The λ Abroad

A Functional Approach To Software Components

Een functionele benadering van software componenten
(met een samenvatting in het Nederlands)

Proefschrift ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de Rector Magnificus, Prof. dr W.H. Gispen, ingevolge het besluit van het College voor Promoties in het openbaar te verdedigen op dinsdag 4 november 2003 des middags te 12.45 uur
door

Daniel Johannes Pieter Leijen

geboren op 7 Juli 1973, te Alkmaar

promotor:	Prof. dr S.D. Swierstra, Universiteit Utrecht.
co-promotor:	dr H.J.M. Meijer, Microsoft Research.

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics), and has been financed by Ordina.

Printed by Febodruk 2003. Cover illustration shows the heavily cratered surface of the planet Mercury, photographed by the mariner 10.

ISBN 90-9017528-8

Contents

Dankwoord

Ik herinner mij nog goed dat ik in het 3e jaar van mijn studie voor het eerst het boek van Richard Bird en Philip Wadler, “Introduction to Functional Programming”, onder ogen kreeg. Functioneel programmeren, gebaseerd op de zogeheten λ-calculus, bezat een zodanige elegantie en klaarheid, dat ik onmiddellijk besloot mijn afstudeer scriptie daar aan te wijden. Niet iedereen was daarover even enthousiast – functionele talen zouden inefficient zijn, en niet geschikt om “echte” programma’s mee te maken. Wie had toen kunnen denken dat ik nu, 7 jaar later, mijn proefschrift volledig aan dat onderwerp zou wijden!

Allereerst zou ik daarom mijn leraren van het eerste uur willen bedanken. John Mountjoy die mij heeft ingewijd in de schoonheid van de λ-calculus en puur functioneel programmeren, en Marcel Beemster die mij hielp om af te kunnen studeren bij Simon Peyton Jones, een van de beste wetenschappers op het gebied van funtionele talen. Hoewel ik toen dacht naar Glasgow te gaan, bleek Simon op sabbatical te zijn en ging ik uiteindelijk naar het Oregon Graduate Institute of Science and Technology in de VS.

Op het OGI, ontmoette ik ook Erik Meijer, en terug in Nederland kon ik bij hem mijn promotie onderzoek doen aan de Universiteit Utrecht. Ik had mij de eerste jaren geen betere begeleider kunnen wensen – ik ken weinig mensen die zo inspirerend en motiverend kunnen zijn. Erik, bedankt voor al je inzichten die je met me deelde en je niet aflatende passie voor onderzoek, die mij ook vandaag de dag nog inspireert.

Aan alle goede dingen komen een eind, en helaas verliet Erik na twee jaar de Universiteit Utrecht om bij Microsoft nieuwe uitdagingen aan te gaan. Na zijn vertrek heb ik toch enige tijd moeite gehad om mijn eigen richting in het onderzoek te vinden en mijn onderzoeks resultaten te bundelen. Gelukkig heb ik toen veel steun gekregen van mijn promotor Doaitse Swierstra, die mij voortdurend motiveerde om ook de minder aangename kanten van het promoveren onder ogen te zien. Zonder zijn kritische discussies en het voortdurend lettten op kwaliteit had dit proefschrift er zeker anders uitgezien.

Ook wil ik alle collegae van “software technologie” bedanken voor de goede sfeer

en samenwerking. In het bijzonder wil ik mijn lunchgenoten Martijn Schrage en Arjan van IJzendoorn bedanken voor hun vriendschap, wereldlijke discussies en voortdurende afleiding van mijn werk. Hoewel ze eigenlijk bij een andere club hoort, heeft mijn voormalige kamergenote Silja Renooij mij enorm geholpen bij alle praktische zaken die bij het proefschrift behoren.

I would also like to thank Sigbjorn Finne and Mark Shields for being good friends and scientific colleagues – with fond memories of skiïng Mount Hood with Sigbjorn, and climbing the same mountain with Mark a few years earlier. I would also like to take this opportunity to recognise Sigbjorn for all of work that he silently has been doing for the Haskell community over the years.

Furthermore, I would like to thank Microsoft for arranging multiple internships and their good coöperation in general. In particular, I would like to thank James Plamondon, Frank Gocinski, and Andrew Jewsbury for their support, and of course Bill Gates for inviting me to a pizza party at his home!

Natuurlijk wil ik mijn familie en vrienden bedanken voor hun interesse en morele steun – hoewel ik maar zelden duidelijk heb kunnen maken wat ik eigenlijk aan het doen ben. Helaas is tijdens mijn promotie onderzoek mijn oma overleden. “Het leven is een lach en een traan” schreef zij in haar laatste brief. Deze zin heeft veel indruk op mij gemaakt en ook bij deze promotie moet ik er aan denken – ik had heel graag gezien dat ook Henk erbij had kunnen zijn.

De afsluiting van dit dankwoord is voorbehouden aan mijn vriendin Toos Baars, die mij al die jaren met raad, daad, en vooral liefde terzijde stond. Geweldig!

Daan Leijen.

Chapter 1

Overview

In his seminal paper "Why functional programming matters", John Hughes argues that the strength of lazy, higher order languages lies in the ability to glue different functions of a program together. In this thesis we explore how these languages are suited to glue not only functions but also software components written in different languages.

The first chapter of this thesis describes the design of a foreign function interface for the non-strict, higher order language Haskell. A foreign function interfaces (FFI) enables a program to call programs written in other languages and vice versa. Different languages can use different calling conventions and data representations – the foreign language interface ensures a proper calling convention and transforms data values into the representation of the foreign language. The transformation of data values to their foreign representation, called marshalling, is where most complications arise.

Since they cater for a variety of languages, foreign function interfaces tend to become rich, complex, incomplete, and only described by example. In contrast, we offer a formal description of our foreign function interface based on a standed interface definition language (IDL). Furthermore it is carefully factored in two layers: a minimal primitive mechanism that has to be supported by the compiler, and a separate tool, H/Direct, that leverages on this primitive facility to support comprehensive data marshalling.

Building on the basic foreign function interface, the second chapter describes the binding of Haskell to Microsoft's Component Object Model (COM) framework. Component frameworks go beyond a basic foreign function interface by prescribing a system wide protocol for interaction between components. It is an approach to software construction where a program is assembled from software components, perhaps written in different languages, glued together by a common protocol. The

language neutral nature of these architectures offers tremendous opportunites for exotic languages like Haskell. A Haskell program can be sealed as COM component and can therefore interoperate with other client programs that will neither know, nor care, that the component is written in Haskell. A would-be user of Haskell is no longer facing an all-or-nothing choice.

COM is a rich and complex framework and it normally requires quite a bit of C++ code to build a COM component, usually supported by "wizards". We are instead able to provide a library of higher-order functions that make it easy to create components without wizard support. Furthermore, we show that we can model inheritance interface subtyping using just parametric polymorphism and *phantom* types – this has proved essential to conveniently support the object oriented nature of most component frameworks. Even at the most primitive level, the rich type system of Haskell can be used to ensure many properties statically, for example, interface pointers are associated with their corresponding globally unique identifiers and virtual method tables are paired with the appropiate instance data.

After reading the chapters about the interface between Haskell and the imperative world, the reader may wonder whether the rewards are worth the trouble. One potential problem is that a typical COM component is designed with an imperative model in mind, and imposes an imperative style of programming within the functional host language. In the next two chapters, we try to show that it is actually possible to build expressive functional combinator libraries on top of the basic imperative interfaces. The resulting libraries can be seen as an embedded *domain specific language* (DSL) tailored to a certain collection of components. The claim of these chapters is that a higher-order, typed, garbage-collected language such as Haskell can open up new avenues for scripting components. A general strategy is described for embedding domain specific languages in the context of database servers.

The final chapter describes the design of the *lazy virtual machine* (LVM). Just like the Java virtual machine, it defines a portable instruction set and file format, but it is specifically designed to execute languages with non-strict semantics. Part of the design is a compiler toolkit that translates enriched lambda calculus to LVM instructions. The goal was to build a system that lends itself well to experimentation by being modular and extensible. In particular, the work described in the previous chapters gave rise to various extensions to the Haskell language – the LVM proves a great platform to test these ideas.

We focus specifically on the overall design of the instruction set, the operational semantics, and the translation scheme. Instead of giving complex optimized translation schemes, we use a naive and straightforward translation and define a small set of rewrite rules on instructions that achieve the same effect. The correctness of the rewrite rules is relatively easy to prove with the operational semantics. In contrast, an optimized translation scheme is much harder to prove correct, as one has to show a correspondence between the operational semantics of the host language and the generated instructions. Furthermore, the abstract machine is closely related to

the capabilities of contemporary hardware, its state constisting of a code pointer, a stack, and a heap. Therefore, we can reason about implementation techniques that are normally only described informally; examples include exception handling, returning constructors in registers, and black holing.

Chapter 2

H/Direct: a binary language interface for Haskell

This chapter is closely based on the following article, written together with Sigbjorn Finne, Erik Meijer, and Simon Peyton-Jones. Sigbjorn Finne has implemented the full version of the H/Direct compiler.

> *H/Direct: A Binary Foreign Language Interface to Haskell*. In the International Conference on Functional Programming (ICFP), Baltimore,USA, 1998, (Finne *et al.*, 1998). Also appeared in ACM SIGPLAN Notices 34, 1, (Jan. 1999).

2.1 Introduction

In this chapter, we describe the development of a new foreign language interface for Haskell. The interface provides direct access to libraries written in C (or any other language using C's calling convention), and makes it possible to write Haskell procedures that can be called from C. Designing a foreign language interface for a lazy, functional, and garbage collected language like Haskell is rather subtle, and we divide the interface in two separate layers.

The lowest layer defines a set of four primitives that can perform a foreign call, export a Haskell function, manage pointers to foreign data, and export pointers to Haskell data. This layer is called the *Foreign Function Interface* (FFI) and has

evolved into an addendum to the Haskell98 standard (Chakravarty (ed.), 2002)[1]. It is currently supported by most Haskell compilers and interpreters.

However, most complications arise when transforming values built in one language to the other (marshaling). The other part of our interface is therefore a tool, called *H/Direct* that automatically generates Haskell marshaling code from an *interface description* in IDL. The primitive FFI can only deal with the basic binary data that require no marshaling – *H/Direct* provides the means to leverage that primitive facility into the full glory of IDL.

Because they cater for a variety of languages, foreign-language interfaces tend to become rich, complex, incomplete, and described only by example. The main contribution of this chapter is to provide (part of) a formal description of the interface. This precision encompasses not only the programmer's-eye view of the interface, but also its implementation. The bulk of this chapter is devoted to this description.

2.2 Background

The basic way in which almost any foreign-language interface works is by expressing the signature of each foreign-language procedure in some formal notation. From this signature, stub code is generated that *marshals* the parameters "across the border" between the two languages, calls the procedure using the foreign language's calling convention, and then unmarshals the results back across the border. Dealing with the different calling conventions of the two languages is usually the easy bit. The complications come in the parameter marshaling, which transforms data values built by one language into a form that is comprehensible to the other.

A major design decision is the choice of notation in which to describe the signatures of the procedures that are to be called across the interface. There are three main possibilities:

- *Use the host language (Haskell, in our case).* That is, write a Haskell type signature for the foreign function, and generate the stub code from it. Green Card uses this approach (Peyton Jones *et al.*, 1997), as does J/Direct (Mic, 1998) (Microsoft's foreign-language interface for Java).
- *Use the foreign language (say C).* In this case the stub code must be generated from the C prototype for the procedure. SWIG (Beazley, 1996) uses this approach.
- *Use a separate Interface Definition Language (IDL),* designed specifically for the purpose.

[1]For reasons of presentation, some FFI definitions given in this chapter may differ in syntax from the official standard.

We discuss the first two possibilities in Section 2.2.1 and the third in Section 2.2.2.

2.2.1 Using the host or foreign language

At first sight the first two options look much more convenient than the third, because the caller is written in one language and the callee in the other, so the interface is conveniently expressed for at least one of them. Here, for example, is how J/Direct allows Java to make foreign-language calls:

```
class ShowMsgBox {
  public static void main(String args[])
  {
    MessageBox(0,"Hello!","Java Messagebox",0);
  }

  /** @dll.import("USER32") */
  private static native
    int MessageBox( int hwndOwner, String text
                  , String title, int fuStyle
                  );
}
```

The `dll.import` directive tells the compiler that the Java `MessageBox` method will link to the native Windows `USER32.DLL`. The parameter marshaling (for example of the strings) is generated based on the Java type signature for `MessageBox`.

The fatal flaw is that *it is invariably impossible, in general, to generate adequate stub code based solely on the type signature of a procedure in one language or the other*. There are three kinds of difficulties.

1. First, some practically-important languages, notably C, have a type system that is too weak to express the necessary distinctions. For example:
   - The stub code generator must know the mode of each parameter — `in`, `in out`, or `out` — because each mode demands different marshaling code.
   - Some pointers have a significant `NULL` value while others do not. Some pointers point to values that can (and sometimes should) be copied across the border, while others refer to mutable locations whose contents must not be copied.
   - There may be important inter-relationships between the parameters. For example, one parameter might point to an array of values, while another gives the number of elements in the array, or worse, the number of bytes. The marshaling code needs to know about such dependencies.

2. On the other hand, it may not even be enough to give the signature in a language with an expressive type system, such as Haskell. The trouble is that the type signature still says too little about the foreign procedures type signature. For example, is the result of a Haskell procedure returned as the result of the foreign procedure, or via an out- parameter of that procedure? In the case of J/Direct, when a record is passed as an argument, Java's type signature is not enough to specify the layout of the record because Java does not specify the layout of the fields of an object.

3. The signature of a foreign procedure may say too little about allocation responsibilities. For example, if the caller passes a data structure to the callee (such as a string), can the latter assume that the structure will still be available after the call? Does the caller or callee allocate space to hold the results?

The predescessor to H/Direct, callled Green Card, used Haskell as the language in which to give the type signatures for foreign procedures (Peyton Jones *et al.*, 1997). To deal with the issues described above we provided ways of augmenting the Haskell type signature to allow the programmer to "customise" the stub code that would be generated. However, Green Card grew larger and larger – and we realised that what began as a modest design was turning into a full-scale language.

2.2.2 Using an IDL

Of course, we are not the first to encounter these difficulties. As part of the CORBA standard (Object Management Group, 1993), the OMG group defined a separate *Interface Definition Language* (IDL) to describe the signatures of procedures that are to be called across the border. Many variants have been derived and IDL's have become rich and complicated, for precisely the reasons described above, but they are at least somewhat standardised and come with useful tools. We focus on the IDL used to describe COM interfaces (Mic, 1992), which is closely based on DCE IDL(Ltd, 1993). We also provide support for the OMG IDL, using the translation from OMG to DCE IDL defined by (Vogel and Gray, 1995; Vogel *et al.*, 1996).

Like COM, but unlike CORBA[2], we take the view that the IDL for a foreign procedure defines a *language-independent, binary interface to the foreign procedure* — a sort of *lingua franca*. The interface thus defined is supposed to be complete: it covers calling convention, data format, and allocation rules. It may be necessary to generate stub code on both sides of the border, to marshal parameters into the IDL-mandated format, and then on into the format demanded by the foreign procedure. But these two chunks of marshaling code can be generated separately, each by a tool

[2]CORBA does not define a binary interface. Rather, each ORB vendor provides a *language binding* for a number of supported languages. This language binding essentially provides the marshaling required to an ORB-specific common calling convention. If you want to use a language that the ORB vendor does not support, you are out of luck.

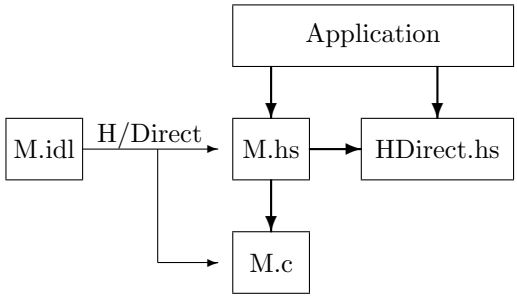

Figure 2.1: The big picture

specialised to its host language. By design, however, IDL's binary conventions are more or less identical to C's, so marshaling on the C side is hardly ever necessary.

Here, for example, is the IDL describing the interface to a function `foo`:

```
int foo( [out] long* l
       , [string, in] char* s
       , [in, out] double* d
       );
```

The parts in square brackets are called *attributes*. In this case they describe the mode of each parameter, but there are a rich set of further attributes that give further (and often essential) information about the type of the parameters. For example, the `string` attribute tells that the parameter `s` points to a null-terminated array of characters, rather than pointing to a single character.

2.2.3 Overview

The "big picture" is given by Figure 2.1. The interface between Haskell and the foreign language is specified in IDL. This IDL specification is read by *H/Direct*, which then produces Haskell and C[3] source files files containing Haskell and C *stub code*.

H/Direct can generate stub code that allows Haskell to call C, or C to call Haskell. It can also generate stub code that allows Haskell to create and invoke COM components, and that allows COM components to be written in Haskell. The interface to COM is rather involved and described seperately in the next chapter. Much of

[3] For the sake of definiteness we concentrate on C as the foreign language in this chapter.

the work in all four cases concerns the marshaling of data between C and Haskell, and that is what we concentrate on in this chapter.

Since *H/Direct* generates Haskell source code, it expresses the actual foreign language call (or entry for the inverse case) with the Haskell foreign function interface (Chakravarty (ed.), 2002). The FFI defines a `foreign` declaration that asks the Haskell implementation to generate code for a foreign-language call (or entry). The `foreign` declaration deals with the most primitive layer of marshaling, which is necessarily implementation dependent; *H/Direct* generates all the implementation-independent marshaling.

To make all this concrete, suppose we have the following IDL interface specification:

```
typedef struct { int x,y; } Point;

void Move( [in,out,ref] Point* p );
```

If asked to generate stub code to enable Haskell to call function `Move`, *H/Direct* will generate the following (Haskell) code:

```
data Point = Point { x,y::Int }
marshalPoint :: Point -> IO (Ptr Point)
marshalPoint = ...

unmarshalPoint :: Ptr Point -> IO Point
unmarshalPoint = ...

move :: Point -> IO Point
move p =
  do{ a <- marshalPoint p
    ; primMove a
    ; r <- unmarshalPoint a
    ; hdFree
    ; return r
    }
foreign import stdcall "Move"
  primMove :: Ptr Point -> IO ()
```

This code illustrates the following features:

- For each IDL declaration, *H/Direct* generates one or more Haskell declarations.
- From the IDL procedure declaration `Move`, *H/Direct* generates a Haskell function `move` whose signature is intended to be "what the user would expect".

In particular, the Haskell type signature is expressed using *"high-level" types*; that is, Haskell equivalents of the IDL types. For example, the signature for `move` uses the Haskell record type `Point`. The translation for a procedure declaration is discussed in Section 2.5.

- The body of the procedure marshals the parameters into their *"low-level" types*, before calling the "low-level" Haskell function `primMove`. The latter is defined using a `foreign` declaration; the Haskell implementation generates code for the call to the C procedure `Move`. Section 2.6 specifies the high-level and low-level type corresponding to each IDL type.

- A "low-level" type is still a perfectly first-class Haskell type, but it has the property that it can trivially be marshaled across the border. There is fixed set of primitive "low-level" types, including `Int`, `Float`, `Char` and so on. `Addr` is a low-level type that holds a raw machine address. The type constructor `Ptr` is just a synonym for `Addr`:

  ```
  type Ptr a = Addr
  addPtr :: Ptr a -> Int -> Ptr b
  ```

  The type argument to `Ptr` is used to allow *H/Direct* to document its output somewhat, by giving the "high-level" type that was marshaled into that `Addr`. Section 2.7 describes how high-level types are marshaled to and from their low-level equivalents.

- An IDL `typedef` declaration, will result in a corresponding Haskell type declaration together with some marshaling functions. In general, a marshaling function transforms a "high-level" Haskell value (in this case `Point`) into a "low-level" Haskell value (in this case `Ptr Point`). These marshaling functions are in the `IO` monad because, as we shall see, they often work imperatively by allocating some memory and explicitly filling it in, so as to construct a memory layout that matches the interface specification. The translations for `typedef` declarations are discussed in Section 2.8.

- The function `hdFree :: IO ()` releases all the memory allocated by the marshaling functions.

