
```

```
main :: IO ()
main
  = do putStrLn "explicit SumFibEuler"
       fibResult <- newEmptyMVar
       forkIO (fibThread 40 fibResult)
       pseq s1 (return ())
       f <- takeMVar fibResult
       putStrLn ("sum: " ++ show (s1+f))
```

The result of running this program with one and two threads is:

```
satnams@MSRC-1607220 ~/papers/afp2008/explicit
$ time ExplicitWrong +RTS -N1
explicit SumFibEuler
sum: 119201850

real    0m40.473s
user    0m0.000s
sys     0m0.031s

satnams@MSRC-1607220 ~/papers/afp2008/explicit
$ time ExplicitWrong +RTS -N2
explicit SumFibEuler
sum: 119201850

real    0m38.580s
user    0m0.000s
sys     0m0.015s
```

To fix this problem we must ensure the computation of fib fully occurs inside the fibThread thread which we do by using pseq.

```
module Main
where
import Control.Parallel
import Control.Concurrent
import Control.Concurrent.MVar

fib :: Int -> Int
-- As before

fibThread :: Int -> MVar Int -> IO ()
fibThread n resultMVar
  = do pseq f (return ()) -- Force evaluation in this thread
       putMVar resultMVar f
    where
    f = fib n

sumEuler :: Int -> Int
```

```
-- As before

s1 :: Int
s1 = sumEuler 7450

main :: IO ()
main
  = do putStrLn "explicit SumFibEuler"
       fibResult <- newEmptyMVar
       forkIO (fibThread 40 fibResult)
       pseq s1 (return ())
       f <- takeMVar fibResult
       putStrLn ("sum: " ++ show (s1+f))
```

6 Nested data parallelism

This chapter was written in collaboration with Manuel Chakravarty, Gabriele Keller, and Roman Leshchinskiy (University of New South Wales, Sydney).

The two major ways of exploiting parallelism that we have seen so far each have their disadvantages:

- The `par/seq` style is semantically transparent, but it is hard to ensure that the granularity is consistently large enough to be worth spawning new threads.
- Explicitly-forked threads, communicating using `MVar`s or STM give the programmer precise control over granularity, but at the cost of a new layer of semantic complexity: there are now many threads, each mutating shared memory. Reasoning about all the inter leavings of these threads is hard, especially if there are a lot of them.

Furthermore, neither is easy to implement on a distributed-memory machine, because any pointer can point to any value, so spatial locality is poor. It is possible to support this anarchic memory model on a distributed-memory architecture, as Glasgow Parallel Haskell has shown [3], but it is very hard to get reliable, predictable, and scalable performance. In short, we have no good *performance model*, which is a Bad Thing if your main purpose in writing a parallel program is to improve performance.

In this chapter we will explore another parallel programming paradigm: *data parallelism*. The basic idea of data parallelism is simple:

Do the same thing, in parallel, to every element of a large collection of values.

Not every program can be expressed in this way, but data parallelism is very attractive for those that can, because:

– Everything remains purely functional, like `par/seq`, so there is no new semantic complexity.
– Granularity is very good: to a first approximation, we get just one thread (with its attendant overheads) for each physical processor, rather than one thread for each data item (of which there are zillions).
– Locality is very good: the data can be physically partitioned across the processors without random cross-heap pointers.

As a result, we get an excellent performance model.

6.1 Flat data parallelism

Data parallelism sounds good doesn't it? Indeed, data-parallel programming is widely and successfully used in mainstream languages such as High-Performance Fortran. However, there's a catch: the application has to fit the data-parallel programming paradigm, and only a fairly narrow class of applications do so. But this narrow-ness is largely because mainstream data-parallel technology only supports so-called *flat* data parallelism. Flat data parallelism works like this

> Apply the same *sequential* function `f`, in parallel, to every element of a large collection of values `a`. Not only is `f` sequential, but it has a similar run-time for each element of the collection.

Here is how we might write such a loop in Data Parallel Haskell:

```
sumSq :: [: Float :] -> Float
sumSq a = sumP [: x*x | x <- a :]
```

The data type `[: Float :]` is pronounced "parallel vector of `Float`". We use a bracket notation reminiscent of lists, because parallel vectors are similar to lists in that consist of an sequence of elements. Many functions available for lists are also available for parallel vectors. For example

```
mapP     :: (a -> b) -> [:a:] -> [:b:]
zipWithP :: (a -> b -> c) -> [:a:] -> [:b:] -> [:c:]
sumP     :: Num a => [:a:] -> a

(+:+)    :: [:a:] -> [:a:] -> [:a:]
filterP  :: (a -> Bool) -> [:a:] -> [:a:]
anyP     :: (a -> Bool) -> [:a:] -> Bool
concatP  :: [:[:a:]:] -> [:a:]
nullP    :: [:a:] -> Bool
lengthP  :: [:a:] -> Int
(!:)     :: [:a:] -> Int -> a   -- Zero-based indexing
```

These functions, and many more, are exported by `Data.Array.Parallel`. Just as we have list comprehensions, we also have parallel-array comprehensions, of which one is used in the above example. But, just as with list comprehensions, array comprehensions are syntactic sugar, and we could just as well have written