So much for our example. The difficulty is that IDL is a complex language, so it is not always straightforward to construct the Haskell type that will correspond to a particular IDL type, nor to generate correct marshaling code. (The former is important to the programmer, the latter only to *H/Direct* itself.) Our goal in this chapter is to give a systematic translation of IDL to Haskell stub code.

However, before we dive into the details of the IDL translation, we first describe the design of the low-level foreign function interface.

2.3 The Foreign Function Interface

Since *H/Direct* only generates Haskell source code, it expresses the actual foreign language call (or entry for the inverse case) with the low-level Haskell foreign function interface (FFI). The FFI has currently evolved into an addendum to the Haskell98 standard (Chakravarty (ed.), 2002) and is supported by the main Haskell systems.

We have carefully minimised what is required from the language implementation, while maximising the work done by H/Direct. In this way, any Haskell system that implements our extensions can interface with foreign languages, using the implementation-independent H/Direct to do most of the work.

Surprisingly, there are only four primitive operations needed to implement a full fledged foreign function interface:

- Call foreign code, supported by `foreign import` declarations.
- Export Haskell code that can be called from foreign code, supported by `foreign export` declarations.
- Manage pointers to foreign data, supported by *foreign object* pointers.
- Export pointers to Haskell data, supported by *stable* pointers.

2.3.1 Foreign static import and export

Earlier versions of GHC (the Glasgow Haskell Compiler) provided `ccall` (or even `casm`) to invoke a C procedure (Peyton Jones and Wadler, 1993). However, while this facility is (fairly) easy to support in a compiler that uses C as an intermediate language, it is a bit more difficult when using a native code generator, and well-nigh impossible when using an interpreter such as Hugs. Furthermore, it says nothing about how to allow C to call Haskell, or how to inter-operate with procedures with non-C calling conventions.

The new foreign function interface is much simpler. Here is an example of how to import a foreign procedure:

```
foreign import "hash32" hash :: Int -> IO Int
```

This `foreign` declaration is modelled directly on the `primitive` declaration that Hugs has supported for some time. The declaration defines the Haskell `IO` action

`hash` which, when invoked, will call the external procedure `hash32`. The implementation of `hash` also takes care of converting between the Haskell representation of an `Int` and the corresponding external representation.

The result of `hash` has type `IO Int` rather than `Int`, to signal that `hash` might perform some input/output or have some other side effect.

The range of types that can be passed to and from a foreign-imported procedure is deliberately restricted to the (small) set of primitive types. By a "primitive type" we mean one that cannot be defined in Haskell, such as `Int`, `Float`, `Char`. Only the language implementation knows the representation of primitive types, and so only the language implementation can marshall them. For all other types, such as lists or `Bool`, H/Direct is used to generate marshalling code. The same restriction applies to the other variants of `foreign` that we discuss later, for the same reasons.

2.3.2 Variations on the theme

We support several variants of the basic `foreign` declaration:

- The name of the external procedure can be omitted, in which case it defaults to the same as the Haskell procedure.

  ```
  foreign import hash :: Int -> IO Int
  ```

- If the programmer is sure that the foreign procedure is really a function — that is, it has no side effects — he can write the type as a non-`IO` type:

  ```
  foreign unsafe import "sin"
    sin :: Double -> Double
  ```

  The "`unsafe`" keyword highlights the fact that the programmer undertakes a proof obligation, namely that the function really is a function. We use this convention uniformly (also e.g. in `unsafePerformIO`), so that a programmer can find all his proof obligations by saying `grep unsafe`.

- By default, `foreign import` uses the C calling convention, but the convention can instead be specified explicitly:

  ```
  foreign unsafe import ccall "sin"
    sin :: Double -> Double
  ```

  We also support the standard calling convention (`stdcall`) used in Win32 environments.

- In many systems it is necessary to specify the (dynamic) library in which the external procedure can be found.

```
foreign unsafe import "MathLib" "sin"
  sin :: Double -> Double
```

A similar declaration allows the programmer to expose a Haskell function to the outside world:

```
foreign export "put_char" putChar :: Char -> IO ()
```

This exports a C-callable procedure `put_char` that in turn invokes the Haskell function `putChar`, marshalling the parameter appropriately. The calling convention can be specified, just as with `foreign import`, and a pure (non-I/O) Haskell function can be exported just as easily (no need for "`unsafe`" here):

```
foreign export fibonacci :: Int -> Int
```

Similar to the foreign import case, when the external name of the exposed function is ommited, it defaults to the same name as the Haskell function.

2.3.3 Stable pointers and foreign objects

It is often necessary to pass a Haskell value (pointer) to an external procedure. This raises two difficulties: first, the Haskell garbage collector cannot tell when the Haskell value is no longer required; and second, the value may be moved by the (copying) garbage collector. We solve both these problems by registering the Haskell value as a *stable pointer* (`StablePtr`). This registration (a) returns a stable value (a small integer) that names the value, and will not change during garbage collection, and (b) tells the garbage collector to retain the value until told otherwise. Subsequently, the stable pointer can be dereferenced (in Haskell) to recover the original Haskell value.

An exactly dual problem arises when we want to pass to a Haskell program a pointer to an external object (e.g. a file handle, `malloc`'d block, or COM interface pointer). Often, we would like to be able to call `fclose`, or `free`, on the external reference when the Haskell garbage collector finds that the resource is no longer required. Such "run this finalizer when the object dies" is called *finalization*. The FFI defines the `ForeignPtr` type to model these foreign pointers, together with functions that attach finalizers to these pointers.

The implementation of both extensions is rather subtle and is described extensively in a separate article (Peyton Jones *et al.*, 1999).

2.3.4 Dynamic import

The `foreign import` primitive is fine if we know the name of the C function we want to invoke. But sometimes we don't. Notably, when invoking a COM object, we start from an *interface pointer*, which points to a location that points to a virtual method table (we discuss this in detail in the next chapter). To invoke the method, we must fetch the address of the method from the virtual method table, and call it. `foreign import` simply doesn't do the job; it works fine for link-time or load-time binding, but not at all for run-time binding.

To address this deficiency, we first need a new primitive Haskell data type, `Addr`, that represents a machine address. (We could have used `Int`, but that seems unsavory.) Next, we extend `foreign import` with a `dynamic` attribute:

```
foreign import dynamic
  hashMethod :: Addr -> (Int -> IO Int)
```

This defines a Haskell function `hashMethod` with the specified type. Function `hashMethod` takes the address of the foreign procedure, which must be of type `Addr`, and returns a fully-fledged Haskell function that, when applied, will invoke the foreign procedure. Consider the following example:

```
do h  <- ...get addr of hash procedure...
       -- h has type Addr
  ; let hash = hashMethod h
  ; r1 <- hash 34
  ; r2 <- hash 39
  ; ...
```

`h` is the address of a suitable C procedure; `hashMethod` turns `h` into a Haskell function of type `Int -> IO Int`, which is then invoked twice. Of course, if `h` is bound to a bogus address terrible things will happen.

It is rather simple to implement `foreign import dynamic`. The only difference from the static version is that the call takes place to a supplied argument, rather than to a static label. This contrasts sharply with its dual, dynamic export, which we study next.

2.3.5 Dynamic export

Just as `foreign import` is inadequate in general, so is `foreign export`, for two reasons. First, `foreign export` only makes sense in a compiled setting, since its

effect is to generate a code label that is visible to a linker – an interpreter cannot reasonably implement `foreign export`.

Second, `foreign export` works on *top-level* functions. But we might want to export arbitrary functions. For example, external library procedures quite often take a *callback* parameter; that is, a pointer to a procedure that the external procedure will itself call. For example, the Win32 API provides a function that allows you to iterate over the current list of open windows:

```
typedef BOOL (*WNDENUMPROC)(HWND, LPARAM);
BOOL EnumWindows( WNDENUMPROC enumFunc
                , LPARAM lParam
                );
```

The system call takes a pointer to a callback procedure to invoke for each open window, together with a value `lparam` that we'll ignore for now. The callback procedure returns a boolean value to indicate whether we should stop iterating over the windows or not.

The system call itself can easily enough be imported into Haskell[4]

```
type BOOL   = Int
type LPARAM = Int
type WNDENUMPROC = Addr

foreign import "EnumWindows"
  enumWindows :: WNDENUMPROC -> LPARAM -> IO BOOL
```

But what to do with the callback? We want to implement it in Haskell, so the callback will have to be dressed up to appear like a C function pointer. One way would be to use `foreign export` to export a Haskell procedure as a C procedure, and add some mechanism to give Haskell access to the address of that C procedure, to pass to `enumWindows`.

But there is a much more elegant solution. We provide a `dynamic` form of `foreign export`, thus:

```
type HWND   = Addr
foreign export dynamic
   mkWndEnumProc :: (HWND -> LPARAM -> IO BOOL)
                 -> IO WNDENUMPROC
```

[4] We declare types `BOOL`, `LPARAM`, etc as Haskell type synonyms that mimic the C header file definitions of these types. Such type declarations are usually generated automatically by H/Direct.

This declaration defines a Haskell function `mkWndEnumProc`, with the type specified. Function `mkWndEnumProc` takes an *arbitrary Haskell function value* of the given type as its single argument, and returns a C function pointer. This C function expects to find two arguments on the C stack; it marshalls them into the Haskell world, and passes them to the Haskell function that was passed to `mkWndEnumProc`. Here is an example of its use[5]:

```
windowTitles :: IO [String]
windowTitles =
  do ref <- newIORef []
    ; let getTitle :: HWND -> LPARAM -> IO BOOL
          getTitle hwnd lp =
            do t  <- getWindowTitle hwnd
              ; ts <- readIORef ref
              ; writeIORef ref (t:ts)
              ; return (boolToInt True)

    ; cback <- mkWndEnumProc getTitle
    ; enumWindows cback (0::Int)
    ; readIORef ref
```

Here, `getTitle` is the callback procedure; it is called for each window, passing the window handle and the `LPARAM` value. It in turn calls `getWindowTitle` (another foreign-imported procedure) to get the window title, and puts it onto the front of a list of window titles, kept in a Haskell mutable variable `ref`.

The *Haskell* function `getTitle` is turned into a *C-callable* procedure `cback` (of type `Addr`) by `mkWndEnumProc`, the function defined by the `foreign export dynamic` declaration. Finally `cback` is passed to `enumWindows`.

We do not claim that this is beautiful programming style. For example, it is rather gruesome to use a mutable variable in `getTitle`. But the style is dictated by the architecture of Windows system calls; we are stuck with it. However, we are now ready to understand quite a bit about `foreign export dynamic`:

- The callback function `getTitle` is a first class Haskell value. It is not a top-level function, as must be the case for a static `foreign export`. In this case, `getTitle` has a free variable, `ref`, the mutable cell that it updates.

  This capability is modeled in C by the `LPARAM` parameter. The system call accepts `LPARAM` as well as the callback procedure, and passes `LPARAM` each time it calls the procedure. In effect, the (callback, `LPARAM`) pair constitutes a closure, of code plus environment.

[5]The Appendix introduces `IORefs`.

In this particular case, a C programmer would use `LPARAM` to point to a location in which the list is accumulated, just like `ref`. If there were many free variables, matters would be less simple. The Haskell programmer does not need to bother with `LPARAM` — indeed, `lp` is unused in the definition of `getTitle`. `mkWndEnumProc` captures a first-class Haskell value, free variables and all. Higher-order programming in C!

- `mkWndEnumProc` encapsulates a Haskell value as a C function pointer. To do this, it first registers the Haskell value as a stable pointer (Section 2.3.3), and then embeds the stable pointer in the C function. The programmer can explicitly free the retained Haskell value using:

  ```
  freeHaskellFunctionPtr :: Addr -> IO ()
  ```

  This operation cannot be done automatically, since it depends on knowing that the exported function pointer is no longer needed externally.

- As with the other `foreign` declaration variants, a `foreign export dynamic` also allows you to specify which calling convention the returned function pointer should expect.

2.3.6 Implementing dynamic export

Dynamic export is considerably harder to implement than dynamic import, because we have to generate a C function pointer *that cannot be static*, because it must somehow refer to the Haskell function it encapsulates. This forces us to perform a little bit of dynamic code generation.

Our implementation for the Glasgow Haskell Compiler works by taking advantage of the *static* version of `foreign export`. Here, for example, is how we implement `mkWndEnumProc`. We repeat its declaration here:

```
foreign export dynamic
  mkWndEnumProc :: (HWND -> LPARAM -> IO BOOL)
                -> IO WNDENUMPROC
```

GHC first generates code exactly as if the programmer had written:

```
foreign export
  wndEnumProc :: HWND -> LPARAM
              -> StablePtr (HWND->LPARAM->IO BOOL)
              -> IO BOOL
wndEnumProc h l sp =
  do f <- deRefStablePtr sp
    ; f h l
```

`wndEnumProc` takes an extra argument, a stable pointer to the function value (Section 2.3.3); it dereferences the stable pointer, and calls the function it gets back. Now, GHC generates code for `mkWndEnumProc`, which does three things:

- registers the Haskell function as a stable pointer;
- dynamically generates a code fragment;
- returns the address of this dynamically generated code.

The dynamically-generated code consists of two or three instructions:

```
add-param <function pointer>
jump wndEnumProc
```

The `add-param` "instruction" must be whatever machine code is necessary to pass one extra parameter — often this is just a matter of pushing it on the stack (perhaps also moving the return address). Once this is done, the statically-exported `wndEnumProc` will do the rest. Clearly, the dynamic-code-generation part is highly architecture dependent, but it is also very short, and is not hard in practice.

Unfortunately, this solution won't work at all for the Hugs interpreter, because an interpreter can't support static `foreign export`. Instead, the Hugs implementation of `mkWndEnumProc` dynamically generates the following segment of machine code:

```
push <function pointer>
push <type descriptor>
jump GenericCaller
```

Here `<type descriptor>` is a (pointer to a C-format) string that encodes the type signature of the function. The `<function pointer>` is a stable pointer to the Haskell function value, as before. Finally, `GenericCaller` is a fixed piece of code that (a) uses the type descriptor to marshall data from C to Haskell, (b) calls the specified Haskell function, (c) marshalls the Haskell result back, and (d) returns to the C caller. `GenericCaller` is highly machine dependent, since it must know all about the caller's calling conventions; but at least it need only be written once.

2.3.7 Related work

Foreign function interfaces (FFIs) are clearly of great use, but not may articles have been written about them. Most functional programming systems provide a FFI, allowing calls to external functions to be embedded within functional code. However,

few provide equally good support for the outside to call in. The `esh` Scheme implementation (Rose and Muller, 1992) is a notable exception; it was designed with the explicit goal of making hybrid Scheme and C/C++ applications easier to write. Another, more recent system is the Bigloo Scheme compiler (Serrano, 1999).

For ML-based languages, the Standard ML of New Jersey compiler's foreign function interface also provides support for call-ins (Huelsbergen, 1996). Function closures can be dressed up behind a C function pointer, which can then be passed out to the outside world, making it similar in power to `foreign export dynamic`.

A similar approach is provided by the Objective Caml FFI (Leroy, 1996), which requires exported functions to be registered by giving them a name (an arbitrary string) from within OCaml code. The run-time system provides a C callable entry point for looking up the OCaml function closure that hides behind a name, and invoke through a class of invocation functions. This scheme requires that the user makes up the difference using C, writing a little bit of stub code that does the lookup and invokes the function by marshalling and unmarshalling the arguments and results. Contrast this with `foreign export dynamic` which makes the Haskell-nature of the function pointers it returns transparent to the user.

To our knowledge, the only other Haskell system that provides support for Haskell functions that can be called externally is the NHC 1.3 compiler (Wallace, 1998), which provides a basic export mechanism similar to that of Objective Caml's.

2.4 Translating IDL to Haskell

H/Direct generates Haskell marshaling code from an IDL description. However, IDL is a complex language and it is not always straightforward to perform the translation from IDL types to Haskell types, nor to generate correct marshaling code. (The former is important to the programmer, the latter only to *H/Direct* itself.) The rest of this chapter is concerned with a systematic and formal translation of IDL to Haskell marshaling code. We have found this detailed treatment invaluable, both for designing the system and in the implementation of *H/Direct*.

To simplify translation we assume that the IDL source is brought into a standard form, that is, we factor the translation into a translation of full IDL to a core subset and a translation from core IDL to Haskell. In particular, we assume that: `out` parameters always have an explicit "`*`", the pointer default is manifested in all pointer types, and all enumerations have value fields. The details are unimportant here and normalisation is straighforward to implement (although one has to take care of many details and dialects).

IDL is a large language, and space precludes giving a complete translation here. We do not even give a syntax for IDL, relying on the left-hand sides of the transla-

tion rules to specify the syntax we treat. However, the framework we give here is sufficient to treat the whole language, and our implementation does so.

2.5 Procedure declarations

The translation function $\mathcal{D}[\![\ ]\!]$ maps an IDL declaration into one or more Haskell declarations. We begin with IDL procedure declarations. To start with, we concentrate on allowing Haskell to call C; we discuss other variants in Section 2.9. Here is the translation rule for procedure declarations:

```
  D[[t_res f([in]t_in, [out]t_out, [in,out]t_inout)]]
↦
  T[[f]] :: T[[t_in]] -> T[[t_inout]]
       -> IO (T[[t_out]],T[[t_inout]],T[[t_res]])
  N[[f]] = \m -> \n ->
    do { a <- M[[t_in]] m
       ; b <- O[[t_out]]
       ; c <- M[[t_inout]] n
       ; r <- primN[[f]] a b c
       ; x  <- U[[t_out]] b
       ; y  <- U[[t_inout]] c
       ; z   <- U[[t_res]] r
       ; hdFree
       ; return (x,y,z)
       }

  foreign import stdcall primN[[f]]
    :: B[[t_in]] -> B[[t_out]] -> B[[t_inout]]
    -> IO B[[t_res]]
```

Despite our claim of formality, the fully formal version of this rule has an inconvenient number of subscripts. Instead, we illustrate by giving one parameter of each mode (`[in]`, `[out]`, and `[in, out]`); more complex cases are handled exactly analogously. The translation produces a Haskell function that takes one argument for each IDL `[in]` or `[in, out]` parameter, and returns one result of each IDL `[out]` or `[in, out]` parameter, plus one result for the IDL result (if any). In general, foreign functions can perform side effects, so the result type is in the `IO` monad. We have added a (non- standard) attribute `[pure]`, that declares the procedure to have no side effects; in this case, the Haskell procedure can simply return a tuple rather than an `IO` type.

The generic translation for procedure declaration uses several auxiliary translation schemes:

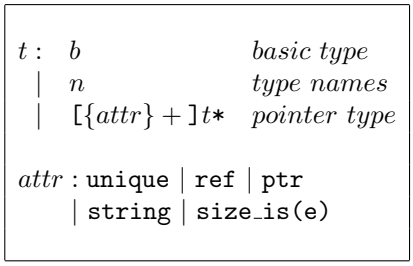

Figure 2.2: IDL type syntax

- The translation scheme $\mathcal{T}[\![t]\!]$ gives the "high-level" Haskell type corresponding to the IDL type t.
- The translation scheme $\mathcal{N}[\![n]\!]$ does the name mangling required to translate IDL identifiers to valid Haskell identifiers. For example, it accounts for the fact that Haskell function names must begin with a lower-case letter.
- The translation scheme $\mathcal{B}[\![t]\!]$ gives the "low-level" Haskell type corresponding to the IDL type t.
- The translation scheme $\mathcal{M}[\![t]\!]$ `::` $\mathcal{T}[\![t]\!]$ `->` `IO` $\mathcal{B}[\![t]\!]$ will generate Haskell code that marshals a value of IDL type t from its high-level type $\mathcal{T}[\![t]\!]$ to its low-level form $\mathcal{B}[\![t]\!]$. This is used to marshal all the in-parameters of the procedure (`[in]` and `[in,out]`).
- The translation scheme $\mathcal{U}[\![t]\!]$ `::` $\mathcal{B}[\![t]\!]$ `->` `IO` $\mathcal{T}[\![t]\!]$ generates Haskell code that unmarshals a value of IDL type t. This is used to unmarshal all the out-parameters of the procedure, and its result (if any). $\mathcal{M}[\![\ ]\!]$ and $\mathcal{U}[\![\ ]\!]$ are mutual inverses (up to memory allocation).
- In addition, for `[out]` parameters the caller is required to allocate a location to hold the result. $\mathcal{O}[\![$`[`$attr$`]`t`*`$]\!]$ `::` `IO` `(Ptr` $\mathcal{B}[\![t]\!]$`)` is Haskell code that allocates enough space to contain a value of IDL type t.

2.6 Mapping for types

Next, we turn our attention to the translations $\mathcal{T}[\![\ ]\!]$ and $\mathcal{B}[\![\ ]\!]$ that translate IDL types to Haskell types. The syntax of IDL types that we treat is given in Figure 2.2, while Figure 2.3 gives their translation into Haskell types. We deal with user-defined structured types later, in Section 2.8.

Translating base types that have direct Haskell analogues, is easy. The high-level and low-level type translations coincide, except that the high-level representation of IDL's 8-bit characters is Haskell's 16 bit `Char` type. To give more precise mapping

$$
\begin{array}{rcl}
\mathcal{B}[\![\texttt{short}]\!] & \mapsto & \texttt{Int32} \\
\mathcal{B}[\![\texttt{unsigned short}]\!] & \mapsto & \texttt{Word32} \\
\mathcal{B}[\![\texttt{float}]\!] & \mapsto & \texttt{Float} \\
\mathcal{B}[\![\texttt{double}]\!] & \mapsto & \texttt{Double} \\
\mathcal{B}[\![\texttt{char}]\!] & \mapsto & \texttt{Word8} \\
\mathcal{B}[\![\texttt{wchar}]\!] & \mapsto & \texttt{Char} \\
\mathcal{B}[\![\texttt{boolean}]\!] & \mapsto & \texttt{Bool} \\
\mathcal{B}[\![\texttt{void}]\!] & \mapsto & \texttt{()} \\
\mathcal{B}[\![\texttt{[}attr\texttt{]}\,t\texttt{*}]\!] & \mapsto & \texttt{Ptr}\ \mathcal{T}[\![t]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T}[\![\texttt{char}]\!] & \mapsto & \texttt{Char} \\
\mathcal{T}[\![b]\!] & \mapsto & \mathcal{B}[\![b]\!] \\
\mathcal{T}[\![n]\!] & \mapsto & \mathcal{N}[\![n]\!] \\
\mathcal{T}[\![\texttt{[ref]}\,t\texttt{*}]\!] & \mapsto & \mathcal{T}[\![t]\!] \\
\mathcal{T}[\![\texttt{[unique]}\,t\texttt{*}]\!] & \mapsto & \texttt{Maybe}\ \mathcal{T}[\![t]\!] \\
\mathcal{T}[\![\texttt{[ptr]}\,t\texttt{*}]\!] & \mapsto & \texttt{Ptr}\ \mathcal{T}[\![t]\!] \\
\mathcal{T}[\![\texttt{[string]char*}]\!] & \mapsto & \texttt{String} \\
\mathcal{T}[\![\texttt{[size\_is(}v\texttt{)]}\,t\texttt{*]}]\!] & \mapsto & \texttt{[}\mathcal{T}[\![t]\!]\texttt{]}
\end{array}
$$

Figure 2.3: Type translations

we have extended Haskell with new base types: `Word8`, `Word16`, and so on. Similarly, IDL type names are translated to the (Haskell-mangled) name of the corresponding Haskell type.

Matters start to get murkier when we meet pointers. Since a pointer is always passed to and from C as a machine address, the low-level translation of all pointer types is a raw machine address:

$$\mathcal{B}[\![\texttt{[}attr\texttt{]}\,t\texttt{*}]\!] \quad \mapsto \quad \texttt{Ptr}\ \mathcal{T}[\![t]\!]$$

(Recall that `Ptr` t is an abbreviation for `Addr`, but the `Ptr` form is somewhat more informative.)

In contrast, the high-level translation of pointers depends on what type of pointer is concerned. IDL has no fewer than five kinds of pointer, distinguished by their attributes! We treat them one at a time (refer in each case to Figure 2.3):

- A value of IDL type `[ref]`t`*` is the unique pointer, or indirection, to a value of type t. A value of type `[ref]`t`*` should be marshalled by copying the structure over the border. Since pointers are implicit in Haskell, the corresponding high-level Haskell type is $\mathcal{T}[\![t]\!]$.

- The IDL type `[unique]`t`*` is exactly the same as `[ref]`t`*`, except that the pointer can be `NULL`. The natural way to represent this possibility in Haskell is using the `Maybe` type. The latter is a standard Haskell type defined like this:

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

- An IDL value of type `[ptr]`t`*` is the address of a value that might be shared, and might contain cycles. It is far from clear how such a thing should be marshaled, so we adopt a simple convention:

  $$\mathcal{T}[\![\texttt{[ptr]}\,t\texttt{*}]\!] \mapsto \texttt{Ptr}\ \mathcal{T}[\![t]\!]$$

  That is, `[ptr]` values are not moved across the border at all. Instead they are represented by a value of type `Ptr` $\mathcal{T}[\![t]\!]$, a raw machine address.

  This is often useful. For a start, some libraries implement an abstract data type, in which the client is expected to manipulate only pointers to the values. Similarly, COM interface pointers should be treated simply as addresses. Finally, some operating system procedures (notably those concerned with windows) return such huge structures that a client might want to marshal them back selectively.

- A value of type `[string]char*` is the address of a null-terminated sequence of characters. (Contrast `[ref]char*`, which is the address of a single character.) The corresponding Haskell type is, of course, `String`. The `[string]` attribute applies to the following array types `char`, `byte`, `unsigned short`, `unsigned long`, `structs` with `byte` (only!) fields and, in Microsoft-only IDL, `wchar`.

- Sometimes a procedure takes a parameter that is a pointer to an array of values, where another parameter of the procedure gives the size of the array. (CORBA IDL calls such arguments "sequences".) For example:

  ```
  void DrawPolygon
    ( [in,size_is(nPoints)] Point* points
    , [in] int nPoints
    );
  ```

  The `[size_is(nPoints)]` attribute tells that the second parameter, `nPoints`, gives the size of the array. (This is quite like the `[string]` case, except that the size of the array is given separately, whereas strings have a sentinel at the end.) There is a second variant in which `nPoints` is a static constant, rather than the name of another parameter.

  At the moment we translate an IDL array to a Haskell list, but another possibility would be to translate it to a Haskell array.

While each of these variants has a reasonable rationale, we have found the plethora of IDL pointer types to be a rich source of confusion. The translations in Figure 2.3

look innocuous enough, but we have found them extremely helpful in clarifying and formalising just exactly what the translation of an IDL type should be.

Even if the translations are not quite "right" (whatever that means), we now have a language in which to discuss variants. For example, it may eventually turn out that the IDL `[ptr]` attribute is conventionally used for subtly different purposes than the ones we suggest above. If so, the translations can readily be changed, and the changes explained to programmers in a precise way.

2.7 Marshaling

In the translation of the IDL type signature for a procedure (Section 2.5), we invoked marshaling functions $\mathcal{M}[\![\ ]\!]$ and $\mathcal{U}[\![\ ]\!]$ for each of the types involved. Now that we have defined the high and low-level translations of each type, the marshaling code is relatively easy to define. In this section we define these marshaling functions.

Marshaling a structured value consists, as we shall see, of two steps: allocate some memory in the *parameter-marshaling area* to hold the value, and then actually marshal the Haskell value into that memory. The translations are much more elegant if we define auxiliary schemes, $\mathcal{W}[\![\ ]\!]$ and $\mathcal{R}[\![\ ]\!]$, that perform this "by-reference" marshaling. We also need a number of functions to manipulate the parameter-marshaling area. More precisely:

$\mathcal{W}[\![t]\!]$ `::` `Ptr` $\mathcal{T}[\![t]\!]$ `->` $\mathcal{T}[\![t]\!]$ `->` `IO ()` will marshall its second argument into the memory location(s) pointed to by its first argument; the latter is a raw machine address.

$\mathcal{R}[\![t]\!]$ `::` `Ptr` $\mathcal{T}[\![t]\!]$ `->` `IO` $\mathcal{T}[\![t]\!]$ unmarshals a value of IDL type t out of memory location(s) pointed to by its argument. $\mathcal{W}[\![\ ]\!]$ and $\mathcal{R}[\![\ ]\!]$ are mutually inverse (up to memory allocation).

$\mathcal{S}[\![t]\!]$ `::` `Int` is the number of bytes occupied by an IDL value of type t. The function $\mathcal{O}[\![\ ]\!]$, mentioned in Section 2.5, is defined thus:

$$\mathcal{O}[\![\texttt{[}\mathit{attr}\texttt{]}\,t\texttt{*}]\!] \quad \mapsto \quad \texttt{hdAlloc}\ \mathcal{S}[\![t]\!]$$

`hdAlloc :: Int -> IO (Ptr a)` allocates the specified number of bytes in the parameter-marshaling area, returning a pointer to the allocated area.

`hdWrite`b `::` `Ptr` $\mathcal{T}[\![b]\!]$ `->` $\mathcal{T}[\![b]\!]$ `->` `IO ()`, where b is a basic type, marshals a value of IDL type b into the specified memory location(s).

`hdRead`b `::` `Ptr` $\mathcal{T}[\![b]\!]$ `->` `IO` $\mathcal{T}[\![b]\!]$, where b is a basic type, unmarshals a value of IDL type t.

`hdFree :: IO ()` frees the whole parameter-marshaling area.

```
M⟦t⟧ :: T⟦t⟧ -> IO B⟦t⟧

M⟦char⟧          ↦  marshalChar
M⟦b⟧             ↦  return
M⟦n⟧             ↦  marshaln
M⟦[ref]t*⟧       ↦  \x ->
                    do{ px <- hdAlloc S⟦t⟧
                      ; W⟦t⟧ px x}
M⟦[unique]t*⟧    ↦  \x ->
                    case x of
                      Nothing -> return nullPtr
                      Just y -> M⟦[ref]t*⟧ y
M⟦[ptr]t*⟧       ↦  return
M⟦[string]t*⟧    ↦  marshalString
```

```
W⟦t⟧ :: Ptr T⟦t⟧ -> T⟦t⟧ -> IO ()

W⟦b⟧             ↦  hdWriteb
W⟦[attr]t*⟧      ↦  \p x ->
                    do{ a <- M⟦[attr]t*⟧ x
                      ; hdWriteAddr p a}
```

```
U⟦t⟧ :: B⟦t⟧ -> IO T⟦t⟧

U⟦char⟧          ↦  unmarshalChar
U⟦b⟧             ↦  return
U⟦n⟧             ↦  unmarshaln
U⟦[ref]t*⟧       ↦  R⟦t⟧
U⟦[unique]t*⟧    ↦  \p ->
                    if p == nullPtr then
                      return Nothing
                    else
                      do{ x <- R⟦t⟧ p
                        ; return (Just x)}
U⟦[ptr]t*⟧       ↦  return
U⟦[string]t*⟧    ↦  unmarshalString
```

```
R⟦t⟧ :: Ptr T⟦t⟧ -> IO T⟦t⟧

R⟦b⟧             ↦  hdReadb
R⟦[attr]t*⟧      ↦  \p ->
                    do{ a <- hdReadAddr p
                      ; U⟦[attr]t*⟧ a}
```

Figure 2.4: The marshaling schemes

```
t :  t[e]                                          array type
  |  enum {
       tag_1 = v_1, ..., tag_n = v_n}              enumeration
  |  struct tag {
       f_1 : t_1; ...; f_n : t_n;}                 record type
  |  union tag_1
       switch ( b tag_2 ) {
       case v_1:t_1 f_1; ... case v_n:t_n f_n;}    union type
```

Figure 2.5: IDL constructed type syntax

With these definitions in mind, Figure 2.4 gives the marshaling schemes. We omit the schemes for `[size_is]` because it is tiresomely complicated. Apart from that, the translations are easy to read:

- For basic types there is no marshaling to do, except that we convert between the 16-bit Haskell `Char` and 8-bit IDL `char` types.
- Marshaling a `typedef`'d type can be done by invoking its marshaling function.
- Marshaling a `[ref]` pointer is done by allocating some memory with `hdAlloc`, and then marshaling the value into it with $\mathcal{W}[\![\ ]\!]$. Unmarshaling is similar, except that there is no allocation step; we just invoke $\mathcal{R}[\![\ ]\!]$.
- Dealing with `[unique]` pointers is similar, except that we have to take account of the possibility of a `NULL` value.

Again, it is very helpful to have a precise language in which to discuss these translations. Though they look simple, we can attest that it is very easy to get confused by pointers to pointers to things, and we have far greater confidence in our implementation as a result of writing the translations formally.

One might wonder about the run-time cost of all this data marshalling. Indeed, historically foreign-language interfaces have taken it for granted that data is *not* copied across the border. However, such non-marshalling interfaces are extremely restrictive: they require the two languages to share common data representations to the bit level, and to share a common address space. In moving decisively towards IDL-based component-based programming, the industry has accepted the performance costs of marshalling in exchange for its flexibility. This in turn discourages very fine-grain, intimate interaction between components with many border-crossings, instead encouraging a coarser-grain approach. We are happy to adopt this trend, because there is no way to make (lazy) Haskell and C share data representations.

2.8 Type declarations

On top of the primitive base types, IDL supports the definition of a number of constructed types. For example

```
typedef int trip[3];
typedef struct TagPoint { int x,y; } Point;
typedef enum { Red=0, Blue=1, Green=2 } RGB;
typedef union _floats switch (int ftype) {
  case 0: float  f;
  case 1: double d;
} Floats;
```

which declares array, record, enumeration and union (or sum) types, respectively. Figure 2.5 shows the syntax of IDL's constructed types.

The translation provides rules for converting between IDL constructed types into corresponding Haskell representations. To ease the task of defining this type mapping, we assume that each constructed type appears as part of an IDL type declaration. In general, a type declaration has the following form:

```
typedef t name;
```

declaring *name* to be a synonym for the type t, which is either a base type or one of the above constructed types. A type declaration for an IDL type t gives rise to the definition of the following Haskell declarations:

- A Haskell type declaration for the Haskell type $\mathcal{N}[\![name]\!]$. It is defined in such a way that $\mathcal{T}[\![name]\!] = \mathcal{N}[\![name]\!]$.
- `marshal`$\mathcal{N}[\![name]\!]$ `::` $\mathcal{T}[\![name]\!]$ `->` `IO` $\mathcal{B}[\![t]\!]$ this implements the translation scheme $\mathcal{M}[\![\ ]\!]$ for converting from the Haskell representation $\mathcal{T}[\![t]\!]$ to the IDL type t.
- `unmarshal`$\mathcal{N}[\![name]\!]$ `::` $\mathcal{B}[\![t]\!]$ `->` `IO` $\mathcal{T}[\![name]\!]$ which implements the dual $\mathcal{U}[\![\ ]\!]$ scheme for unmarshaling.
- `marshal`$\mathcal{N}[\![name]\!]$`At :: Ptr` $\mathcal{B}[\![t]\!]$ `->` $\mathcal{T}[\![name]\!]$ `->` `IO ()` for types that are marshalled by reference.
- `unmarshal`$\mathcal{N}[\![name]\!]$`At :: Ptr` $\mathcal{B}[\![t]\!]$ `->` `IO` $\mathcal{T}[\![name]\!]$ which implements the $\mathcal{R}[\![\ ]\!]$ scheme for unmarshaling a constructed type by-reference.
- `sizeof`$\mathcal{N}[\![name]\!]$ `::` `Int`, a constant holding the size of the external representation of the type (in 8-bit bytes.)

$\mathcal{D}[\![$`typedef` t $name$`;`$]\!]$
 $\mapsto$ `type` $\mathcal{N}[\![name]\!]$ `=` $\mathcal{T}[\![t]\!]$
  `marshal`$\mathcal{N}[\![name]\!]$ `=` `marshal`$\mathcal{T}[\![t]\!]$
  `marshal`$\mathcal{N}[\![name]\!]$`At` `=` `marshal`$\mathcal{T}[\![t]\!]$`At`
  `unmarshal`$\mathcal{N}[\![name]\!]$ `=` `unmarshal`$\mathcal{T}[\![t]\!]$
  `unmarshal`$\mathcal{N}[\![name]\!]$`At` `=` `unmarshal`$\mathcal{T}[\![t]\!]$`At`
  `sizeof`$\mathcal{N}[\![name]\!]$ `=` $\mathcal{S}[\![t]\!]$

$\mathcal{D}[\![$`typedef` t $name$`[`dim`];`$]\!]$
 $\mapsto$ `type` $\mathcal{N}[\![name]\!]$ `= [` $\mathcal{T}[\![t]\!]$ `]`
  `marshal`$\mathcal{N}[\![name]\!]$ `=` `marshalArray` dim `marshal`$\mathcal{T}[\![t]\!]$`At`
  `marshal`$\mathcal{N}[\![name]\!]$`At` `=` `marshalArrayAt` dim `marshal`$\mathcal{T}[\![t]\!]$`At`
  `unmarshal`$\mathcal{N}[\![name]\!]$ `=` `unmarshalArray` dim `unmarshal`$\mathcal{T}[\![t]\!]$`At`
  `unmarshal`$\mathcal{N}[\![name]\!]$`At` `=` `unmarshalArrayAt` dim `unmarshal`$\mathcal{T}[\![t]\!]$`At`
  `sizeof`$\mathcal{N}[\![name]\!]$ `=` dim `*` $\mathcal{S}[\![t]\!]$

Figure 2.6: Translating simple type declarations

The general rules for converting type declarations into Haskell types is presented in Figure 2.6 and Figure 2.7. Here is what they generate when applied:

- In the case of a type declaration for a base type, this merely defines a type synonym. For example

  ```
  typedef int year;
  ```

  is translated into the type synonym

  ```
  type Year = Int
  ```

  plus marshaling functions for `Year`.

- For a record type such as `Point`:

  ```
  typedef struct TagPoint {int x,y;} Point;
  ```

  generates a single constructor Haskell data type:

  ```
  data Point = TagPoint { x:: Int, y::Int }
  ```

  In addition to this, the $\mathcal{D}[\![\ ]\!]$ scheme generates a collection of marshaling functions, including `marshalPoint`:

```
𝒟⟦typedef struct tag{...; t_i field_i; ...} name;⟧
 ↦
  data 𝒩⟦name⟧ = 𝒩⟦tag⟧{ ...,𝒩⟦field_i⟧ :: 𝒯⟦t_i⟧, ...}
  marshal𝒩⟦name⟧ rec = do
   ptr <- hdAlloc 𝒮⟦name⟧
   marshal𝒩⟦name⟧At ptr rec
   return ptr

  marshal𝒩⟦name⟧At ptr (𝒩⟦tag⟧{ ... ,𝒩⟦field_i⟧, ...} = do
   let ptr_1 = addPtr ptr 0
    ...
   let ptr_i = addPtr ptr_(i-1) 𝒮⟦t_(i-1)⟧
   𝒲⟦t_i⟧ ptr_i field_i
    ...
   return ()

  unmarshal𝒩⟦name⟧ = unmarshal𝒩⟦name⟧At

  unmarshal𝒩⟦name⟧At ptr = do
   let ptr_1 = addPtr ptr 0
    ...
   let ptr_i = addPtr ptr_(i-1) 𝒮⟦t_(i-1)⟧
   𝒩⟦field_i⟧ <- ℛ⟦t_i⟧ ptr_i
    ...
   return (𝒩⟦tag⟧ ... 𝒩⟦field_i⟧ ... )
  sizeof𝒩⟦name⟧ = structSize [...,𝒮⟦field_i⟧,...]

𝒟⟦typedef enum {...,alt = value,...} name;⟧
 ↦
  data 𝒩⟦name⟧ = ... | 𝒩⟦alt⟧ |...
  marshal𝒩⟦name⟧ x =
   case x of {...; 𝒩⟦alt⟧ -> 𝒩⟦value⟧;...}
  unmarshal𝒩⟦name⟧ x =
   case x of {...; 𝒩⟦value⟧ -> return 𝒩⟦alt⟧;...}
  unmarshal𝒩⟦name⟧At ptr = do
   v <- hdReadInt ptr
   unmarshal𝒩⟦name⟧ v
  sizeof𝒩⟦name⟧ = sizeofint
```

Figure 2.7: Translating type declarations

```
marshalPoint :: Point -> IO (Ptr Point)
marshalPoint (Point x y) =
  do{ ptr <- hdAlloc sizeofPoint
    ; let ptr1 = addPtr ptr 0
    ; marshalintAt ptr1 x
    ; let ptr2 = addPtr ptr1 sizeofint
    ; marshalintAt ptr2 y
    ; return ptr
    }
```

It marshals a `Point` by allocating enough memory to hold the external representation of the point. The size of the record type is computed as follows:

```
sizeofPoint :: Int32
sizeofPoint = structSize [sizeofint,sizeofint]
```

where `structSize` is a (platform specific) function that computes the size of a `struct` given the field sizes.[6]

`Point`'s two fields are marshaled into the external representation of `Point` by calling the by-reference marshaler for the basic type `Int`, supplying a pointer that has been appropriately offset.