```
sumSq :: [: Float :] -> Float
sumSq a = sumP (mapP (\x -> x*x) a)
```

Notice that there is no `forkIO`, and no `par`. The parallelism comes implicitly from use of the primitives operating on parallel vectors, such as `mapP`, `sumP`, and so on.

Flat data parallelism is not restricted to consuming a single array. For example, here is how we might take the product of two vectors, by multiplying corresponding elements and adding up the results:

```
vecMul :: [:Float:] -> [:Float:] -> Float
vecMul a b = sumP [: x*y | x <- a | y <- b :]
```

The array comprehension uses a second vertical bar "`|`" to indicate that we interate over `b` in lockstep with `a`. (This same facility is available for ordinary list comprehensions too.) As before the comprehension is just syntactic sugar, and we could have equivalently written this:

```
vecMul :: [:Float:] -> [:Float:] -> Float
vecMul a b = sumP (zipWithP (*) a b)
```

6.2 Pros and cons of flat data parallelism

If you can express your program using flat data parallelism, we can implement it really well on a N-processor machine:

- Divide `a` into N chunks, one for each processor.
- Compile a sequential loop that applies `f` successively to each element of a chunk
- Run this loop on each processor
- Combine the results.

Notice that the granularity is good (there is one large-grain thread per processor); locality is good (the elements of `a` are accessed successively); load-balancing is good (each processor does $1/N$ of the work). Furthermore the algorithm works well even if `f` itself does very little work to each element, a situation that is a killer if we spawn a new thread for each invocation of `f`.

In exchange for this great implementation, the programming model is horrible: *all the parallelism must come from a single parallel loop.* This restriction makes the programming model is very non-compositional. If you have an existing function `g` written using the data-parallel `mapP`, you can't call `g` from another data-parallel map (e.g. `mapP g a`), because the argument to `mapP` must be a *sequential* function.

Furthermore, just as the control structure must be flat, so must the data structure. We cannot allow `a` to contain rich nested structure (e.g. the elements of `a` cannot themselves be vectors), or else similar-run-time promise of `f` could not be guaranteed, and data locality would be lost.

6.3 Nested data parallelism

In the early 90's, Guy Blelloch described *nested* data-parallel programming. The idea is similar:

> Apply the same function `f`, in parallel, to every element of a large collection of values `a`. However, `f` may *itself* be a (nested) data-parallel function, and does not need to have a similar run-time for each element of `a`.

For example, here is how we might multiply a matrix by a vector:

```
type Vector = [:Float:]
type Matrix = [:Vector:]

matMul :: Matrix -> Vector -> Vector
matMul m v = [: vecMul r v | r <- m :]
```

That is, for each row of the matrix, multiply it by the vector `v` using `vecMul`. Here we are calling a data-parallel function `vecMul` from inside a data-parallel operation (the comprehension in `matMul`).

In very regular examples like this, consisting of visible, nested loops, modern FORTRAN compilers can collapse a loop nest into one loop, and partition the loop across the processors. It is not entirely trivial to do this, but it is well within the reach of compiler technology. But the flattening process only works for the simplest of cases. A typical complication is the matrices may be *sparse*.

A sparse vector (or matrix) is one in which almost all the elements are zero. We may represent a sparse vector by a (dense) vector of pairs:

```
type SparseVector = [: (Int, Float) :]
```

In this representation, only non-zero elements of the vector are represented, by a pair of their index and value. A sparse matrix can now be represented by a (dense) vector of rows, each of which is a sparse vector:

```
type SparseMatrix = [: SparseVector :]
```

Now we may write `vecMul` and `matMul` for sparse arguments thus[1]:

```
sparseVecMul :: SparseVector -> Vector -> Float
sparseVecMul sv v = sumP [: x * v!:i | (i,x) <- sv :]