- For the union type example given at the start of Section 2.8, the following Haskell type is generated:

  ```
  data Floats = F Float | D Double
  ```

  together with actions for marshaling between the algebraic type and a union (omitting the type signatures for the by-reference marshalers):

  ```
  marshalFloats   :: Floats -> IO (Ptr Floats)
  unmarshalFloats :: Ptr Floats -> IO Floats
  ```

  The external representation of a union is normally a `struct` containing the discriminant and enough room to accommodate the largest member of the union. In the case of `Floats`, the external representation must be large enough to contain an `int` and a `double`.

- Enumerations have a direct Haskell equivalent as algebraic data types with nullary constructors. For example, the `RGB` declaration:

  ```
  typedef enum {red=0,green=1,blue=2} RGB;
  ```

  is translated into the Haskell type

[6] Similarly, a function that returns the offsets at which to marshal each field into is also provided. Due to lack of space, `marshalPoint` makes the simplifying assumption that structures contain no internal padding.

```
data RGB = Red | Green | Blue
```

with concrete representation $\mathcal{B}[\![\texttt{RGB}]\!] = \texttt{Int32}$

The marshaling actions simply map between the nullary constructors and `Int32`:

```
marshalRGB :: RGB -> IO Int32
marshalRGB nm =
  return (case nm of {
              Red   -> 0
              Green -> 1
              Blue  -> 2 })

unmarshalRGB :: Int32 -> IO RGB
unmarshalRGB v =
 case v of
   0 -> return Red
   1 -> return Green
   2 -> return Blue
   _ -> fail (userError ...)
```

Haskell data structures can contain shared sub-components, or even cycles. However, such sharing is not observable by a Haskell program, so the marshalling code cannot take account of it. DAGs are therefore marshalled just as if they were trees, and a cyclic data structure (which is indistinguishable from an infinite data structure) make the marshaler fail to terminate.

It might be possible to "fix" these shortcomings, but we are not unduly bothered about them. Rather than marshal complex data structures (whether or not they contain sharing) across the border, a better approach is usually to leave them where they are and instead marshal a pointer to the data structure. When a component technology such as COM is being used (chapter 3), the right thing to do is to marshal an interface pointer, through which the client can access the data structure.

In short, if loss of sharing is a worry then you are probably marshalling too much data; we look forward to learning from experience whether this viewpoint is "right".

2.9 The inverse mapping

Once marshaling and unmarshaling functions are defined for each data type, it is not hard to reverse the mapping and build code that allows C to call Haskell. The translation for a `typedef` remains unchanged, but the translation for an IDL procedure declaration is reversed. Since the procedure is being implemented in

Haskell, its `[in]`-parameters are unmarshaled, the Haskell procedure is called, its results are marshaled, and returned to the caller. (We omit the details, but the translation rule can be expressed just as we did in Section 2.5.) For example, the `Move` IDL declaration of that Section would be compiled to the following Haskell code:

```
foreign export stdcall "Move"
  primMove :: Ptr Point -> IO ()

primMove a =
  do { p <- unmarshalPoint a
     ; q <- move p
     ; marshalPointAt a q
     ; return ()
     }

move :: Point -> IO Point
move = error "Not yet implemented"
```

The `foreign export` declaration asks the Haskell compiler to make `Move` externally callable with a `stdcall` interface. `primMove` does the marshaling, before calling `move`, which should be provided by the programmer.

2.10 Future work

Despite the extensive capabilities of H/Direct, it is still not widely used in practice. Perhaps surprisingly, one of the main culprits is the use of IDL! When integrating with large (C++) libraries, one has to manually translate all the signatures to IDL. A tool to remedy this situation is C-to-Haskell (Chakravarty, 2000) that reads C signatures directly. However, it suffers from exactly the same problems as described in section 2.2.1 and it has a special annotations language to specify marshalling hints. We think that a better way to approach this problem is to extend H/Direct to read both C signatures directly (as DCE IDL) with default IDL attributes, and to provide a separate attribute specification file that overrides default attributes for certain functions or classes. With this approach, many libraries could be processed almost automatically without losing the benefits of a formal semantics.

2.11 Conclusions

H/Direct was the fourth attempt at a foreign-language interface for Haskell. The first was `ccall`, a limited and low-level extension roughly equivalent to `foreign`

`import` (Peyton Jones and Wadler, 1993). The second was Green Card, which gradually turned into a domain-specific language (Peyton Jones *et al.*, 1997). The third was a pre-cursor to *H/Direct*, Red Card, which was specifically aimed at interfacing Haskell to COM objects, (Peyton Jones *et al.*, 1998; Leijen, 1998). *H/Direct* embodies the lessons we have learned – strive for implementation-independence and avoid inventing new languages.

We do not claim great originality for these observations. What is new in this chapter is a much more precise description of the mapping between Haskell and IDL than is usually given. This precision has exposed details of the mapping that would otherwise quite likely have been mis-implemented. Indeed, the specification of how pointers are translated exposed a bug in our current implementation of *H/Direct*. It also allows us automatically to support nested structures and other relatively complicated types, without great difficulty. These aspects often go un-implemented in other foreign-language interfaces.

In the next chapter we scale up the foundations laid in this chapter and take a look at integrating Haskell into the object-oriented component framework COM.

Chapter 3

COM components in Haskell

This chapter is closely based on the following article, written together with Sigbjorn Finne, Erik Meijer, and Simon Peyton Jones.

> *Calling Hell from Heaven and Heaven from Hell*. In Proceedings of ICFP'99, Paris, France, 1999. Also appeared in ACM SIGPLAN Notices 34, 9, (Sep. 1999). (Finne *et al.*, 1999)

3.1 Introduction

"The physical realization of a functional component is not, in some sense, its essence. Rather, what makes a functional component the type it is, is characterized in terms of its role in relating inputs to outputs and its relations to other functional components."[1]

Component programming is an approach to software construction in which a program is an assembly of software components, perhaps written in different languages, glued together by some common substrate (Szyperski, 1998). The most widely used substrates are Microsoft's Component Object Model (COM) (Mic, 1992), and the Common Object Request Broker Architecture (CORBA) (Object Management Group, 1993). The language-neutral nature of these architectures offers a tremendous new opportunity to those interested in exotic languages such as Haskell: if we can present our programs in COM or CORBA clothing, then the client programs will neither know nor care that the program is written in Haskell. Our Haskell

[1] Dept. of philosophy, Washington university,
http://artsci.wustl.edu/~philos/MindDict/functionalism.html

programs can thereby inter-operate with a huge variety of other software, and a would-be user of Haskell is not faced with an all-or-nothing choice.

In this chapter, we look at the integration of Haskell with the COM (Mic, 1992; Rogerson, 1997; Brockschmidt, 1995) architecture. COM is a language, machine, and operating system independent architecture designed by Microsoft. It is used extensively on Windows systems, and almost every piece of software contains or can be scripted using a COM interface. For example, almost all aspects of the Internet Explorer can be accessed via COM, including the entire HTML document model.

The main contribution of this chapter is the overall *design* of the Haskell COM binding. More specifically:

- The design is carefully factored, so that it can easily work with a variety of Haskell implementations, including interpreters (the latter is trickier than it may at first appear). Most of the required functionality is encapsulated in our separate H/Direct tool, or in library modules written in Haskell. This "arms-length" design does not come at the price of convenience; it is still easy to create COM components, and to implement a COM component in Haskell. Many other COM interfaces have a tighter, and hence less portable, integration with the compiler (Visual Java, for example).

- The only facility required from the Haskell implementation is a foreign language interface that (a) supports the import and export of Haskell functions, and (b) provides hooks for managing pointers from Haskell to the external world, and back again. The previous chapter described this foreign import/export mechanism.

- Even though COM does not support parametric polymorphism, we show how polymorphism can be used to: encode the single inheritance structure of interface pointers; connect interface pointers with their globally-unique identifiers (GUIDs); and ensure that object vector tables are only paired with appropriate object states (Section 3.4).

- COM is very general, but it requires quite a bit of C++ code to build a COM object, usually supported by "wizards" of some sort. We are instead able to provide a library of higher-order functions that make it easy to construct COM objects without wizardly support (Section 3.6.3).

Overall, we give an elegant and easy-to-use design for building and using COM objects in Haskell. Even though this chapter has its focus on COM, many ideas can readily be used to interface with other object oriented libraries, as has been done for example for the wxWindows library (Smart, 1992). Except for the typed layer, there is nothing intrinsically hard about it, but it has nevertheless taken over two years to evolve, so it is certainly a more subtle task than initially foreseen.

3.2 Components

A software component is a reusable piece of software with a well defined functionality and interface. The traditional approach to implementing software components is a library implementing an abstract data type. This leads to a tight coupling between client and component – function calls are resolved at link time and each update in the component forces a relink of the entire program.

Dynamic link libraries[2] (DLL's) bind function calls at run time instead of link time. With a DLL, it is possible to update a component without updating the client code. An example is the Windows operating system whose API, implemented as a DLL, has been updated many times while staying compatible with the earliest versions.

Still, a DLL has many associated problems:

- DLL's are identified at run time with their file name. Besides portability problems, it could give rise to ambiguities when different vendors ship DLL's with the same name.
- DLL's have to be called in the same process context as the client. To call functions in other processes, or even on remote machines, requires the call and arguments to be explicitly transferred across process boundaries.
- There is no robust way of handling failures or sharing of DLL's across processes. For example, a DLL can not know whether it is still in use or if it is safe to unload itself.

3.3 COM

The COM framework tries to solve the above problems. The naming problem is solved by associating a globally unique class identifier (CLSID) with each component. Clients identify their component with these CLSID's and COM takes care of the association between a CLSID and the file that provides the implementation. Equivalently, each set of functions (an interface) is associated with a globally unique interface identifier (IID). Whenever there is an incompatible change in a function, the IID changes and a client can thus never connect to an incompatible set of functions. The IID mechanism is essentially a crude 'global' type checker.

The out-of-process calling mechanism is implemented by using indirect calls to methods of the interface. It enables COM to intercept calls across process- or machine boundaries by providing a COM generated stub that marshals the call

[2] On Unix, these are normally called *shared object files*.

and arguments in an invisible way. A client uses all components as if they are all in-process.

COM introduces reference counting to handle the problem of resource allocation in the presence of failure and sharing. The COM specification (Mic, 1992) describes precisely how COM components should behave with respect to resource allocation and reference counting.

In the following sections, we explain the core technologies of COM. It is beyond the scope of this chapter to explain *why* COM is designed this way but there are many books written about this subject (Mic, 1992; Box, 1997; Rogerson, 1997).

3.3.1 Creating a COM component

The procedure `CoCreateInstance` creates a fresh COM component:

```
typedef long HRESULT;

HRESULT CoCreateInstance( [in] CLSID* clsid
                        , [in] IUnknown* outer
                        , [in] CLSCTX context
                        , [in] IID* iid
                        , [out] void** iface );
```

The `HRESULT` value is the standard way in COM to return a success or failure (Mic, 1992). The `clsid` argument specifies the class that implements our requested interface. The CLSID is a globally unique identifier or GUID. Here "globally unique" means that the GUID will not (ever) be re-used for any other purpose anywhere on the planet. A standard utility generates locally an unlimited supply of fresh GUID's based on the machine's IP address, date and time.

The code for the class is found indirectly via a global database[3]. This indirection makes the client code independent of the actual machine configuration and specific location of the code for the class. Furthermore, COM uses the registry to determine whether the component should be loaded in-process, or in a new local or remote process.

The `outer` parameter is used for aggregating components (Brockschmidt, 1995) while the `context` parameter is used to configure remote components. Both parameters are beyond the scope of this chapter though. The next two parameters determine the returned interface.

[3]The *registry* on windows.

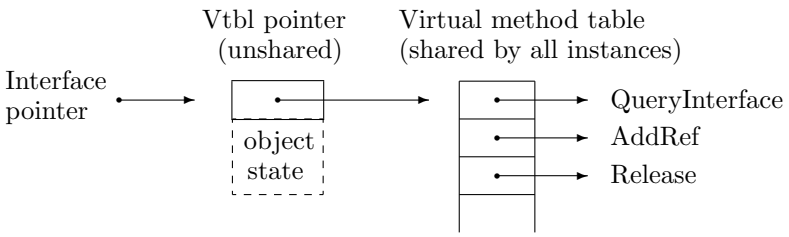

Figure 3.1: Interface pointers

3.3.2 Interfaces

A COM object supports one or more *interfaces*, each of which has its own globally-unique interface identifier or IID. The `iid` parameter in `CoCreateInstance` specifies the IID of the initial interface that is returned to the `iface` parameter. The `iface` parameter receives an *interface* pointer. In COM there is no "object pointer", or "object identifier" – objects are only accessed via their interface pointers.

The IID of an interface uniquely identifies the complete *signature* of that interface; that is, what methods the interface has (including what order they appear in), their calling convention, what arguments they take, and what results they return. If we want to change the signature of an interface, we must give the new interface a different IID from the old one. That way, when a client asks for an interface with a particular IID, it knows exactly what that interface provides.

An interface pointer is a pointer to a pointer to a so-called *virtual method table* (VTBL) pointer (Figure 3.1). A virtual method table is an array of function pointers. The double indirection allows the table of function pointers to be shared among all object instances. Instance specific data of a component can be stored at some fixed offset from the virtual method table pointer. The format of this data is entirely up to the component implementation, the client knows nothing about it. When a function is called via a pointer in the virtual method table, the interface pointer must be its first argument. This allows the component to access its instance specific state.

For example, suppose we want to call the third method from some interface pointer `iface`. Here is some (untyped) C code to do that:

```
(*iface)[2]( iface, ... );
```

None of this is language specific. That is, COM is a binary interface standard. Provided the code that creates an object instance returns an interface pointer that points to the structures just described, the client will be happy. In theory, the pa-

rameter passing conventions for each method can be different (but fixed in advance). In practice, they match the `stdcall` convention used by C and C++.

Interface pointers provide the sole way in which one can interact with a COM object. This restriction makes it possible to implement *location transparency* (a major COM war-cry), whereby an object's client interacts with the object in the same way regardless of whether or not the object is in the same address space, or even in the same machine, as the client. All that is necessary is to build a *proxy* interface pointer, that *does* point into the client's address space, but whose methods are stub procedures that marshal the data to and from across the border to the remote object.

3.3.3 Getting other interfaces

A single COM object can support more than one interface. But as we have seen before `CoCreateInstance` returns only one interface pointer. Fortunately, every COM interface is required to support the `QueryInterface` method, which maps an IID to an interface pointer for the requested IID or fails if the object does not support the requested interface. So, from any interface pointer, `iface`, on an object we can get to any other interface pointer, `iface2`, which that object implements. The `QueryInterface` method is always the first pointer in the virtual method table, for example:

```
hresult = (*iface)[0]( iface, iid2, &iface2 );
```

The COM specification requires that `QueryInterface` behaves consistently. The standard `IUnknown` interface on an object is the identity of that object – queries for `IUnknown` from any interface on an object should all return exactly the same interface pointer.

Queries for interfaces on the same object should always fail or always succeed. Thus, the call `(*iface)[0](iface,iid2,&iface2)` should not succeed at one point, but fail at another. Finally, when viewed as a binary relation over interfaces on a component, `QueryInterface` should be an equivalence relation (i.e. reflexive, symmetric and transitive).

3.3.4 Reference counting

Each object keeps a *reference count* of all the interface pointers it has handed out. Beside `QueryInterface`, every COM interface is required to support two more methods, namely `AddRef` and `Release`. When a client discards an interface pointer it should call the `Release` method *via* that interface pointer. Similarly, when it

duplicates an interface pointer it holds, the client should call the `AddRef` method *via* the interface pointer. When an object's reference count drops to zero it can safely unload itself – but it is up to the object, not the client, to cause this to happen. All the client does is make correct calls to `AddRef` and `Release`.

Every COM interface supports the three methods `QueryInterface`, `AddRef`, and `Release`. The three together constitute the `IUnknown` interface, from which every other COM interface is derived.

3.3.5 Describing interfaces

Since every IID uniquely identifies the signature of the interface, it is useful to have a common language in which to describe that signature. Just as H/Direct and CORBA, COM uses the IDL (Interface Definition Language) to describe its interfaces, but IDL is not part of the core COM standard. You do not have to describe an interface using IDL, you can describe it in classical Greek prose if you like. COM only requires that one IID must identify one signature and that you use the prescribed binary layout to implement the component.

Describing an interface in IDL is useful, though, because it is a language that all COM programmers understand. Furthermore, there are tools that read IDL descriptions and produce language-specific declarations and glue code. For example, the Microsoft MIDL compiler can read IDL and produce C++ class declarations that make COM objects look exactly like C++ objects (or Java, or Visual Basic objects). Furthermore, our H/Direct tool can read IDL and generate Haskell COM wrappers automatically, as described in the next sections.

As an example, here is the IDL code for the `IUnknown` object and the DirectX media control. Direct/X is a COM framework for graphics and sound that is used in games and other demanding graphics applications. The media control is an interface that can be used to play some sound. The `FilGraphManager` is a specific component that implements this interface. The IDL reads:

```
typedef unsigned long ULONG;

[uuid(00000000-0000-0000-C000-000000000046)]
interface IUnknown {
  HRESULT QueryInterface( [in] IID* iid, [out] void** iface );
  ULONG   AddRef(void);
  ULONG   Release(void);
};

[uuid(56A868B1-0AD4011CE0Bo34-0020AF0BA770)]
interface IMediaControl : IUnknown {
  HRESULT Run();
```

```
  HRESULT RenderFile( [in] BSTR strFileName );
  ...
};

[uuid(E436EBB3-524F-11CE-9F53-0020AF0BA770)]
coclass FilGraphManager {
  interface IMediaControl;
  interface IMediaEvent;
  ...
};
```

3.3.6 Example in C++ and Haskell

Using the above components, we can write a small example in both Haskell and C++ to get a feel for how COM components are used. For the C++ example, we first use the Microsoft MIDL compiler to translate the IDL specification into a C++ header file that is used to access the components.

```
#include <objbase.h>
#include <stdio.h>
#include <conio.h>
#include "media.h"    //the MIDL generated header file.

int main() {
  IMediaControl* media = NULL;
  BSTR           str   = NULL;
  HRESULT        hr    = S_OK;

  hr = CoInitialize(NULL)             //initialize COM
  if (SUCCEEDED(hr)) {
    hr = CoCreateInstance(            //create instance
           CLSID_FilGraphManager,
           NULL,
           CLSCTX_INPROC_SERVER,
           IID_IMediaControl,
           (void**)&media
         );
    if (SUCCEEDED(hr)) {
      str = SysAllocString( "media.wav" );
      if (str!=NULL) {
        hr = media->RenderFile(str);    //load the file
        if (SUCCEEDED(hr)) {
          hr = media->Run();            //play the sound
          printf("press a key to quit...");
          getch();
        }
        SysFreeString(str);             //release string
      }
```

```
      media->Release();                 //release interface
    }
    CoUninitialize();                   //uninitialize
  }
  return (hr==S_OK ? 0 : 1);
}
```

COM maps nicely to C++ since the method invocation mechanism of COM is exactly the same for C++ virtual methods and the (`->`) operator can directly be used to call COM methods. However, at the same time many low level details are exposed to the C++ programmer, like error handling and resource allocation for strings and objects. The same example is already simpler in Java and in Haskell even more.

Since all interfaces are described in IDL, we have extended the H/Direct compiler with extra rules to generate marshaling code for COM components. H/Direct generates a Haskell module `Media` that exports the needed functionality.

```
module Main where
import Com                                  -- basic COM functions
import Media                                -- H/Direct generated

main
  = coRun $                                 -- does Co(Un)initialize
    do media <- coCreateInstance            -- create an instance
                  clsidFilGraphManager
                  iidIMediaControl
       media # renderFile "media.wav"
       media # run
       putStr "press a key to quit.."
       getChar
```

The `coRun` function initializes and uninitializes the COM library. The call to `coCreateInstance` always creates an in-process component without aggregation. Since H/Direct generates all the marshaling code according to the rules in the previous chapter, the `"media.wav"` string is automatically converted to string of type `BSTR`. Interfaces are automatically finalized with a call to `Release` when they are garbage collected. Furthermore, H/Direct automatically generates code that checks the returned `HRESULT` to throw a standard `IO` exception on errors.

With all this support, the Haskell example is a lot more concise than the equivalent example in C++. By careful design, all the support code is written in Haskell itself and is therefore completely customizable and extensible – in contrast to Visual Basic or Java where the support is built into the language itself. We will show in chapter 4 that this is a great advantage when programming more complicated examples.

3.4 Polymorphism expresses inheritance

A problem we run into when trying to use COM components (or any other object oriented framework) from Haskell, are inheritance relations. For example, suppose we have a component interface `IWindow` with a method, say, `GetSize`. The naive Haskell types would be:

```
createWindow  :: IO IWindow
windowGetSize :: IWindow -> IO Size
```

Now suppose that the `IFrame` component extends the `IWindow` interface with an extra method `GetTitle`:

```
createFrame   :: IO IFrame
frameGetTitle :: IFrame -> IO String
```

Since `IFrame` extends the `IWindow` interface, we would expect that we can call `GetSize` on a frame too. However, as it stands, the Haskell type system will not allow that! We need some way to model the interface inheritance relationship in Haskell. Our first design to model inheritance used a complex system of type classes. That didn't work well however due to hard to understand type errors and the need to generate instance specific `QueryInterface` functions. Fortunately, we discovered a particularly simple model based on plain parametric polymorphism.

Each interface is associated with an *interface type*. Strangely, such an interface type is an abstract data type with no operations, nor do we ever create a value of the type. This is why we call this a *phantom type*. There are many more compelling uses of phantom types, as we will see in chapter 5 in the context of relational databases. For our example, this looks like:

```
data IWindowT a   = IWindowT
data IFrameT a    = IFrameT
```

The interface type for `IWindow` is declared as `IWindowT`. Since we will never construct a value of this type, it would have been better to just use a type declaration without a definition but currently, Haskell doesn't allow that. With each interface type, we also define a type synonym for pointers to such an interface:

```
type IWindow a   = IUnknown (IWindowT a)
type IFrame a    = IWindow (IFrameT a)
```

Note how we use the type parameters to encode the inheritance relationship. We see that an `IWindow` extends the `IUnknown` interface, while an `IFrame` extends an `IWindow`. The types of their methods now become:

```
windowGetSize :: IWindow a -> IO Size
frameGetTitle :: IFrame a -> IO String
```

Now, the good thing is that whenever we have an interface pointer `IFrame a`, we can use it to call `windowGetSize`. Why? Because:

```
IFrame a == IWindow (IFrameT a)
```

(just by expanding the type synonym for `IFrame`). That is, *every interface pointer for an* `IFrame` *is automatically an interface pointer for* `IWindow`*, and indeed also an interface pointer for* `IUnknown`.

Now, we can also understand the types for the creation functions:

```
createWindow  :: IO (IWindow ())
createFrame   :: IO (IFrame ())
```

The function `createWindow` creates an `IWindow` interface *exactly*, expressed by instantiating the type parameter to `()`. This basically enforces the covariant and contra-variant constraints. For example, it is not possible to call `frameGetTitle` with a plain window of type `IWindow ()`.

In short, we have been able to use simple polymorphic instantiation to model single inheritance (which is precisely what COM requires).

The polymorphic model of inheritance also carries over nicely to the COM interface identification and creation. Since each IID uniquely determines an interface, we parameterize each IID with its corresponding *interface type*. For example:

```
iidIWindow :: IID (IWindow ())
iidIFrame  :: IID (IFrame ())
```

Now, we can strongly type the interface creation functions with an IID:

```
coCreateInstance :: CLSID     -> IID iface   -> IO iface
queryInterface   :: IID iface -> IUnknown a  -> IO iface
```

Note how `queryInterface` can be called on any component that supports the `IUnknown` interface. The polymorphism in `coCreateInstance` and `queryInterface` elegantly ensures that the interface pointer returned is statically checked to support the same methods as the IID that was passed.

Another thing to note is that the interface pointer is the last argument to a method like `queryInterface`. In most object oriented languages this pointer is actually

passed (implicitly) as the *first* argument to a method. For Haskell though, the last argument is more appropriate as we are able to define a polymorphic application operator `(#)` as the Haskell equivalent of the arrow `(->)` in C++ or the dot `(.)` in Java.

```
infix 5 (#)

(#) :: a -> (a -> b) -> b
object # method   = method object
```

Although a simple function, we have used all kinds of cumbersome methods where the interface pointer was passed as a first argument before discovering this simple trick. Now we can call methods in a very natural way:

```
do window <- coCreateInterface clsidFrame iidIWindow
   frame  <- window # queryInterface iidIFrame
   ...
```

Note that in contrast with conventional object oriented languages, the method, the object and the method invocation operator are all first class values. This makes it easy to define expressive abstractions, which we will discuss further in chapter 4.

3.5 Marshaling COM components to Haskell

Just as with plain IDL functions, we can also give a formal translation from IDL interfaces to Haskell. We have extended the H/Direct compiler with new rules to handle the marshaling of components automatically. We need two extra cases for the declaration translation function $\mathcal{D}[\![\ ]\!]$, one for class declarations (`coclass`) and one for interface declarations (`interface`).

A class declaration is translated into a plain CLSID, the statically supported interfaces are ignored as they are always retrieved dynamically from Haskell.

$\mathcal{D}[\![$`[uuid(`*clsid*`),...] coclass` *name* `{` *interfaces* `}`$]\!]$
$\mapsto$
`clsid`$\mathcal{N}[\![$*name*$]\!]$ `:: CLSID`
`clsid`$\mathcal{N}[\![$*name*$]\!]$ `= makeCLSID "`*clsid*`"`

For an interface declaration, we generate a fresh type for the interface together with a typed IID declaration. The methods are translated separately with the $\mathcal{V}_{iface}[\![\ ]\!]$ translation function.

```
  𝒟⟦[uuid(iid),...] interface iface : extends { ... method; ... }⟧
↦
  type 𝒩⟦iface⟧ a  = 𝒩⟦extends⟧ (𝒩⟦ifaceT⟧ a)
  data 𝒩⟦ifaceT⟧ a = 𝒩⟦ifaceT⟧

  iid𝒩⟦iface⟧ :: IID (𝒩⟦iface⟧ ())
  iid𝒩⟦iface⟧ = unsafeMakeIID "iid"

  ...
  𝒱iface⟦method⟧
  ...
```

The function `unsafeMakeIID` creates a fresh IID from an IID string with an uninstantiated type:

```
data IID a  = IID String    -- hidden constructor

unsafeMakeIID :: String -> IID a
unsafeMakeIID iidString  = IID iidString
```

Since the result type is still polymorphic, the function is in general unsafe as it allows us to attach any interface type to an IID. The use of this function by H/Direct is safe though since it has a type signature that constrains this type to the correct interface type specified by the IDL declaration.

Methods in a class are translated with the $\mathcal{V}_{iface}[\![\ ]\!]$ scheme. Just as in section 2.5 we illustrate the general concept with a specific scheme as the general version has an inconvenient number of subscripts. Here is the scheme for a method that takes a single argument:

```
  𝒱iface⟦tres  method( [in] tin )⟧
↦
  𝒯⟦method⟧ :: 𝒯in⟦tin⟧ -> 𝒯in⟦iface⟧ -> IO 𝒯res⟦tres⟧
  𝒩⟦method⟧ = \in -> \self ->
    do { a <- ℳ⟦tin⟧ in
       ; s <- ℳ⟦iface⟧ self
       ; m <- rdVtblPtr s (offset𝒩⟦method⟧)
       ; r <- prim𝒩⟦method⟧ s a
       ; x <- 𝒰⟦tres⟧ r
       ; hdFree
       ; return x
       }

  offset𝒩⟦method⟧
    = ...
```

```
foreign import stdcall dynamic prim𝒩⟦method⟧
  :: Addr -> ℬ_in⟦iface⟧ -> ℬ_in⟦t_in⟧ -> IO ℬ_res⟦t_res⟧
```

There are several small differences to the $\mathcal{D}[\![\ ]\!]$ scheme presented in section 2.5 for normal procedures:

- A method gets an extra last argument that points to the interface.
- All the type translation schemes are now parameterized by their mode as input type ($\mathcal{T}_{in}[\![\ ]\!]$) or output type ($\mathcal{T}_{res}[\![\ ]\!]$). The type schemes behave the same as the previous $\mathcal{T}[\![\ ]\!]$ scheme except for interface types. As explained earlier, in a covariant position, the class type has a polymorphic type variable while in a result type, the type variable is instantiated to `()`.
- The marshalling scheme for an interface $\mathcal{M}[\![iface]\!]$ extracts the raw interface pointer from the Haskell interface.
- The `rdVtblPtr` function uses the raw interface pointer to read a specific function pointer from the *virtual method table*. The offset is statically determined by the method and given by `offset`$\mathcal{N}[\![method]\!]$. The translation scheme for method offsets is straightforward but beyond the scope of this chapter.
- Instead of normal `foreign import`, we use the dynamic variant (2.3.4) since we call the method via a dynamic virtual method table pointer.

The above extensions to H/Direct are basically all that is needed to call COM components from Haskell. There are a few more rough edges but nothing really difficult:

- As said before, most COM methods return a value of type `HRESULT` to signal errors. H/Direct needs to recognize these to translate these errors into standard Haskell `IO` exceptions.
- There are a few more COM specific attributes, like `is_iid`, that can be handled by H/Direct to generate more specific type signatures or marshaling code.

As described in the previous chapter, it is not hard to reverse the mapping and build code that allows clients to call Haskell components. However, exposing Haskell as COM component requires much more runtime support than just the marshalling of the method calls to Haskell. The encapsulation of Haskell as a COM component is therefore be the main theme for the rest of this chapter.

3.6 Encapsulating Haskell as a COM component

The starting point of encapsulating Haskell as a COM component is an IDL specification for the interface(s) a component must offer; as a running example we use a telephone book component. This example is closely based on the example used in (Szyperski, 1998) to introduce Component Pascal's support for interacting with COM.

```
[uuid(...)]
interface ILookup : IUnknown {
  HRESULT GetPhoneNumber( [in,string] char* name
                        , [string,out] char* number );
};

[uuid(...)]
interface IInsert : IUnknown {
  HRESULT AddPhoneNumber( [in,string] char* name
                       , [in,string] char* number );
};

[uuid(...)]
coclass PhoneBook {
  interface ILookup;
  interface IInsert;
};
```

We tackle the encapsulation in three clearly-separated "layers":

- Code written by the application programmer (section 3.6.1). An implementation is provided and the component is registered with COM.
- Code generated by H/Direct from the `PhoneBook` IDL (section 3.6.2). This boilerplate code deals with marshalling arguments between Haskell and the client; it also deals with creating the component's virtual method tables and interface pointers in exactly the form expected by COM clients.
- Fixed code that lives in the `Com` library (Section 3.6.3).

3.6.1 The programmer's eye view

What does the Haskell programmer have to do to implement the `PhoneBook` in Haskell? First H/Direct generates a Haskell module `PhoneBookProxy.hs`. This module imports a Haskell module `PhoneBook.hs`, which provides the programmer's implementation of the `PhoneBook` functionality. H/Direct optionally outputs a skeleton for this module, but the programmer must complete it by providing:

- A type declaration for the state of the `PhoneBook` object. This type is given the same name as the class. For example:

```
type Name      = String
type Number    = String
type PhoneBook = IORef [(Name,Number)]
```

  Here the state `PhoneBook` is held in a mutable `IORef` cell. The advantage of using mutable references over a more "functional" state-transformer style is that under the former scheme, the Haskell signatures for *using* components and for *implementing* components in Haskell are uniform.

- An initializer, `initPhoneBook`, for the `PhoneBook` state. For example:

```
initPhoneBook :: IO PhoneBook
initPhoneBook = newIORef []
```

- An implementation for each method. The Haskell type of each method is derived from the corresponding IDL type as described in the previous section. For example:

```
getPhoneNumber :: Name -> PhoneBook -> IO Number
getPhoneNumber name phonebook
  = do{ pairs <- readIORef phonebook
      ; case (lookup name pairs) of
          Just number -> return number
          Nothing     -> coError E_Fail
      }
```

  When implementing components, the last parameter of each method is the state of the object (instead of the interface pointer). The function `coError` raises an exception in the `IO` monad, passing the `E_Fail` return code, which is automatically marshaled into COM's `E_FAIL` return code by H/Direct in the proxy module.

Finally, the programmer must make the new component known to COM by supplying a main module, `Main.hs`, as follows:

```
module Main where

import Com(coRegisterComponents)
import PhoneBookProxy(phoneBook)

main :: IO ()
main = coRegisterComponents [phoneBook]
```

When the Haskell program is run, the call to `coRegisterComponents` registers the component(s) defined in its argument list. Clients are now able to create instances of the Haskell components. If a single Haskell program implements more than one COM component, `main` would import several `Proxy` modules, and would have several items in the list passed to `coRegisterComponents`.

And that is all the programmer has to do! Next, we look behind the scenes, and study the generated `PhoneBookProxy` module and the library `Com` module.

3.6.2 The generated proxy module

H/Direct generates the Haskell module `PhoneBookProxy.hs` from the IDL specification of the `PhoneBook` component, which exports the single value `phoneBook`. The value `phoneBook` encapsulates the complete implementation of the component[4]

```
phoneBook :: CoComponent

data CoComponent
  = CoComponent
    { componentCLSID  :: CLSID
    , componentNew    :: IO (IUnknown ())
    }
```

A `CoComponent` contains both the CLSID of the component and an instance creation function. The `componentCLSID` is easy to generate from the IDL, and we will focus on the `componentNew` field.

To create an instance of a COM component we need to construct an interface pointer that looks precisely as depicted in Figure 3.1. We represent an interface pointer as a pointer to a `malloc`'d pair of (a) a virtual method table pointer and (b) a stable pointer to the objects state[5]. There are two things we must be able to do:

1. *Create a virtual method table.* In a compiled implementation we could do this statically, but that would rule out interpreters like Hugs, so we provide a function that dynamically builds a virtual method table.

   ```
   type CoVTable iid st = Addr
   ```

[4]The actual data type contains a few extra fields – for example a string giving a short description of the component.

[5]In principle, we could instead create a fresh method table for each instance of the object; the methods could then have the object state as a free variable, just like `getTitle` did in Section 2.3.5. But that would mean much method-table duplication, so instead we follow COM's hint, and use a fixed method table, shared among all instances.

```
newCoVTable :: [Addr] -> IO (CoVTable iid st)
```

The function `newCoVTable` uses `malloc` to allocate a fixed, vector table, returning its address as a Haskell closure.

The `CoVTable` type is parameterized by the interface type (`iid`) of the interface it implements, and object state (`st`) understood by the methods. The method addresses passed to `newCoVTable` point to procedures that can be called directly by other COM objects. These addresses are generated using dynamic `foreign export` declarations. The function `newCoVTable` prefixes this list with three further addresses for the standard `IUnknown` interface methods – `QueryInterface`, `AddRef`, and `Release`. (A variant of `newCoVTable` is provided for those who want to write their own implementations of these methods – see Section 3.6.4.)

2. *Create an instance of the object.* A COM object may support several interfaces, so we must pass a list of (IID,VTable) pairs for every interface the object implements, each of type `Interface`:

```
data Interface st
  = forall iid. Interface (IID iid) (CoVTable iid st)

newCoInstance :: st -> [Interface st] -> IO (IUnknown ())
```

Function `newCoInstance` takes an initial state, a list of interfaces (each specified as a (IID,VTable) pair), and returns a pointer to the `IUnknown` interface of the new instance.

The `data` type declaration for `Interface` uses an existential type `iid`. This extension, first suggested by Laufer (Läufer and Oderski, 1992), is implemented by several Haskell compilers. The data type has one constructor, `Interface`, with type:

```
Interface :: IID iid -> CoVTable iid st -> Interface st
```

The `IID` and `CoVTable` must have compatible `iid` types, but that type does not show up in the type of the constructed value. It expresses elegantly that each interface on the instance has a different virtual method table, i.e. IID, but that all interfaces share a common state `st`. It is interesting that one can use advanced type systems even at the core of a low level COM implementation.

We are finally ready to give the generated code for the `PhoneProxy.hs` module. Remember that its sole export is the component `phoneBook`.

```
module PhoneBookProxy( phoneBook ) where

import PhoneBook( PhoneBook, initPhoneBook
                , getPhoneNumber, addPhoneNumber )
```

```
import Com( CoComponent(..),
            Interface(..),
            newCoInstance, newCoVTable,
            CoIPRep, getCoState )


phoneBook :: CoComponent
phoneBook
  = CoComponent {
      componentCLSID = makeCLSID "...",
      componentNew   = newPhoneBook
    }

newPhoneBook :: IO (IUnknown ())
newPhoneBook
  = do{ init <- initPhoneBook
      ; newCoInstance init phoneBookInterfaces
      }

phoneBookInterfaces :: [Interface PhoneBook]
phoneBookInterfaces
  = [Interface iidILookup vtableILookup
    ,Interface iidIInsert vtableIInsert]

vtableILookup :: CoVTable (ILookup ()) PhoneBook
vtableILookup
  = unsafePerformIO $
    do addr <- wrapGetPhoneNumber primGetPhoneNumber
       newCoVTable [addr]

vtableIInsert :: CoVTable (IInsert ()) PhoneBook
vtableIInsert
  = unsafePerformIO $
    do addr <- wrapAddPhoneNumber primAddPhoneNumber
       newCoVTable [addr]

foreign export dynamic wrapGetPhoneNumber
  :: (CoState PhoneBook -> Addr -> Addr -> IO Int) -> IO Addr

primGetPhoneNumber :: CoState PhoneBook -> Addr -> Addr -> IO Int
primGetPhoneNumber iface p_name p_number
  = do{ st     <- getUserState iface
      ; name   <- unmarshallString p_name
      ; number <- getPhoneNumber name st    -- call haskell implementation
      ; writeString p_number number
      ; return 0                            -- HRESULT
      }
    `catch` err -> coException err

-- Similar wrapper for AddPhoneNumber
```

```
type ILookup a  = IUnknown (ILookupT a)
data ILookupT a = ILookupT

iidILookup :: IID (ILookup ())
iidILookup      = unsafeMakeIID "..."

type IInsert a  = IUnknown (IInsertT a)
data IInsertT a = IInsertT

iidIInsert :: IID (IInsert ())
iidIInsert      = unsafeMakeIID "..."
```

All the above code is automatically generated from the `PhoneBook` IDL by the H/Direct compiler.

The definitions of the compnent information `phoneBook`, the creation function `newPhoneBook` and and the interfaces `phoneBookInterfaces` are straightforward. The virtual method tables, `vtableILookup` and `vtableIInsert` are allocated on demand by using `unsafePerformIO`. Since they are top-level CAF's (see section 6.3.3), we will automatically share the virtual method table across all instances. Furthermore, since the method tables are only allocated on demand, we automatically implement so called *tear-off* method tables, a refined form of *tear-off* interfaces (Box, 1997).

The addresses in the vector table are obtained using a dynamic `foreign export`. The function thus exported is a wrapper function that takes the raw "self" interface pointer as an argument. The purpose of this interface pointer is to get the object state, so we give it the type `CoState PhoneBook`, and provide the operation:

```
getUserState :: CoState st -> IO st
```

which extracts the user state component from an interface pointer. Now we can pass that state on to the user-written method `getPhoneNumber`, imported from module `PhoneBook.hs`. We cannot use the local state directly as we need some extra information to implement the basic `IUnknown` interface (see section 3.6.3).