sparseMatMul :: SparseMatrix -> Vector -> Vector
sparseMatMul sm v = [: sparseVecMul r v | r <- sm :]
```

[1] Incidentally, although these functions are very short, they are important in some applications. For example, multiplying a sparse matrix by a dense vector (i.e. `sparseMatMul`) is the inner loop of the NAS Conjugate Gradient benchmark, consuming 95% of runtime [4].

We use the indexing operator `(!:)` to index the dense vector `v`. In this code, the control structure is the same as before (a nested loop, with both levels being data-parallel), but now the data structure is much less regular, and it is *much* less obvious how to flatten the program into a single data-parallel loop, in such a way that the work is evenly distributed over N processors, regardless of the distribution of non-zero data in the matrix.

Blelloch's remarkable contribution was to show that it is possible to take *any* program written using nested data parallelism (easy to write but hard to implement efficiently), and transform it systematically into a program that uses flat data parallelism (hard to write but easy to implement efficiently). He did this for a special-purpose functional language, NESL, designed specifically to demonstrate nested data parallelism.

As a practical programming language, however, NESL is very limited: it is a first-order language, it has only a fixed handful of data types, it is implemented using an interpreter, and so on. Fortunately, in a series of papers, Manuel Chakravarty, Gabriele Keller and Roman Leshchinskiy have generalized Blelloch's transformation to a modern, higher order functional programming language with user-defined algebraic data types – in other words, Haskell. Data Parallel Haskell is a research prototype implementation of all these ideas, in the Glasgow Haskell Compiler, GHC.

The matrix-multiply examples may have suggested to you that Data Parallel Haskell is intended primarily for scientific applications, and that the nesting depth of parallel computations is statically fixed. However the programming paradigm is much more flexible than that. In the rest of this chapter we will give a series of examples of programming in Data Parallel Haskell, designed to help you gain familiarity with the programming style.

Most (in due course, all) of these examples can be found at in the Darcs repository `http://darcs.haskell.org/packages/ndp`, in the sub-directory `examples/`. You can also find a dozen or so other examples of data-parallel algorithms written in NESL at `http://www.cs.cmu.edu/~scandal/nesl/algorithms.html`.

6.4 Word search

Here is a tiny version of a web search engine. A `Document` is a vector of words, each of which is a string. The task is to find all the occurrences of a word in a large collection of documents, returning the matched documents and the matching word positions in those documents. So here is the type signature for `search`:

```
type Document = [: String :]
type DocColl  = [: Document :]
search :: DocColl -> String -> [: (Document, [:Int:]) :]
```

We start by solving an easier problem, that of finding all the occurrences of a word in a single document:

```
wordOccs :: Document -> String -> [:Int:]
wordOccs d s = [: i | (i,s2) <- zipP [:1..lengthP d:] d
                    , s == s2 :]
```

Here we use a *filter* in the array comprehension, that selects just those pairs (i,s2) for which s==s2. Because this is an array comprehension, the implied filtering is performed in data parallel. The (i,s2) pairs are chosen from a vector of pairs, itself constructed by zipping the document with the vector of its indices. The latter vector [: 1..lengthP d :] is again analogous to the list notation [1..n], which generate the list of values between 1 and n. As you can see, in both of these cases (filtering and enumeration) Data Parallel Haskell tries hard to make parallel arrays and vectors as notationally similar as possible.

With this function in hand, it is easy to build search:

```
search :: [: Document :] -> String -> [: (Document, [:Int:]) :]
search ds s = [: (d,is) | d <- ds
                        , let is = wordOccs d s
                        , not (nullP is) :]
```

6.5 Prime numbers

Let us consider the problem of computing the prime numbers up to a fixed number n, using the sieve of Erathosthenes. You may know the cunning solution using lazy evaluation, thus:

```
primes :: [Int]
primes = 2 : [x | x <- [3..]
                , not (any (‘divides‘ x) (smallers x))]
       where
         smallers x = takeWhile (\p -> p*p <= x) primes

divides :: Int -> Int -> Bool
divides a b = b ‘mod‘ a == 0
```

(In fact, this code is *not* the sieve of Eratosthenes, as Melissa O’Neill’s elegant article shows [5], but it will serve our purpose here.) Notice that when considering a candidate prime x, we check that is is not divisible by any prime smaller than the square root of x. This test involves using primes, the very list the definition produces.

How can we do this in parallel? In principle we want to test a whole batch of numbers in parallel for prime factors, so we must specify how big the batch is:

```
primesUpTo :: Int -> [: Int :]
primesUpTo 1 = [: :]
primesUpTo 2 = [: 2 :]
primesUpTo n = smallers +:+
               [: x | x <- [: ns+1..n :]
                    , not (anyP (‘divides‘ x) smallers) :]
```

```
    where
      ns       = intSqrt n
      smallers = primesUpTo ns
```

As in the case of `wordOccs`, we use a boolean condition in a comprehension to filter the candidate primes. This time, however, computing the condition itself is a nested data-parallel computation (as it was in `search`). used here to filter candidate primes `x`.

To compute `smallers` we make a recursive call to `primesUpTo`. This makes `primesUpTo` unlike all the previous examples: the depth of data-parallel nesting is determined *dynamically*, rather than being statically fixed to depth two. It should be clear that the structure of the parallelism is now much more complicated than before, and well out of the reach of mainstream flat data-parallel systems. But it has abundant data parallelism, and will execute with scalable performance on a parallel processor.

6.6 Quicksort

In all the examples so far the "branching factor" has been large. That is, each data-parallel operations has worked on a large collection. What happens if the collection is much smaller? For example, a divide-and-conquer algorithm usually divides a problem into a handful (perhaps only two) sub-problems, solves them, and combines the results. If we visualize the tree of tasks for a divide-and-conquer algorithm, it will have a small branching factor at each node, and may be highly un-balanced.

Is this amenable to nested data parallelism? Yes, it is. Quicksort is a classic divide-and-conquer algorithm, and one that we have already studied. Here it is, expressed in Data Parallel Haskell:

```
qsort :: [: Double :] -> [: Double :]
qsort xs | lengthP xs <=  1 = xs
         | otherwise        = rs!:0 +:+ eq +:+ rs!:1
         where
           p = xs !: (lengthP xs `div` 2)
           lt = [:x | x <- xs, x < p :]
           eq = [:x | x <- xs, x == p:]
           gr = [:x | x <- xs, x > p :]
           rs = mapP qsort [: lt, gr :]
```

The crucial step here is the use of `mapP` on a *two-element* array `[: lt, gr :]`. This says "in data-parallel, apply `qsort` to `lt` and `gr`". The fact that there are only two elements in the vector does not matter. If you visualize the binary tree of sorting tasks that quicksort generates, then each horizontal layer of the tree is done in data-parallel, even though each layer consists of many unrelated sorting tasks.

6.7 Barnes Hut

All our previous examples worked on simple flat or nested collections. Let's now have a look at an algorithm based on a more complex structure, in which the elements of a parallel array come from a *recursive* and *user-defined* algebraic data type.

In the following, we present an implementation[2] of a simple version of the Barnes-Hut n-body algorithm[7], which is a representative of an important class of parallel algorithms covering applications like simulation and radiocity computations. These algorithms consist of two main steps: first, the data is clustered in a hierarchical tree structure; then, the data is traversed according to the hierarchical structure computed in the first step. In general, we have the situation that the computations that have to be applied to data on the same level of the tree can be executed in parallel. Let us first have a look at the Barnes-Hut algorithm and the data structures that are required, before we discuss the actual implementation in parallel Haskell.

An n-body algorithm determines the interaction between a set of particles by computing the forces which act between each pair of particles. A precise solution therefore requires the computations of n^2 forces, which is not feasible for large numbers of particles. The Barnes-Hut algorithm minimizes the number of force calculations by grouping particles hierarchically into *cells* according to their spatial position. The hierarchy is represented by a tree. This allows approximating the accelerations induced by a group of particles on distant particles by using the centroid of that group's cell. The algorithm has two phases: (1) The tree is constructed from a particle set, and (2) the acceleration for each particle is computed in a down-sweep over the tree. Each particle is represented by a value of type `MassPoint`, a pair of position in the two dimensional space and mass:

```
type Vec       = (Double, Double)
type Area      = (Vec, Vec)
type Mass      = Double
type MassPoint = (Vec, Mass)
```

We represent the tree as a node which contains the centroid and a parallel array of subtrees:

```
data Tree = Node MassPoint [:Tree:]
```

Notice that a `Tree` contains a parallel array of `Tree`.

Each iteration of `bhTree` takes the current particle set and the area in which the particles are located as parameters. It first splits the area into four subareas `subAs` of equal size. It then subdivides the particles into four subsets according to the subarea they are located in. Then, `bhTree` is called recursively for each subset and subarea. The resulting four trees are the subtrees of the tree representing the particles of the area, and the centroid of their roots is the centroid of the complete area. Once an area contains only one particle, the recursion terminates.

[2] Our description here is based heavily on that in [6].

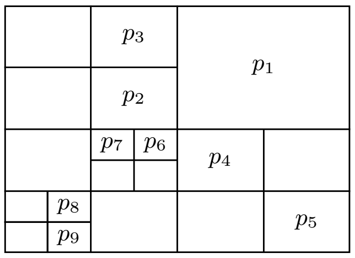


Fig. 7. Hierarchical division of an area into subareas

Figure 7 shows such a decomposition of an area for a given set of particles, and Figure 8 displays the resulting tree structure.

```
bhTree :: [:MassPnt:] -> Area -> Tree
bhTree p  area = Node p [::]
bhTree ps area =
  let
     subAs = splitArea area
     pgs   = splitParticles ps subAs
     subts = [: bhTree pg a| pg <- pgs | a <- subAs :]
     cd    = centroid [:mp | Node mp _ <- subts :]
   in Node cd subts
```

The tree computed by `bhTree` is then used to compute the forces that act on each particle by a function `accels`. It first splits the set of particles into two subsets: `fMps`, which contains the particles far away (according to a given criteria), and `cMps`, which contains those close to the centroid stored in the root of the tree. For all particles in `fMps`, the acceleration is approximated by computing the interaction between the particle and the centroid. Then, `accels` is called recursively for with `cMps` and each of the subtrees. The computation terminates once there are no particles left in the set.