It may seem strange that in section 3.4 we gave interface pointers a type (`IUnknown`) parameterized by an *interface type*, while here we parameterize a the interface pointer (`CoState`) by the object *state*. How peculiar! However, even though both are *represented* by a single address, they play quite different roles. A value of type `IUnknown iid` is a *client-side* interface pointer for an object held elsewhere; its state is invisible, and when it is finalized (section 2.3.3) we must call its `Release` method. In contrast, a value of type `CoState st` is a server-side interface pointer; its state is visible (because the 'this' pointer is passed to the method implementation), and when there are no further references we need only call `free` to return the store to `malloc`.

3.6.3 The Com library

The bottom layer of the encapsulation is the fixed Haskell library `Com.lhs` that support the COM objects. It is beyond the scope of this chapter to present detailed code – instead we focus on the harder parts of the implementation and summarize how the operations work.

Activation

When a client calls `CoCreateInstance` (see section 3.3.1) to create a COM object, COM looks in the global class database to find which DLL to activate. If the DLL has not already been loaded, COM will load it and invoke its initialization procedure. If the DLL holds a Haskell program, this initialization procedure runs the Haskell program function `main`. As indicated in Section 3.6.1, `main` in turns calls `coRegisterComponents`, passing it a list of all the components that this DLL serves:

```
coRegisterComponents :: [CoComponent] -> IO ()
```

This routine will set a global variable holding a reference to all implemented components. The global variable is implemented using `unsafePerformIO` as described in (Marlow, 2000).

Once `CoCreateInstance` has ensured that the DLL is loaded, it calls a standard entry point `DllGetClassObject`, passing the `CLSID` of the object to be instantiated. This Haskell procedure searches the list of components that is passed to `coRegisterComponents`, looking for one with a matching `CLSID`, and creates an instance of that component. (In reality, it creates a so-called *class factory* object for the object, which in turn can be called to create instances of the object, but the idea stays the same.)

3.6.4 The IUnknown interface

In section 3.6.2 we said that `newCoVTable` and `newCoInstance` worked together to provide implementation of the `IUnknown` methods, `QueryInterface`, `AddRef`, and `Release`. In this section we outline how this is done.

The basic idea is simple enough. Recall that we represent an interface pointer by a `malloc`'d pair of a pointer to the method vector table, and (a stable pointer to) the Haskell state for the object. For COM objects that use the `Com` library support, the Haskell state is a pair of two values: the user state (`PhoneBook` in the above example), and the system state. The system state in turn is a pair of (a) a reference count for the whole object, and (b) a mapping from IID's to interface pointers.

```
type CoState st    = CoState{ userState :: st
                            , sysState  :: CoSysState st
                            }

type CoSysState st = CoSysState{ refCount :: IORef Int
                               , ifaces   :: [(IID (),IUnknown ())]
                               }
```

With this object state in mind, we can provide standard `AddRef` and `Release` methods. They adjust the reference count held in the `CoState`. When the reference count drops to zero, `Release` frees the stable pointer that keeps the object's state alive. That, in turn, may cause a number of finalizers to get called, see section 2.3.3.

The standard `QueryInterface` method uses the interface list to create new interface pointers. The typing of the mapping looks strange, for two reasons. First, the Haskell type system cannot express the idea of a mapping in which the argument *value* determines the result *type*. One needs dependent types for that. Second, the result of `QueryInterface` is in any case returned immediately to the external client, so little is gained by a sophisticated typing.

With this in mind, `newCoVTable` uses the even-more-primitive `newVTable` to do its work:

```
newVTable :: [Addr] -> VTable iid st
-- calls malloc

type    CoVTable iid st = VTable iid (CoState st)
newtype VTable iid st   = VTable Addr
```

The function `newCoVTable` prepends the standard implementations for standard methods `QueryInterface`, `AddRef`, and `Release`, before calling `newVTable`.

Finally `newCoInstance` allocates an interface pointer for each requested interface. The allocation of an interface pointer can actually be done on demand by using `unsafePerformIO` – laziness ensures that we will only allocate it if demanded and than only once. This gives the behavior of *tear-off* interfaces for free (Box, 1997).

3.7 Related work

Haskell is not the only advanced programming language to provide a mapping to COM. The Harlequin Dylan system (Gray *et al.*, 1998) provides a well-engineered COM component framework for Dylan, letting the programmer both create and use COM components. Equipped with such powers, Harlequin Dylan also provides a

framework for writing ActiveX controls, something we have yet to tackle. Component Pascal (Gruntz, 1998) and Microsoft's implementation of Java (Mic, 1998) are two other examples of garbage collected languages which have been integrated with COM.

To our knowing, we have been the first though to implement a full mapping from an imperative object-oriented component architecture, to a pure, non-strict language. Furthermore, we were able to impose a strong type discipline upon the component model. Component integration is becoming more widespread for other functional languages too. Mercury has a CORBA interface (Jeffery *et al.*, 1999) and there is a version of OCaml that supports COM components, using an architecture similar to that described in this paper.

3.8 Conclusions

The main goal of this chapter has been to give the details of calling and building COM components in Haskell.

It is worth stressing that a programmer need to know little of this. All that the programmer needs to do, is feed the IDL for the component to H/Direct and write the application code. All the details of component construction, reference counting, interface querying, and simple finalization (such as calling `Release` on interface pointers held by the object), are handled automatically.

Behind the scenes, though, there are many details to attend to, and we have not even discussed them all. (For example, we omitted details about object registration and finalization.) Still, we hope to have conveyed the essential ideas.

We have made good use of Haskell's type system to make application code completely type secure, and the H/Direct-generated code largely so and we found some interesting uses of polymorphism in so doing. All that we described is implemented in H/Direct and GHC (though some of the function names may differ).

Chapter 4

Functional programming in an imperative world

Parts of this chapter are based on the following articles:

> Daan Leijen, Erik Meijer, and James Hook. *Haskell as an automation controller*. In 3rd International Summerschool on Advanced Functional Programming, 1608, Braga, Portugal, 1999. Springer Lecture Notes in Computer Science (LNCS). (Leijen *et al.*, 1999)
>
> Erik Meijer, Daan Leijen, and James Hook. *Client-side web scripting with HaskellScript*. In Proceedings of Practical Aspects of Declarative Languages (PADL), 1999. (Meijer *et al.*, 1999)

4.1 Introduction

After reading the previous chapters about the interface between Haskell and the imperative world, the reader may wonder whether the rewards are worth the trouble. One potential problem is that a typical COM component is designed with an imperative model in mind, and imposes an imperative style of programming within the functional host language. As we have seen, this is certainly true on the lowest abstraction layer; for example, every COM method has a monadic `IO` type.

In this chapter we try to show that it is actually possible to build expressive functional combinator libraries on top of the basic imperative interface we have seen so far. The resulting libraries can be seen as an embedded *domain specific language* (DSL) tailored for a certain collection of components. This idea is fairly old and

many examples exist of embedding DSL's as a combinator library (Hudak, 1998; Swierstra *et al.*, 1999), including reactive animations (Elliott and Hudak, 1997), CGI scripting (Meijer, 2000), parsing (Swierstra and Azero Alcocer, 1999; Leijen and Meijer, 2001), pretty printing (Wadler, 1997; Hughes, 1995) and hardware description languages (Bjesse *et al.*, 1998).

The advantages of using a custom DSL for programming components include:

- We can often enforce constraints on the component interface that are normally left implicit. For example, a combinator that ensures that the `close` method is always the last method that is invoked once the `open` method is called.
- Furthermore, it is often possible to use expressive type systems of the host language to statically verify contraints on an interface. An extensive example of this technique is shown in the following chapter where queries to a database server component are statically typed.
- Finally, the designer of the DSL is guided by the clear, equational semantics of the functional language and is less likely to make poor design decisions. Indeed, many COM interfaces expose a rather baroque interface just to support languages with few abstraction mechanisms. Often, these irregularities can be hidden within a carefully designed combinator.

We can identify at least four essential ingredients to combinator libraries: parametric polymorphism, higher-order types, laziness and (first-class) computational values. We have already seen how polymorphism is used to capture the notion of interface inheritance. Higher-order types allows us to treat methods as first-class values storing them into lists for example. Laziness is essential to capture orthogonal features in seperate combinators (Hughes, 1989) and to perform computations on demand (as shown in the next chapter when iterating through the results of a database query). Finally, we rely on monads to control side-effects by representing them as functional values. In particular, values of type `IO`, and thus COM method calls, are treated as first-class computational values and can scheduled in arbitrary ways by the high level combinators.

Note that not all functional languages support all these features. In particular, the ML family of languages are eager with implicit side effects, and computational values must be simulated. This reduces our capability to define combinator libraries with a high level of abstraction and may indeed force an imperative model upon the programmer.

In the rest of this chapter, we give an example of how to build a simple combinator library on top of the basic COM interface we have seen so far. As a running example, we use the Microsoft Agent component. This component creates cartoon characters on the screen (see figure 4.1) which can talk, move around and react on speech input. Microsoft Agent is freely available and there is an excellent book about

Figure 4.1: Some agent characters.

programming the agents (Microsoft, 1998). Already, many companies are using the agent component to guide users through web sites or as a personal assistant in an application.

4.2 Running 'hello world'

The agent component is freely available from the Microsoft web site[1]. Of course, H/Direct is used to automatically generate the Haskell stub file `agent.hs` that interfaces to the component from Haskell.

The agent component is actually a server that coordinates the agent character components. The server can be asked to instantiate a character and to return a unique identifier to this character. With this identifier, the server can return a COM interface pointer to the instantiated agent character. The next listing shows a simple agent component program:

```
module Main where

import Com            -- basic support
import Agent          -- the H/Direct generated stub module

main
  = coRun $
    do server     <- coCreateInstance clsidAgentServer iidIAgent
       (charID,_) <- server # load "genie"
       genie      <- server # getCharacter charID
       genie      <- genie  # query iidIAgentCharacter
       genie # showUp 0
       genie # speak "Hello, COM World" ""
```

[1] `http://www.microsoft.com/oledev/agent`

```
putStr "press enter.."
getChar
```

4.3 Abstraction

The previous example does not differ much from an equivalent program in C++, Java or VB. The following sections try to show how the combination of higher-order functions and parametric polymorphism makes it possible to abstract over many commonly occurring code patterns.

4.3.1 Extending the characters' repertoire

The methods `play` and `speak` are rather limited. We would like to be able to define a new, compound method, so that

```
robby # dancesAndSings
```

would make `robby` execute a sequence of `play` and `speak` actions. Here's how we can do that in Haskell:

```
type Action = IAgentCharacter () -> IO ReqId

dancesAndSings :: Action
dancesAndSings agent
  = do{ agent # speak "La la la"
      ; agent # play "Dance"
      }
```

Here we have defined the type `Action` as a shorthand to denote actions that can be performed by an agent (like `play "Dance"` or `dancesAndSings`).

In C++ or Java one could define `dancesAndSings` as the method of a class that inherits from `IAgentCharacter`, using implementation inheritance to arrange to call the character's own `play` and `speak` procedure. To us, it seems rather unnatural to introduce a type distinction between agents that can `dance` and `sing` and agents that can `danceAndSing`. Object oriented languages are good in expressing new objects as extensions of existing objects, while functional languages are good in expressing new functions in terms of existing functions.

In Visual Basic we could certainly define a procedure like `dancesAndSings`, but in that case, we can only call it using a different syntax than native methods calls:

```
Sub DancesAndSings (Byref Agent)
  Agent.Speak ("La la la")
  Agent.Play  ("Dance")
End Sub
...
Robby.Speak ("Hello")
DancesAndSings (Robby)
...
```

If the sequence of actions a particular agent has to perform gets long, it becomes a bit tiresome writing all the "`agent #`" parts, so we can rewrite the definition as a little script, like this:

```
dancesAndSings :: Action
dancesAndSings agent
  = agent # sequence [ speak "La la la", play "Dance" ]
```

where `sequence` is a re-usable function that executes a list of actions from left to right:

```
sequence :: [Action] -> Action
sequence [a]    agent = agent # a
sequence (a:as) agent = do{ agent # a; sequence as agent }
```

Notice that the type of the first argument of `sequence` is a *list* of *functions* that return *I/O performing computations*. The ability to treat functions and computations as first-class values, and to be able to build and decompose lists easily, has a real payoff. In Java, C++, or VB it is much harder to define custom control structures such as `sequence`. For example in Java 1.1 one would use the package `java.lang.reflect` to reify classes and methods into first class values, or use the Command pattern (Gamma *et al.*, 1995) to implement a command interpreter on top of the underlying language. Note that in our case `sequence [...]` is just another composite method on agents, and is called exactly the same way as a native method.

The low cost of abstraction in Haskell is even more apparent when we define a family of higher-order functions to ease moving agents around the screen. First we define a function `movePath` as:

```
type Pos  = (Int,Int)

movePath :: [Pos] -> Action
movePath path agent
  = agent # sequence (map moveTo path)
```

The expression (`movePath path robby`) moves agent `robby` along all the points in the list `path`. In Visual Basic (or Java) we can define a similar function quite easily as well by using the built-in `For ... Each ...Next` control structure:

```
Sub MovePath (Byref AgentCharacter, Byref Path)
  For Each Point In Path
     Agent.MoveTo (Point)
  Next point
End Sub
```

However, in Haskell we don't have to rely on foresight of the language designers to built in every control structure we might ever need in advance, since we can define our own custom control structures on demand. Lazy evaluation and higher order functions are essential for this kind of extensibility (Hughes, 1989).

We can use function `movePath` to construct functions that move an agent along more specific figures, such as squares and circles:

```
moveSquare :: Pos -> Int -> Action
moveSquare (x,y) width agent
 = agent # movePath square
 where
   w = width `div` 2
   square = [ (x-w,y-w), (x+w,y-w)
            , (x+w,y+w), (x-w,y+w)
            , (x-w,y-w)
            ]

moveCircle :: Pos -> Int -> Action
moveCircle (x,y) radius agent
 = agent # movePath circle
 where
  circle = [ ( x + (radius*cos t), y + (radius*sin t))
           | t <- [0,pi/100..2*pi]
           ]
```

Because Haskell uses lazy evaluation, the lists of points are generated on demand and therefore never completely in memory. Having this amount of re-use seems much harder in VB, C++ or Java. When we would define a function like `moveSquare` or `moveCircle` we would have to use a for-each loop each time.

4.4 Agent combinators

The agent server manages each character as a separate, sequential process, running concurrently with the other characters. The following example shows how two characters concurrently sing and dance:

```
do erik  # sings
   simon # dances
```

It looks as if these actions take place sequentially, but they are actually performed in parallel. Each character maintains a queue of requests that it got from the server and performs these in sequence. Hence a call such as `erik # sings` returns immediately, while `erik` starts singing. The same happens for the call `simon # dances`, and they will perform their respective actions in parallel.

The parallelism between two agents `a` and `b`, that perform the actions `A` and `B` respectively, is characterised by the *action-swap* law:

(action-swap)
```
x <- a#A; y <- b#B
≡  y <- b#B; x <- a#A                    { if a ≠ b }
```

Of course, we also require that both `x` and `y` do not occur as free variables, i.e $\texttt{x} \notin fv(\texttt{B})$ and $\texttt{y} \notin fv(\texttt{A})$. From now, this propery is implicitly assumed.

Now suppose we want `daan` to do something when both `erik` and `simon` have terminated; how can we ask the Agent server to do that? The answer is that every `Action` returns a *request-id*, of type `ReqId`, on which any character can `wait`, to synchronize on the completion of that request. Thus:

```
do erikDone  <- erik # sings
   simonDone <- simon # dances
   daan # wait erikDone
   daan # wait simonDone
   daan # speak "They're both done"
```

The `wait` method also has some associated laws. For example, waiting again for some request-id has no effect:

(wait-again)
```
x <- a#wait r; ...; y <- a#wait r
≡  x <- a#wait r; ...; let y = x
```

Furthermore, it doesn't matter in what order one waits for request-id's:

(wait-comm)
```
x <- a#wait r1; y <- a#wait r2
≡  y <- a#wait r2; x <- a#wait r1
```

From the (wait-comm) and (action-swap) law, it follows that we can always wait in any order:

```
(wait-swap)        x <- a#wait r1; y <- b#wait r2
              ≡    y <- b#wait r2; x <- a#wait r1
```

Unfortunately, it is hard to create complex animations with this basic interface, since it hinders abstraction. For example, in the previous animation, `daan` had to wait for all other characters before performing the `speak` action. If you add another participant, you also have add a line where `daan` waits for that participant too. In this small example this may be easy, but when the animation becomes more complex, the amount of hidden dependencies grows fast and managing them becomes error-prone. Matters are further complicated by the fact that the agent server dead-locks whenever an agent tries to wait for itself.

Instead of this low-level interface, we would rather describe the animations *declaratively*. For example, the previous animation could be described as:

```
(action erik sings) <|> (action simon dances)
<*>
(action daan (speak "They're both done"))
```

We have introduced the infix operator (`<*>`) to compose two animations in sequence, and the operator (`<|>`) to compose two animations in parallel. The `action` combinator creates an animation from an agent and an associated method.

Since all the synchronization is now implicit, we can no longer make synchronisation errors. The declarative specification has abstracted away from all the details of the low-level synchronization mechanism between agents. The resulting building blocks can be combined to create complex animations.

We can state various laws about these combinators that we expect to hold. For example, sequential composition should be associative:

```
a1 <*> (a2 <*> a3)  ≡  (a1 <*> a2) <*> a3
```

while parallel composition should both be associative and commutative:

```
a1 <|> (a2 <|> a3)  ≡  (a1 <|> a2) <|> a3
        a1 <|> a2   ≡   a2 <|> a1
```

Since Haskell is a pure and non-strict language, we can use equational reasoning to check our implementation against these expected laws. This means that our proofs

and implementation are written in the same notation, and that the specifications are actually executable programs. As we will see, the ability to treat computations as first class values makes it much easier to reason about side effecting programs.

Proving properties about combinators is not just a technical nicety! As we have already seen, obtaining correct synchronization among the characters is somewhat subtle, and conducting proofs of properties like these can reveal nasty bugs or expose unexpected side effects. In the following sections, we will prove these laws for a particularly simple implementation, and discover that the implementation has some unexpected behaviour.

Before we continue by giving an implementation for the `(<|>)`, `(<*>)`, and `action` combinators, we will in the next two sections develop some machinery that allows us to reason about *unordered* computations. These unordered computations are used to model the implicit parellelism of the agents.

4.4.1 Bags

Parallelism in the agent server is expressed by a sequence of actions whose particular order doesn't matter due to the (action-swap) law. We are going to use *bags* to reason about these unordered actions. Following Bird (1998), we use an abstract representation of bags with the following operations:

```
empty  :: Bag a
member :: a -> Bag a -> Bool
insert :: a -> Bag a -> Bag a
union  :: Bag a -> Bag a -> Bag a
```

For clarity we will write $\varnothing$, $(\in)$, $(\oplus)$, and $(\cup)$ respectively for these operations. We will also use bag comprehension notation ($\{\!|\ |\!\}$).

We can now characterise a bag by the following axioms:

```
x⊕(y⊕xs) = y⊕(x⊕xs)

x ∈ ∅      = False
x ∈ (y⊕xs) = (x = y) ∨ (x ∈ xs)

xs ∪ ∅      = xs
xs ∪ (y⊕ys) = y⊕(xs ∪ ys)
```

The first axiom is crucial for bags: it states that it doesn't matter in what order two elements are inserted into the bag – a bag is unordered. A possible implementation

for these bags would be simple lists as described by (Bird, 1998). Note that it is not possible to specify a function that returns an arbitrary element of a bag. For example, the definition:

```
choose (x⊕xs) = x
```

would lead to inconsistencies, as we can use it to prove that any two elements of a bag are the same:

```
    x
≡   choose (x⊕(y⊕xs))          { by definition }
≡   choose (y⊕(x⊕xs))          { axiom }
≡   y
```

We can prove that the usual bag laws, like commutativity for union, hold under the above axioms.

4.4.2 Unordered computations

Bags can be used to reason about unordered, or parallel, computations. The `seq` operation executes a bag of computations in some order:

```
seq ∅      = return ∅
seq (m⊕ms) = do x <- m; xs <- seq ms; return (x⊕xs)
```

Unfortunately, this definition leads to the same kind of inconsistencies as the previous `choose` function. However, we can restore the consistency in this case, if we restrict the use of `seq` to bags where the elements are computations that can be executed in any order! That is, the following commutative *swap* law should hold for all possible elements:

```
(swap)      do x <- m; y <- n
        ≡   do y <- n; x <- m
```

With this law, we can also prove a commutative law for `seq` itself:

```
(seq-swap)      do x <- seq ms; y <- seq ns
            ≡   do y <- seq ns; x <- seq ms
```

The proof of this case is not entirely straighforward, and we first need to prove the following lemma, where we assume that the (swap) law applies.

```
do x <- m; xs <- seq ms  ≡  do xs <- seq ms; x <- m
```

We prove this by induction over `ms`. It holds trivially for the empty bag. The inductive case is:

```
    do x <- m; xs <- seq (n⊕ms)
≡   do x <- m; y <- n; ys <- seq ms      { unfold, monad laws }
       let xs = (y⊕ys)
≡   do y <- n; x <- m; ys <- seq ms      { (swap) }
       let xs = (y⊕ys)
≡   do y <- n; ys <- seq ms; x <- m      { induction }
       let xs = (y⊕ys)
≡   do xs <- seq ms; x <- m              { fold, monad laws }
```

With this lemma, we can prove the (seq-swap) law by induction over both arguments. It holds trivially when one of the arguments is the empty bag. That leaves only the case of two non-empty arguments:

```
    do xs <- seq (m⊕ms); ys <- seq (n⊕ns)
≡   do x <- m; xx <- seq ms; y <- n; yy <- seq ns
       let xs = (x⊕xx); ys = (y⊕yy)
≡   do x <- m; y <- n; xx <- seq ms; yy <- seq ns     { lemma }
       let xs = (x⊕xx); ys = (y⊕yy)
≡   do y <- n; x <- m; xx <- seq ms; yy <- seq ns     { (swap) }
       let xs = (x⊕xx); ys = (y⊕yy)
≡   do y <- n; x <- m; yy <- seq ns; xx <- seq ms     { induction }
       let xs = (x⊕xx); ys = (y⊕yy)
≡   do y <- n; yy <- seq ns; x <- m; xx <- seq ms     { lemma }
       let xs = (x⊕xx); ys = (y⊕yy)
≡   do ys <- seq (n⊕ns); xs <- seq (m⊕ms)
```

4.4.3 Requests

Now that we can reason about unordered computations, we are able to extend the basic interface of the agent characters. For the implementation of the synchronisation, we will need an operator that returns the *top* of a bag of request-id's. The top is defined as the request-id that is returned from the action that is completed last. Unfortunately, we are unable to define `top` directly since we can not predict beforehand which action takes the longest time to complete.

However, there is no observable difference between waiting for the top of a bag of request-id's and waiting for all those request-id's separately. That is, we can characterise `top` (and `wait'`) as:

```
    a#wait' (top rs)
≡   seq {|a#wait r | r <- rs|}
```

Note that we can use `seq` here due to the (wait-comm) law. From this characterisation, we can derive an implementation of `top` and `wait'`. First, we notice that `wait'` returns a bag of request-id's instead of a single request-id. We take this idea further, and let all methods return a bag of request-id's. The `top` function thus works on a bag of request-id bags, and the characterisation becomes:

```
    a#wait' (top rss)
≡   seq {|a#wait r | rs <- rss, r <- rs|}
≡   seq {|a#wait r | r <- ⋃rss|}
```

With this characterisation, we can simplify and substitute `top` with $\bigcup$ to derive an implementation that satisfies the characterisation automatically.

```
type Request = Bag ReqId

top :: Bag Request -> Request
top rss = ⋃rss

wait' :: Request -> IAgentCharacter a -> IO Request
wait' rs a = seq {|a#wait r | r <- rs|}
```

Furthermore, we define a new operator (`#'`) that returns a singleton bag with a request-id, instead of just a request-id. This is also a good opportunity to tackle another small problem with the agent characters: an agent deadlocks whenever it tries to wait for a request-id that it has generated itself! We can easily circumvent this problem by tupling every request-id with its owner, and checking that no agent tries to wait for itself. Our final implementation becomes:

```
type Request = Bag (IAgentCharacter (),ReqId)

a #' method = do r <- a#method
                 return {|(a,r)|}

top :: Bag Request -> Request
top rss = ⋃rss

wait' :: Request -> IAgentCharacter () -> IO Request
wait' rs a  = seq {|a#wait r | (b,r) <- rs, a ≠ b |}
```

We can prove with (seq-swap), that the (wait-swap) law also holds for `wait'`:

```
(wait'-swap)      do x <- wait' r1; y <- wait' r2
              ≡   do y <- wait' r2; x <- wait' r1
```

The proof for the swap law for (`#'`) is equally simple:

(action'-swap)
```
    do x <- a#'A; y <- b#'B
≡   do y <- b#'B; x <- a#'A
```

Due to the above laws, we can use the new primed operators freely in place of the original operators, and we will write `wait` for `wait'`, and (`#`) for (`#'`) from now on. We no longer need to refer to the primitive versions.

4.4.4 A first implementation

Finally, we have implemented enough machinery to start implementing the actual animation combinators. Deriving an initial implementation is not an option since we only have laws relating operations to themselves instead of other operations. It is beyond the scope of this thesis to formulate a complete semantic model of the agent combinators and to derive an implementation from this model. However, we can formulate a simplified model of our animation combinators and derive an implementation that corresponds to this model. We can than try to prove that this implementation satisfies the basic laws of the animation combinators.

Our simplified interpretation function is written as ($[\![$`anim`$]\!]$ t), and we say that animation `anim` starts after time t and that the animation ends at time ($[\![$`anim`$]\!]$ t). For sequential composition, the start time of the second argument is determined by the end time of the first:

$[\![$`a <*> b`$]\!]$ $t = [\![$`b`$]\!]$ ($[\![$`a`$]\!]$ t)

Both arguments of a parallel composition can start at the same time, and the maximum of their end times is the end time of the animation:

$[\![$`a <|> b`$]\!]$ $t = max($ $[\![$`a`$]\!]$ t, $[\![$`b`$]\!]$ t $)$

We need an auxiliary function to define an interpretation for a basic action. We assume that we have a function $at(t,$`a#A`$)$ that executes action `a#A` at time t and returns the ending time of that action:

$[\![$`action a A`$]\!]$ $t = at(t,$`a#A`$)$

The type of animations will be a function from time values to time values. However, in order to execute actions, we need to lift this function into the `IO` monad. The type of animations thus becomes:

```
type Anim  = Time -> IO Time
```

By lifting function composition too, we can translate the semantics of sequential and parallel composition into an implementation:

```
(a <*> b) t0 = do t1 <- a t0; b t1
(a <|> b) t0 = do t1 <- a t0; t2 <- b t0; return (max t1 t2)
```

Before continuing to basic actions, we need to define the *at* function. Unfortunately, the agent server has no direct concept of time and can only wait for requests. We will therefore use requests as abstract time values and implement the *max* operation with `top`. The *at* operation can be implemented with the `wait` method – if we want to execute a method after a certain time/request, we simply wait for that time/request and then execute the method:

```
type Anim  = Request -> IO Request

(a <*> b) t0     = do t1 <- a t0; b t1
(a <|> b) t0     = do t1 <- a t0; t2 <- b t0; return (top t1 t2)
(action a A) t0 = do a#wait t0; a#A
```

We have now arrived at a particularly simple and concise implementation that closely matches our semantic interpretation. In the next section, we prove the various animation laws and we will see that even this simple implementation is subtly flawed in the sense that it doesn't maximise parallelism. This flaw is in fact so subtle, that this implementation has been used in student projects for years without anyone noticing!

4.4.5 Associativity

We can use equational reasoning to show that seqential and parallel composition are both associative. Here is the proof for parallel composition:

```
    (a1 <|> (a2 <|> a3)) t0
≡   do t1 <- a1 t0                          { unfold }
       tx <- (a2 <|> a3) t0
       return (top t1 tx)
≡   do t1 <- a1 t0                          { unfold, monad laws }
       t2 <- a2 t0
       t3 <- a3 t0
       return(top t1 (top t2 t3))
≡   do t1 <- a1 t0                          { (top-assoc) }
       t2 <- a2 t0
       t3 <- a3 t0
       return (top (top t1 t2) t3)
≡   do tx <- (a1 <|> a2) t0                 { fold,monad laws }
       t3 <- a3 t0
       return (top tx t3)
≡   ((a1 <|> a2) <|> a3) t0                 { fold }
```

The proof of associativity for sequential composition is equally straightforward.

4.4.6 Commutativity

Unfortunately, proving commutativity for parallel composition is far more involved and will reveal some unexpected behaviour. We prove that parallel composition is commutative by induction over the arguments. The basic case is formed by two basic actions. However, we actually have two different cases here, one where the same agent executes the actions, and one where the actions are executed by different agents:

if `a` $\neq$ `b`:

```
    ((Action a A) <|> (Action b B)) t0
≡   do a#wait t0                          { unfold }
       t1 <- a#A
       b#wait t0
       t2 <- b#B
       return (top t1 t2)
≡   do a#wait t0                          { a ≠ b, (action-swap) }
       b#wait t0
       t1 <- a#A
       t2 <- b#B
       return (top t1 t2)
≡   do b#wait t0                          { a ≠ b, 3∗(action-swap) }
       t2 <- b#B
       a#wait t0
       t1 <- a#A
       return (top t1 t2)
≡   do b#wait t0                          { (top-comm) }
       t2 <- b#B
       a#wait t0
       t1 <- a#A
       return (top t2 t1)
≡   ((Action a A) <|> (Action b B)) t0
```

The other case is formed when a single agent executes both actions, `a` = `b`:

```
    ((Action a A) <|> (Action a B)) t0
≡   do a#wait t0                          { unfold }
       t1 <- a#A
       a#wait t0
       t2 <- a#B
       return (top t1 t2)
≡   do a#wait t0                          { (wait-again) }
       t1 <- a#A
       t2 <- a#B
       return (top t1 t2)
≡   do a#wait t0                          { modulo single agent parallelism }
       t2 <- a#B
       t1 <- a#A
       return (top t1 t2)
≡   do a#wait t0                          { (wait-again) }
       t2 <- a#B
       a#wait t0
       t1 <- a#A
       return (top t1 t2)
≡   ((Action a B) <|> (Action a B)) t0
```

This reveals a subtility: when an agent is composed in parallel with itself, it can't do both actions at the same time and has to choose some action to do first. Parallel composition is only parallel up to single agent parallelism. However, this is quite reasonable behaviour and inherent to the agent component. In general, we can formulate an extension to the (action-swap) law that says that we can also swap two single agent actions `A` and `B` as long as neither action is the `wait` method:

(single-swap)
```
    do x <- a#A; y <- a#B
≡   do y <- a#B; x <- a#A
```

Under this new law, we have to synchronise every sequential action explicitly. That is, if we would like to perform two actions sequentially, we write:

```
do x <- a#A; a#wait x; a#B
```

Before we continue with the inductive case of the proof that parallel composition is commutative, we first establish the (anim-swap) lemma:

(anim-swap)
```
    do x <- a#A; y <- anim t₀
≡   do y <- anim t₀; x <- a#A          { a ∉ cast anim }
```

This law states that we can swap an animation and action when they are not related to each other – the agent that executes the action is not part of the *cast* of the animation. The *cast* of animation is defined as:

```
cast (Action a A)  = ⦇a⦈
cast (a₁ <*> a₂)   = cast a₁ ∪ cast a₂
cast (a₁ <|> a₂)   = cast a₁ ∪ cast a₂
```

We prove this law by induction on the animation. The base case consists of a single action (`do x <- a#A; y <- (Action b B) t₀`). First we prove that under our assumption, `a` ≠ `b`:

```
    a ∉ cast (Action b B)
≡   a ∉ ⦇b⦈
≡   a ≠ b
```

The proof is now a straigtforward application of our laws:

```
    do x <- a#A; y <- (Action b B) t0
≡   do x <- a#A                          { unfold }
       b#wait t0
       y <- b#B
≡   do b#wait t0                         { a ≠ b, (action-swap) }
       x <- a#A
       y <- b#B
≡   do b#wait t0                         { a ≠ b, (action swap) }
       y <- b#B
       x <- a#A
≡   do y <- (Action b B) t0              { fold }
       x <- a#A
```

Since the inductive cases are equally structured, we will only prove the (anim-swap) law for parallel composition. First, we establish the assumption:

```
    a ∉ cast (anim1 <|> anim2)
≡   a ∉ cast (anim1 ∪ anim2)             { unfold }
≡   a ∉ cast anim1 ∧ a ∉ cast anim2      { bag laws }
```

The actual proof is straightforward:

```
    do x <- a#A
       y <- (a1 <|> a2) t0
≡   do x  <- a#A                         { unfold }
       t1 <- a1 t0
       t2 <- a2 t0
       let y = (top t1 t2)
≡   do t1 <- a1 t0                       { a ∉ cast a1, induction }
       x  <- a#A
       t2 <- a2 t0
       let y = (top t1 t2)
≡   do t1 <- a1 t0                       { a ∉ cast a2, induction }
       t2 <- a2 t0
       x  <- a#A
       let y = (top t1 t2)
≡   do t1 <- a1 t0                       { (top-comm) }
       t2 <- a2 t0
       x  <- a#A
       let y = (top t2 t1)
≡   do y <- (a1 <|> a2) t0               { fold }
       x <- a#A
```

Finally, we can use our (anim-swap) lemma, to prove an inductive case of commutativity of parallel composition.

```
      (anim <|> (Action a A)) t0
 ≡   do t1 <- anim t0                    { unfold }
        a#wait t0
        t2 <- a#A
        return (top t1 t2)
 ≡   do a#wait t0                        { a ∉ cast anim, (anim-swap) }
        t2 <- a#A
        t1 <- anim t0
        return (top t1 t2)
 ≡   do a#wait t0                        { (top-comm) }
        t2 <- a#A
        t1 <- anim t0
        return (top t2 t1)
 ≡    ((Action a A) <|> anim) t0         { fold }
```

Unfortunately, this only proves that parallel composition is commutative when the action is unrelated to the animation (`a` ∉ `cast anim`). In contrast to the single agent parallelism, we can no longer dismiss this as being reasonable behaviour. Take for example the following animation:

```
(action a A <*> action b B) <|> (action b C)
```

We unfold this program and compare it to the program where the arguments to (`<|>`) are swapped, we get the following expansions for some initial argument t_0:

```
do a#wait t0
   t1 <- a#A
   b#wait t1
   t2 <- b#B
   b#wait t0
   t3 <- b#C
   return (top t2 t3)
≠
do b#wait t0
   t3 <- b#C
   a#wait t0
   t1 <- a#A
   b#wait t1
   t2 <- b#C
   return (top t3 t2)
```

Close study reveals that we have lost some parallelism. In the first fragment, the agent `b` first waits until agent `a` has completed before executing action `B` and `C`. In the second fragment however, agent `b` immediately executes action `C` in parallel

with action `A` of agent `a`. This 'flaw' is hard to notice in practice since the loss of parallelism is not something obvious.

How could this have surfaced when the implementation was an almost literal translation of our interpretation? The reason is that the *at* operation is not implemented faithfully – the implementation executes the action *after* a certain time, not *at* a certain time. The specification also implicitly assumed that the *at* operation was a pure function that would not interact with other *at* invokations. However, the `wait` methods are not commutative with other actions on the same agent, and therefore, the actual implementation of *at* does interact with other invokations.

Fortunately, this problem can be fixed by adding an extra level of interpretation. Instead of executing the animations directly, we first collect all actions where each action is identified with a unique ending time and tupled with their starting time. This can be interpreted as an explicit temporal dependency graph where each action is executed according to its dependencies. It is beyond the scope of this chapter to fully describe and analyse this, but the implementation is given in the appendix.

4.5 AgentScript

Using the primitive animation combinators that we analysed in the previous section, we have built a library to describe more complex animations. For example, the combinator `seqAnim` combines a list of animations:

```
seqAnim :: [Anim] -> Anim
seqAnim = foldr1 (<*>)
```

Note that the associativity law for `<*>` allows us to define it as:

```
seqAnim = foldl1 (<*>)
```

The `animate` function lifts a list of actions into an animation:

```
animate :: IAgentCharacter a -> [Action] -> Anim
animate a ms
  = seqAnim [action a m | m <- ms]
```

The function `runAnim` runs an animation and `loadCharacter` extends the agentserver to make the loading of agents more easy:

```
runAnim       :: Anim -> IAgent a -> IO ()
loadCharacter :: String -> IAgent a -> IO (IAgentCharacter a)
```

Using this library, complex agent interactions can easily be written:

```
module Demo where

import Com
import Agents

type Actor = [Action] -> Anim

demo :: Actor -> Actor -> Actor -> Anim
demo genie merlin robby
  = seqAnim                   -- top level description of interaction
      [ genie introduces,
        merlin appears <|> robby appears,
        (merlin sayshello <*> robby sayshello) <|> genie disappears
      ]
  where
    -- desription of primitive actions
    introduces        = [ showUp,
                          speak "Hello, my friends will show up now" ]
    appears           = [ showUp,
                          play "Surprised" ]
    sayshello         = [ speak "Hi there!" ]
    disappears        = [ hide ]


main :: IO ()
main  = coRun $
        do server    <- coCreateInstance clsidAgentServer iidIAgent
           robby     <- server # loadCharacter "robby"
           merlin    <- server # loadCharacter "merlin"
           genie     <- server # loadCharacter "genie"
           server # runAnim
                     (demo (animate genie) (animate merlin) (animate robby))
```

The resulting library enables a whole new style of programming the agents. The combinators can be viewed as a domain specific language where the interaction between agents (`(<*>)`, `(<|>)`) can be described separately from the actual primitive actions performed by the agents (`introduces`, `appears`)

4.6 Summary

We have shown in this chapter that it is possible to build an expressive functional combinator library on top of a raw imperative component interface. Essential ingredients for this approach are parametric polymorphism, higher-order types, laziness and first-class computations (that is, values of type `IO` τ)

With the agent component, we can define new methods that enforce constraints on the component interface. For example, the `wait'` method ensures that an agent never waits for itself. Furthermore, we have defined a new custom designed sublanguage (or combinator library) for expressing parallel animations. The implementation of the combinators is consise and terse enough, that we are able to perform simple algebraic proofs and derivations of their properties.

Other programming languages can also be used to access the agent component – mainstream scripting languages such as Tcl and Python provide ways to interface to COM components, and of course one can script components in Java, Visual Basic, or C++. The claim of this chapter is that a higher-order, typed, garbage-collected language such as Haskell can open up new avenues for scripting: it is unlikely that one could come up with the described combinator library in any of the above languages.

The next chapter investigates these possibilities in the context of larger component frameworks of database servers. In that chapter we also use the expressive type system of the host language to statically verify implicit constraints on the database components.