```
accels:: Tree -> [:MassPoint:] -> [:Vec:]
accels _                [::] = [::]
accels (Node cd subts)  mps  =
  let
     (fMps, cMps) = splitMps mps
     fAcs         = [:accel  cd mp | mp <- fMps:]
     cAcs         = [:accels t cMps| t <- subts:]
  in combine farAcs closeAcs

accel :: MassPoint -> MassPoint -> Vec
-- Given two particles, the function accel computes the
-- acceleration that one particle exerts on the other
```

The tree is both built and traversed level by level, i.e., all nodes in one level of the tree are processed in a single parallel step, one level after the other. This

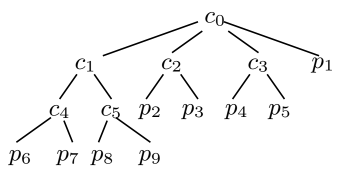


Fig. 8. Example of a Barnes-Hut tree.

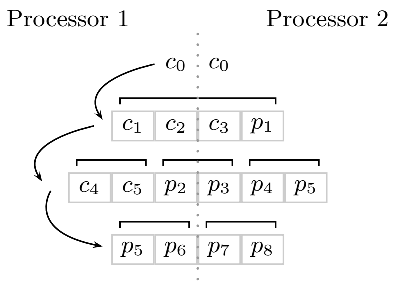


Fig. 9. Distribution of the values of the flattened tree

information is important for the compiler to achieve good data locality and load balance, because it implies that each processor should have approximately the same number of masspoints of each level. We can see the tree as having a sequential dimension to it, its depth, and a parallel dimension, the breadth, neither of which can be predicted statically. The programmer conveys this information to the compiler by the choice the data structure: By putting all subtrees into a parallel array in the type definition, the compiler assumes that all subtrees are going to be processed in parallel. The depth of the tree is modeled by the recursion in the type, which is inherently sequential.

6.8 A performance model

One of the main advantages of the data parallel programming model is that it comes with a *performance model* that lets us make reasonable predictions about the behavior of the program on a parallel machine, including its *scalability* – that is, how performance changes as we add processors. So what is this performance model?

First, we must make explicit something we have glossed over thus far: data-parallel arrays are strict. More precisely, if any element of a parallel diverges, then all elements diverge[3]. This makes sense, because if we demand any element of a parallel array then we must compute them all in data parallel; and if that

[3] What if the elements are pairs? See Leshchinskiy's thesis for the details [8].

computation diverges we are justified in not returning any of them. The same constraint means that we can represent parallel arrays very efficiently. For example, an array of floats, `[:Float:]`, is represented by a contiguous array of unboxed floating-point numbers. There are no pointers, and iterating over the array has excellent spatial locality.

In reasoning about performance, Blelloch [9] characterizes the *work* and *depth* of the program:

- The *work*, W, of the program is the time it would take to execute on a single processor.
- The *depth*, D, of the program is the time it would take to execute on an infinite number processors, under the assumption that the additional processors leap into action when (but only when) a `mapP`, or other data-parallel primitive, is executed.

If you think of the unrolled data-flow diagram for the program, the work is the number of nodes in the data-flow diagram, while the depth is the longest path from input to output.

Of course, we do not have an infinite number of processors. Suppose instead that we have P processors. Then if everything worked perfectly, the work work be precisely evenly balanced across the processors and the execution time T would be W/P. That will not happen if the depth D is very large. So in fact, we have

$$W/P \leq T \leq W/P + L * D$$

where L is a constant that grows with the latency of communication in the machine. Even this is a wild approximation, because it takes no account of bandwidth limitations. For example, between each of the recursive calls in the Quicksort example there must be some data movement to bring together the elements less than, equal to, and greater than the pivot. Nevertheless, if the network bandwidth of the parallel machine is high (and on serious multiprocessors it usually is) the model gives a reasonable approximation.

How can we compute work and depth? It is much easier to reason about the work of a program in a strict setting than in a lazy one, because all subexpressions are evaluated. This is why the performance model of the data-parallel part of DPH is more tractable than for Haskell itself.

The computation of depth is where we take account of data parallelism. Figure 10 shows the equations for calculating the depth of a closed expression e, where $\mathcal{D}[\![e]\!]$ means "the depth of e". These equations embody the following ideas:

- By default execution is sequential. Hence, the depth of an addition is the sum of the depths of its arguments.
- The parallel primitive `mapP`, and its relatives such as `filterP`, can take advantage of parallelism, so the depth is the worst depth encountered for any element.
- The parallel reduction primitive `sumP`, and its relatives, take time logarithmic in the length of the array.

$$
\begin{aligned}
\mathcal{D}[\![k]\!] &= 0 && \text{where } k \text{ is a constant}\\
\mathcal{D}[\![x]\!] &= 0 && \text{where } x \text{ is a variable}\\
\mathcal{D}[\![e_1 + e_2]\!] &= 1 + \mathcal{D}[\![e_1]\!] + \mathcal{D}[\![e_2]\!] &&\\
&&&\\
\mathcal{D}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] &= \mathcal{D}[\![e_1]\!] + \mathcal{D}[\![e_2]\!] && \text{if } e_1 = \texttt{True}\\
&= \mathcal{D}[\![e_1]\!] + \mathcal{D}[\![e_3]\!] && \text{if } e_1 = \texttt{False}\\
\mathcal{D}[\![\texttt{let } x\texttt{=}e \texttt{ in } b]\!] &= \mathcal{D}[\![b[e/x]]\!] &&\\
&&&\\
\mathcal{D}[\![e_1 \texttt{ +:+ } e_2]\!] &= 1 + \mathcal{D}[\![e_1]\!] + \mathcal{D}[\![e_2]\!] &&\\
\mathcal{D}[\![\texttt{concatP } e]\!] &= 1 + \mathcal{D}[\![e]\!] &&\\
\mathcal{D}[\![\texttt{mapP } f\ e]\!] &= 1 + \mathcal{D}[\![e]\!] + \underset{x \in e}{max}\, \mathcal{D}[\![f\ x]\!] &&\\
\mathcal{D}[\![\texttt{filterP } f\ e]\!] &= 1 + \mathcal{D}[\![e]\!] + \mathcal{D}[\![f]\!] &&\\
&&&\\
\mathcal{D}[\![\texttt{sumP } e]\!] &= 1 + \mathcal{D}[\![e]\!] + log(length(e)) &&
\end{aligned}
$$

Fig. 10. Depth model for closed expressions

The rule for `mapP` dirctly embodies the idea that nested data parallelism is flattened. For example, suppose e :: `[:[:Float:]:]`. Then, applying the rules we see that

$$
\begin{aligned}
\mathcal{D}[\![\texttt{mapP } f \ (\texttt{concatP } e]\!] &= 1 + \mathcal{D}[\![\texttt{concatP } e]\!] + \underset{x \in \texttt{concatP } e}{max}\, \mathcal{D}[\![f\ x]\!]\\
&= 1 + 1 + \mathcal{D}[\![e]\!] + \underset{x \in \texttt{concatP } e}{max}\, \mathcal{D}[\![f\ x]\!]\\
&= 2 + \mathcal{D}[\![e]\!] + \underset{xs \in e}{max}\ \underset{x \in xs}{max}\, \mathcal{D}[\![f\ x]\!]\\
\mathcal{D}[\![\texttt{mapP } (\texttt{mapP } f)\ e]\!] &= 1 + \mathcal{D}[\![e]\!] + \underset{xs \in e}{max}\, \mathcal{D}[\![\texttt{mapP } f\ xs]\!]\\
&= 1 + \mathcal{D}[\![e]\!] + 1 + \underset{xs \in e}{max}\ \underset{x \in xs}{max}\, \mathcal{D}[\![f\ x]\!]\\
&= 2 + \mathcal{D}[\![e]\!] + \underset{xs \in e}{max}\ \underset{x \in xs}{max}\, \mathcal{D}[\![f\ x]\!]
\end{aligned}
$$

Notice that although the second case is a *nested* data-parallel computation, it has the same depth expression as the first: the data-parallel nesting is flattened.

These calculations are obviously very approximate, certainly so far as constant factors are concerned. For example, in the inequality for execution time,

$$W/P \leq T \leq W/P + L * D$$

we do not know the value of the latency-related constant L. However, what we are primarily looking for is the *Asymptotic Scalability* (AS) property:

> A program has the Asymptotic Scalability property if D grows asymptotically more slowly than W, as the size of the problem increases.

If this is so then, for a sufficiently large problem and assuming sufficient network bandwidth, performance should scale linearly with the number of processors.

For example, the functions `sumSq` and `search` both have constant depth, so both have the AS property, and (assuming sufficient bandwidth) performance should scale linearly with the number of processors after some fairly low threshold.

For Quicksort, an inductive argument shows that the depth is logarithmic in the size of the array, assuming the pivot is not badly chosen. So $W = O(nlogn)$ and $D = O(logn)$, and Quicksort has the AS property.

For computing primes, the depth is smaller: $D = O(loglogn)$. Why? Because at every step we take the square root of n, so that at depth d we have $n = 2^{2^d}$. Almost all the work is done at the top level. The work at each level involves comparing all the numbers between $\sqrt{n}$ and n with each prime smaller than $\sqrt{n}$. There are approximately $\sqrt{n}/logn$ primes smaller than $\sqrt{n}$, so the total work is roughly $W = O(n^{3/2}/logn)$. So again we have the AS property.

Leshchinskiy *et al* [10] give further details of the cost model.

6.9 How it works

NESL's key insight is that it is possible to transform a program that uses *nested* data-parallelism into one that uses only *flat* data parallelism. While this little miracle happens behind the scenes, it is instructive to have some idea how it works, just as a car driver may find some knowledge of internal combustion engines even if he is not a skilled mechanic. The description here is necessarily brief, but the reader may find a slightly more detailed overview in [11], and in the papers cited there.

We call the nested-to-flat transformation the *vectorization* transform. It has two parts:

– Transform the *data* so that all parallel arrays contain only primitive, flat data, such as `Int`, `Float`, `Double`.
– Transform the *code* to manipulate this flat data.

To begin with, let us focus on the first of these topics. We may consider it as the driving force, because nesting of data-parallel operations is often driven by nested data structures.

Transforming the data As we have already discussed, a parallel array of `Float` is represented by a contiguous array of honest-to-goodness IEEE floating point numbers; and similarly for `Int` and `Double`. It is as if we could define the parallel-array type by cases, thus:

```
data instance [: Int    :] = PI Int ByteArray
data instance [: Float  :] = PF Int ByteArray
data instance [: Double :] = PD Int ByteArray
```

In each case the `Int` field is the size of the array. These `data` declarations are unusual because they are *non-parametric*: the representation of an array depends on the type of the elements[4].

[4] None of this is visible to the programmer, but the `data instance` notation is in fact available to the programmer in recent versions of GHC [12, 13]. Why? Because

Matters become even more interesting when we how to represent a parallel array of pairs. We must not represent it as a vector of pointers to heap-allocated pairs, scattered randomly around the address space. We get much better locality if we instead represented it as a *pair of arrays* thus:

```
data instance [: (a,b) :] = PP [:a:] [:b:]
```

What about a parallel array of parallel arrays? Again, we must avoid a vector of pointers. Instead, the natural representation is obtained by literally concatenating the (representation of) the sub-vectors into one giant vector, together with a vector of indices to indicate where each of the sub-vectors begins.

```
data instance [: [:a:] :] = PA [:Int:] [:a:]
```

By way of example, recall the data types for sparse matrices:

```
type SparseMatrix = [: SparseVector :]
type SparseVector = [: (Int, Float) :]
```

Now consider this tiny matrix, consisting of two short documents:

```
m :: SparseMatrix
m = [: [:(1,2.0), (7,1.9):], [:(3,3.0):] :]
```

This would be represented as follows:

```
PA [:0,2:] (PP [:1,   7,   3  :]
               [:1.0, 1.9, 3.0:])
```

The array as the leaves are themselves represented as byte arrays:

```
PA (PI 2 #<0x0,0x2>)
   (PP (PI 3 #<0x0,    0x7,     0x3>)
       (PF 3 #<0x9383, 0x92818, 0x91813>))
```

Here we have invented a fanciful notation for literal `ByteArray`s (not supported by GHC, let alone Haskell) to stress the fact that in the end everything boils down to literal bytes. (The hexadecimal encodings of floating point numbers are also made up because the real ones have many digits!)

We have not discussed how to represent arrays of sum types (such as `Bool`, `Maybe`, or lists), nor of function types — see [14] and [8] respectively.

GHC has a typed intermediate language so we needed to figure out how to give a *typed* account of the vectorization transformation, and once that is done it seems natural to offer it to the programmer. Furthermore, much of the low-level support code for nested data parallelism is itself written in Haskell, and operates directly on the post-vectorization array representation.

Vectorising the code As you can see, data structures are transformed quite radically by the vectorisation transform, and it follows that the code must be equally radically transformed. Space precludes proper treatment here; a good starting point is Keller's thesis [15].

A data-parallel program has many array-valued sub-expressions. For example, in `sumSq` we see

```
sumSq a = sumP [: x*x | x <- a :]
```

However, if `a` is a big array, it would be silly to compute a new, equally big array of squares, only to immediately consume it with `sumP`. It would be much better for each processor to zip down its chunk of `a`, adding the square of each element into a running total, and for each processor's total to be combined.

The elimination of intermediate arrays is called *fusion* and is crucial to improve the constant factor of Data Parallel Haskell. It turns out that vectorisation introduces many *more* intermediate arrays, which makes fusion even more important. These constant factors are extremely important in practice: if there is a slow-down of a factor of 50 relative to C, then even if you get linear speedup by adding processors, Data Parallel Haskell is unlikely to become popular.

6.10 Running Data Parallel Haskell

GHC 6.6 and 6.8 come with support for Data Parallel Haskell syntax, and a *purely sequential* implementation of the operations. So you can readily try out all of the examples in this paper, and ones of your own devising thus:

- Use `ghc` or `ghci` version 6.6.x or 6.8.x.
- Use flags `-fparr` and `-XParallelListComp`.
- Import module `GHC.PArr`.

Some support for genuinely-parallel Data Parallel Haskell, including the all-important vectorisation transformation, will be in GHC 6.10 (planned release: autumn 2008). It is not yet clear just how complete the support will be at that time. At the time of writing, for example, type classes are not vectorized, and neither are lists. Furthermore, in a full implementation we will want need support for partial vectorisation [16], among other things.

As a result, although all the examples in this paper should work when run sequentially, they may not all vectorise as written, even in GHC 6.10.

A good source of working code is in the Darcs repository `http://darcs.haskell.org/packages/ndp`, whose sub-directory `examples/` contains many executable examples.

6.11 Further reading

Blelloch and Sabot originated the idea of compiling nested data parallelism into flat data parallelism [17], but an easier starting point is probably Blelloch subsequence CACM paper "Programming parallel algorithms" [9], and the NESL language manual [18].

Keller's thesis [15] formalized an intermediate language that models the central aspects of data parallelism, and formalized the key vectorisation transformation. She also studied array *fusion*, to eliminate unnecessary intermediate arrays. Leshchinskiy's thesis [8] extended this work to cover higher order languages ([19] gives a paper-sized summary), while Chakravarty and Keller explain a further generalization to handle user-defined algebraic data types [14].

Data Parallel Haskell is an ongoing research project [11]. The Manticore project at Chicago shares similar goals [20].

References

1. Mohr, E., Kranz, D.A., Halstead, R.H.: Lazy task creation – a technique for increasing the granularity of parallel programs. IEEE Transactions on Parallel and Distributed Systems **2**(3) (July 1991)
2. Trinder, P., Loidl, H.W., Pointon, R.F.: Parallel and Distributed Haskells. Journal of Functional Programming **12**(5) (July 2002) 469–510
3. Trinder, P., Loidl, H.W., Barry, E., Hammond, K., Klusik, U., Peyton Jones, S., Rebón Portillo, Á.J.: The Multi-Architecture Performance of the Parallel Functional Language GPH. In Bode, A., Ludwig, T., Wismüller, R., eds.: Euro-Par 2000 — Parallel Processing. Lecture Notes in Computer Science, Munich, Germany, 29.8.-1.9., Springer-Verlag (2000)
4. Prins, J., Chatterjee, S., Simons, M.: Irregular computations in fortran: Expression and implementation strategies. Scientific Programming **7** (1999) 313–326
5. O'Neill, M.: The genuine sieve of Eratosthenes. Submitted to JFP (2007)
6. Chakravarty, M., Keller, G., Lechtchinsky, R., Pfannenstiel, W.: Nepal – nested data-parallelism in haskell. In Sakellariou, Keane, Gurd, Freeman, eds.: Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference. Number 2150 in LNCS, Springer-Verlag (2001) 524–534
7. Barnes, J., Hut, P.: A hierarchical $O(n \log n)$ force calculation algorithm. Nature **324** (December 1986)
8. Leshchinskiy, R.: Higher-order nested data parallelism: semantics and implementation. PhD thesis, Technical University of Berlin (2006)
9. Blelloch, G.: Programming parallel algorithms. Communications of the ACM **39**(3) (March 1996) 85–97
10. Leshchinskiy, R., Chakravarty, M., Keller, G.: Costing nested array codes. Parallel Processing Letters **12** (2002) 249–266
11. Chakravarty, M., Leshchinskiy, R., Jones, S.P., Keller, G.: Data Parallel Haskell: a status report. In: ACM Sigplan Workshop on Declarative Aspects of Multicore Programming, Nice (January 2007)
12. Schrijvers, T., Jones, S.P., Chakravarty, M., Sulzmann, M.: Type checking with open type functions. Submitted to ICFP'08 (2008)
13. Chakravarty, M., Keller, G., Peyton Jones, S.: Associated type synonyms. In: ACM SIGPLAN International Conference on Functional Programming (ICFP'05), Tallinn, Estonia (2005)
14. Chakravarty, M.M., Keller, G.: More types for nested data parallel programming. In: ACM SIGPLAN International Conference on Functional Programming (ICFP'00), Montreal, ACM Press (September 2000) 94–105

15. Keller, G.: Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines. PhD thesis, Technische Universite at Berlin, Fachbereich Informatik (1999)
16. Chakravarty, M.M., Leshchinskiy, R., Jones, S.P., Keller, G.: Partial vectorisation of Haskell programs. In: Proc ACM Workshop on Declarative Aspects of Multicore Programming, San Francisco, ACM Press (January 2008)
17. Blelloch, G., Sabot, G.: Compiling collection-oriented languages onto massively parallel computers. Journal of Parallel and Distributed Computing **8** (February 1990) 119 – 134
18. Blelloch, G.: NESL: A nested data-parallel language (3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University (September 1995)
19. Leshchinskiy, R., Chakravarty, M.M., Keller, G.: Higher order flattening. In: Third International Workshop on Practical Aspects of High-level Parallel Programming (PAPP 2006). LNCS, Springer (2006)
20. Fluet, M., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Manticore: A heterogeneous parallel language. In: ACM Sigplan Workshop on Declarative Aspects of Multicore Programming, Nice (January 2007)