4.7 Appendix: animation implementation

The full animation implementation builds an explicit temporal dependency graph of all actions. Each node is identified by a unique ending time. A node contains the times after which the action can execute (the dependencies), the agent and the actual action:

```
type Graph = Set (Time,Times,IAgentCharacter (),Action)
type Times = Set Time
type Time  = Unique

split :: Unique -> Set Unique
seed  :: Unique
```

Since the times are only used to record the dependencies, the actual values don't matter at all and we can use unique values to represent them. The implementation of the combinators is closely modelled after the semantic specification but is obscured somewhat by the need to pass around the unique time supply:

```
type Anim :: Time -> Times -> (Times,Graph)

(action a A) u0 ts0 = ({u0},{(u0,ts0,a,A)})
```

```
(a <*> b) u₀ ts₀    = let {u₁,u₂} = split u₀
                          (ts₁,g₁) = a u₁ ts₀
                          (ts₂,g₂) = b u₂ ts₁
                      in (ts₂, g₁ ∪ g₂)
(a <|> b) u₀ ts₀    = let {u₁,u₂} = split u₀
                          (ts₁,g₁) = a u₁ ts₀
                          (ts₂,g₂) = b u₂ ts₀
                      in (ts₁ ∪ ts₂, g₁ ∪ g₂)
```

The actual interpretation function first builds the dependency graph which is subsequently executed:

```
interp anim
  = exec ∅ ∅ (snd (anim seed ∅))
```

The execution function interprets the dependency graph and executes actions whose dependencies are satisfied. The function takes three arguments: a mapping from abstract time values to requests (for already executed actions), a set of time values that have passed (i.e. whose actions have been executed), and the dependency graph.

```
exec rs₀ ts₀ g
  | next==∅  = return ()
  | otherwise = do seq {|waitfor a ts rs₀ | (t,ts,a,A)<-next|}
                   rs₁ <- seq {|at a A t | (t,ts,a,A)<-next|}
                   exec (rs₀ ∪ rs₁) (ts₀ ∪ {t | (t,ts,a,A)<-next}) g
  where
    next = {(t,ts,a,A) | (t,ts,a,A)<-g, ts ⊆ ts₀, t ∉ ts₀}

waitfor a ts rs
  = a#wait {|r₁ | t₀<-ts, (t₁,r₁ )<-rs, t₀==t₁|}

at a A t
  = do r <- a#A; return (t,r)
```

To gain more trust in the execution function, one can prove all kinds of interesting properties. For example, that all actions are executed, that all actions are executed at most once, and that we always make progress.

Chapter 5

Database queries

This chapter is based on the following article:

> Daan Leijen and Erik Meijer. *Domain specific embedded compilers*. In Second USENIX Conference on Domain Specific Languages (DSL'99), pages 109–122, Austin, Texas, October 1999. USENIX Association. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000). (Leijen and Meijer, 1999)

5.1 Introduction

This chapter provides a comprehensive example of how to embed a domain specific language (Hudak, 1998) (DSEL) in a higher-order, strongly typed, and lazy language like Haskell. Our specific domain instance is the embedding of a database queries inside Haskell but we hope to expose the general design pattern underlying this example. H/Direct is used to create the low-level binding to the database COM components of Microsoft SQL server.

Databases are ubiquitous in the computer industry. For instance, a web site is usually nothing more that a fancy facade around a conventional database. Sometimes, servers are even running directly on a database query engine that generates pages from database records on-the-fly. Hence it is not surprising that database vendors provide hooks that enable client applications to access and manipulate their servers in a convenient way. On UNIX platforms this is usually done via ODBC or vendor specific methods, under Windows their are confusingly many possibilities, including ADO, OLE DB and ODBC.

What is common to all the above database bindings is that queries are communicated as unstructured strings (usually) representing SQL expressions. This low-level approach has many disadvantages.

- Programmers get no (static) safeguards against creating syntactically incorrect or ill-typed queries, which can lead to hard to find runtime errors.
- Programmers have to distinguish between at least two different programming languages, SQL and the host language. This makes programming needlessly complex.
- Programmers are exposed to the accidental complexity and idiosyncrasies of the particular database binding, making code harder to write and less robust against the vendor's fads (Brown *et al.*, 1998)

This chapter not only shows how an easy connection between Haskell and database servers is established with the help of H/Direct but also provides a comprehensive example of how to embed a domain specific language (Hudak, 1998) (DSEL) on top of the raw imperative interface provided by the database components. Although our specific instance is the embedding of a database queries inside Haskell but we hope to expose a general design pattern for embedding domain specific languages.

In general, providing a composable framework for domain specific abstractions is of greater utility than a collection of stand-alone domain specific languages.

- Programmers only have to learn one language – domain specific language extensions are provided as a library.
- It is nearly always possible to guarantee that programmers can only produce syntactically correct target programs, and in many cases we are able to impose domain specific typing rules. Of course, this is all limited by the expressiveness of the host language. In Haskell for example, the value $\perp$ is a value of every type and we can not protect programmers from producing infinite or partially defined values.
- Programmers can seamlessly integrate with other domain specific libraries, for example with CGI and mail protocols. These libraries are accessible in the same way as the original library. This advantage is a largely underestimated benefit of using the embedded approach. Connecting different domain specific languages is one of the reasons for the existence of untyped scripting languages like Perl.
- Programmers can leverage on the existing language infrastructure such as modules, type systems and abstraction mechanisms.

The ideas underlying our thesis date way back to 1966 when Peter Landin (1966) already observed that all programming languages compromise a domain independent

linguistic framework and a domain specific set of components. This chapter is novel in the sense that we show how the terms and type system of a *programming language* are embedded in Haskell, which dynamically *compiles* and *executes* programs written in the embedded language. No changes or extensions were needed to embed the language in Haskell. Because of the compilation step, we call this approach a *domain specific embedded compiler* (DSEC) instead of the normal domain specific embedded language (DSEL).

5.2 A crash course in relational algebra

Before we describe how we can embed SQL queries in a type-safe manner we will first give a crash course in relational databases and how to use them from mainstream languages.

In a relation database (Date, 1995), data is represented as sets of tuples. The fields the tuples are called *attributes*. Take for example the database `Boards`:

`brand`	`model`	`price`	`freeride`
Salomon	Fastback	$ 500	True
Burton	Cascade	$ 600	True
Nitro	Glide	$ 300	False

We can conclude from this table that Burton boards are expensive and that a Glide is not suitable for free riding. The relational algebra allows to query the database in a more systematic way.

The *selection* operator σ specifies the subset of rows of which the attributes satisfy some property. For example, we can select all snow boards that are suitable fro free riding using the following expression: $\sigma_{(\texttt{freeride}=\text{True})}\texttt{Boards}$.

`brand`	`model`	`price`	`freeride`
Salomon	Fastback	$ 500	True
Burton	Cascade	$ 600	True

The *projection* operator π specifies a subset of the columns of the database. Here are all brands that manufacture cheap boards: $\pi_{\texttt{brand}}(\sigma_{(\texttt{price}\leq 500)}\texttt{Boards})$.

`brand`
Salomon
Nitro

An attribute is renamed with the rename operator ρ.
$\pi_{(\texttt{company},\texttt{price})}(\rho_{(\texttt{company}=\texttt{brand})}\texttt{Boards})$.

company	price
Salomon	500
Burton	600
Nitro	300

Other typical operations are *cartesian product* ($\times$), *union* ($\cup$) and *difference* ($-$). All of them pose constraints on the schemes of their arguments. For example, we can only take the cartesian product of relations whose schemes are disjoint. The *rename* operator renames attributes and can be used to ensure that the arguments of a cartesian product have disjoint schemes. Suppose we have a table of sponsored `Riders`:

brand	lastname	firstname
Salomon	Taggart	Michele
Burton	Rippey	Jim
Burton	Haakonson	Terje
Rossignol	Jones	Jeremy

To associate each rider with the range of snow boards they can get for free, we take the cartesian product and rename the common `brand` attribute.
$\pi_{(\texttt{brand},\texttt{model},\texttt{firstname},\texttt{lastname})}(\sigma_{(\texttt{brand}=\texttt{brand'})}(\texttt{Boards} \times \rho_{(\texttt{brand'}=\texttt{brand})}\texttt{Riders}))$.

brand	model	firstname	lastname
Salomon	Fastback	Michele	Taggart
Burton	Cascade	Jim	Rippey
Burton	Cascade	Terje	Haakonson

Since the use of the rename operator is quite cumbersome (just think of taking a cartesian product of a relation on itself) and since the above pattern occurs so often in practice, a special operator is defined that doesn't need the elaborate renaming required for a cartesian product.

The *join* operator ($\bowtie$) takes the cartesian product of two relations but merges tuples whose common attributes have identical values. We can rephrase our previous expression with a join operator as: $\pi_{(\texttt{brand},\texttt{model},\texttt{firstname},\texttt{lastname})}(\texttt{Boards} \bowtie \texttt{Riders})$.

5.2.1 SQL

SQL is the de facto standard programming language for expressing queries on relational databases. The standard form of a SQL query is:

```
SELECT columns
FROM tables
WHERE criteria
```

This combines selections, projections and products in one powerful primitive. The `SELECT` clause specifies which columns to project, the `FROM` clause specifies which tables are combined in a product and the `WHERE` clause specifies which rows in the tables are selected. The query $\sigma_{(\texttt{brand=Burton})}\texttt{Riders}$ is expressed in SQL as:

```
SELECT *
FROM Riders AS r
WHERE r.brand = "Burton"
```

A more complicated query as: $\pi_{(\texttt{brand,model,firstname,lastname})}(\texttt{Boards} \bowtie \texttt{Riders})$, is translated as:

```
SELECT b.brand, b.model, r.firstname, r.lastname
FROM Riders AS r, Boards AS b
WHERE r.brand = b.brand
```

We qualify the relations here in order to disambiguate the `brand` attribute we refer to. Unfortunately, the qualification mechanism of SQL is quite restrictive and a machine translation from relation algebra to SQL is easier with explicit renaming in nested queries:

```
SELECT brand, model, firstname, lastname
FROM (SELECT brand AS brand', firstname, lastname FROM Riders)
    , Boards
WHERE brand' = brand
```

5.2.2 Connecting to the database

We use ActiveX Database Objects (ADO) as our database component. ADO is a COM framework that can use any ODBC compliant database system – Microsoft SQL server, Oracle, IBM DB/2, MS Access and many others (even text files). We have used H/Direct to translate the interfaces of ADO from its type library into Haskell modules. This also provided a good test for H/Direct as ADO is a full scale,

industrial database framework consisting of dozens of interfaces with hundreds of methods. Indeed, entire books are written about ADO alone (Sussman, 2000). In this chapter though, we will focus on just the tiny fraction that is needed to support our examples.

ADO abstracts from physical databases using an `IConnection` object. The `Open` method initialises the connection to a specific database. Once a connection has been established, the `Execute` method evaluates a SQL query in the database context. The result of such a query is returned as a set of records, the `RecordSet` object.

```
type IConnection a
open    :: String -> IConnection a -> IO ()
close   :: IConnection a -> IO ()
execute :: String -> IConnection a -> IO (IRecordset ())
...
```

The `IRecordset` interface exposes methods to navigate through the set of database records.

```
type IRecordset a
moveNext  :: IRecordset a -> IO ()
getEOF    :: IRecordset a -> IO Bool
getFields :: IRecordset a -> IO (IFields ())
...
```

The `Fields` interface is subsequently used to navigate through the fields of single record.

```
type IFields a
getCount  :: IFields a -> IO Int
getItem   :: Variant name => IFields a -> name -> IO (IField ())
...
```

Finally, the `getValue` method of a `Field` object can be used to retrieve the value of a column in the current row.

```
type IField a
getValue :: Variant v => IField a -> IO v
getName  :: IField a -> IO String
...
```

5.2.3 Putting it all together

Visual Basic is often used as the glue language between a database server and a web server. Here is a small example of how ADO is used to print the results of the

query $\sigma_{(\texttt{brand}=\text{Burton})}\texttt{Riders}$.

```
query = "SELECT *"
query = query & "FROM Riders"
query = query & "WHERE brand = 'Burton'"

Set con = CreateObject( "ADODB.Connection" )
con.Open "dsn=mydatabase;server=SQL Server"
Set rs  = con.Execute query

Do While Not rs.EOF
  Print rs.GetFields().GetItem("brand").GetValue()
  Print rs.GetFields().GetItem("price").GetValue()
  rs.MoveNext
Loop

con.Close
```

In Haskell, this example is structured exactly in the same way, but for good measure, we show how we can abstract from the iteration through the record set by returning a list of fields. The list of fields can be returned either strictly, reading all fields into memory at once, or lazily, reading each field by demand.

Both strategies can be defined in terms of a function `readFields` that takes an `IO` action transformer that determines the strategy.

```
readFields :: (IO a -> IO a) -> IRecordset a -> IO [IFields ()]
readFields perform records
  = perform $
    do{ atEof <- records # getEOF
      ; if (atEof)
         then return []
         else do{ fields <- records # getFields
                ; records # moveNext
                ; rest   <- records # readFields perform
                ; return ([fields] ++ rest)
                }
      }
```

By taking `perform` to be the identity, we get a function that reads the complete list of fields strictly. We can also pass the `IO` delaying function `unsafeInterleaveIO` to obtain a function that reads the list of fields lazily.

Here is how a general version of the Visual Basic query evaluator given previously is written in Haskell:

```
runQuery :: String -> IO [IFields ()]
```

```
runQuery sql
  = do{ connection <- coCreateObject "ADODB.Connection"
                                     iidIConnection
      ; connection # open "dsn=mydatabase;server=SQL Server"
      ; records    <- connection # execute sql
      ; fields     <- readFields id records
      ; connection # close
      ; return fields
      }
```

5.3 Query embedding

The examples of the previous section show the essence of database programming nowadays. We can identify at least three weaknesses that can cause a query to fail at runtime:

- A syntactically incorrect SQL query, for example `"SELECCT * FROM Riders"`.
- A semantically incorrect SQL query, for example `"SELECT * FROM Riderss"`.
- A weak connection between the host language and the database, for example `GetItem("price")`, where the `price` attribute is represented as a string instead of an identifier. This item is related to semantically incorrect queries.

All of these items are related to the construction of the SQL query. In this chapter we will describe how to embed database queries in the host language in such a way that the resulting SQL query is *always* semantically and syntactically valid. Moreover, the embedding naturally leads to a strong connection between the host language and the query.

The actual embedding is accomplished via *monad comprehensions*. Within the functional programming community, people have argued before that monad (or *list*) comprehensions are a good query notation for database programming languages (Buneman *et al.*, 1996).

Using monad comprehensions, the query $\sigma_{(\texttt{brand}=\text{Burton})}\texttt{Riders}$ is expressed as:

```
query = do{ r <- table riders
          ; restrict (r!brand .==. constant "Burton")
          ; return r
          }
```

More complicated queries with multiple tables are also possible. Take for example the query: $\pi_{(\texttt{brand},\texttt{model},\texttt{firstname},\texttt{lastname})}(\texttt{Boards} \bowtie \texttt{Riders})$.

```
query = do{ r <- table riders
          ; b <- table boards
          ; restrict (r!brand .==. b!brand)
          ; project (brand = b!brand, model = b!model
                    ,firstname = r!firstname, lastname = r!lastname)
          }
```

Only *sanctioned* combinators like `table` and `restrict` can be used to generate a query. The programmer no longer writes queries as strings, and is even, constrained by the module system, unable to do so! This garantees that only syntactically valid queries are generated.

The combinators are also *typed* at *compile-time*, preventing semantically incorrect SQL queries at *run-time*. Furthermore, the attributes and available tables are all accessed via Haskell identifiers instead of strings, leading to a strong connection between Haskell and the database.

Since the queries are not immediately executed but instead generate a seperate SQL query under the hood (which is subsequently sent to the database server) we call this approach a *domain specific embedded compiler* (DSEC). Indeed, the embedding even contains an optimizer that processes the generated SQL query before sending it to the database server.

Although the library that implements the embedding of database queries in Haskell is large and complex, we have identified three common patterns in the definition of a domain specific language:

1. define the abstract syntax;
2. define an embedding of the abstract syntax;
3. impose a typed layer onto the basic combinators.

The following sections therefore not only describe how an embedding is accomplished for database queries, but also try to give the general design patterns for embedding any domain specific language into a strongly typed, polymorphic, higher-order language.

5.4 Formula embedding

Although our final goal is to embed complete relational queries, we look first at the simpler embedding of the boolean expressions f that occur in restrictions (σ_f). These expressions are normally appended to the `WHERE` clause in the final SQL string. The simplest way to represent expressions is thus as a string:

```
type Expr = String
```

This is essentially the style of programming that is in widespread use in industry – an unstructured SQL string is send to the server and the result is a dynamically typed set of fields. There is no mechanism that prevents the programmer to send invalid strings to the server or to unpack fields at the wrong type, leading to errors at runtime and/or unpredictable behavior of the server.

5.4.1 Step 1: Abstract syntax

To prevent the construction of syntactically incorrect expressions, we define an *abstract syntax* for the terms of the language we are targeting, together with a "code generator" to map abstract syntax trees into the concrete syntax of the input language.

The abstract syntax for SQL restriction expressions defines literal constants, unary operators, aggregate operators, binary operators, and attribute selectors. Attribute selectors are explained in section 5.6.

```
data PrimExpr   = AttrExpr  Attribute
                | BinExpr   BinOp PrimExpr PrimExpr
                | UnExpr    UnOp PrimExpr
                | AggrExpr  AggrOp PrimExpr
                | ConstExpr String
                deriving (Read,Show)

data BinOp      = OpEq | OpLt | OpLtEq | ...
                deriving (Show,Read)

data UnOp       = OpNot | OpAsc | OpDesc | OpIsNull | OpIsNotNull
                deriving (Show,Read)

data AggrOp     = AggrCount | AggrSum | AggrAvg | AggrStdDev | ...
                deriving (Show,Read)
```

The types `BinOp`, `UnOp` and `AggrOp` are just enumerations of the permitted operators in the relational algebra. Enforced by the type system, we can now write expressions that have at least a syntactically correct form (when translated into concrete syntax). For example, we now write `BinExpr OpEq (ConstExpr (show 1)) (ConstExpr (show 3))` instead of `"1 = 3"`.

An abstract syntax tree is translated back into concrete syntax just before passing it to the database server. The "code generator" for our expressions is straightforward: print expressions in their fully parenthesized concrete representation by a simple inductive function:

```
pPrimExpr :: PrimExpr -> String
pPrimExpr expr
  = case expr of
      ConstExpr s     -> s
      UnExpr op x     -> pUnOp op ++ parens x
      BinExpr op x y  -> parens x ++ pBinOp op ++ parens y
      ...
  where
    parens x   = " (" ++ pPrimExpr x ++ ") "

pBinOp :: BinOp -> String
pBinOp op
  = case op of
      OpEq  -> "="
      OpLt  -> "<"
      ...
```

Normally however, this step is more involved. As shown in later sections, the full SQL query embedding even performs optimization and renaming before generating the final query string.

5.4.2 Step 2: Abstract Syntax embedding

Writing expression directly in abstract syntax is quite cumbersome, so we provide combinators to make the programmers life more convenient. Each expression operator is represented in Haskell by the same operator surrounded by dots. Some definitions are:

```
constant :: Show a => a -> PrimExpr
constant x
  = ConstExpr (show x)

(.+.) :: PrimExpr -> PrimExpr -> PrimExpr
(.+.) x y
  = BinExpr OpPlus x y
```

The `constant` function is unsafe since any value that is part of the `Show` class can be used. In the real library we introduce a separate class `ShowConstant` which is only defined on basic database types. Now we are able to write: `constant 1 .==. constant 3`. This is what *embedding* domain specific languages is all about!

Of course, we can still use the abstraction mechanisms of the host language to define more complicated expressions:

```
sum :: Int -> PrimExpr
```

```
sum n
  = if (n <= 0)
     then (constant 0)
     else (constant n .+. sum (n-1))
```

Embedding of abstract syntax consists of defining new *concrete* syntax tailored to the needs of the specific domain. Since most languages are unable to redefine their concrete syntax, we are constrained by the pecularities of the host language. For example, in Haskell this means that we are unable to write relational expressions directly. In this respect languages as Scheme that have an expressive macro system are better suited for embedding the abstract syntax. Recently however, there have been interesting proposals for adding powerful macro mechanisms (Peyton Jones and Sheard, 2002), or even mechanisms for extending the concrete syntax dynamically (Baars, 2002).

5.4.3 Step 3: Type embedding

The above embedding is already superior to unstructured strings since it is impossible to construct syntactically incorrect strings but it is still possible to construct *ill typed* requests: `constant 42 .==. constant "world"`. The `PrimExpr` data type is untyped and there is no mechanism to enforce that both arguments of the equality operator are of the same type.

We used abstract syntax trees to ensure that we can only generate syntactically correct expressions and fortunately we can use another trick to only generate type correct expressions. The *phantom types* that we used to encode inheritance (chapter 3) and pointer types (chapter 2) can also be used to type expressions.

We introduce a new polymorphic type `Expr a` such that `expr :: Expr a` means that `expr` is an expression of type `a`. The type variable `a` in the definition of the `Expr` data type is only used to hold a type – it does not occur in the right hand side of its definition and is therefore never physically present:

```
data Expr a  = Expr PrimExpr
```

The next step is to refine our combinators to encode the typing rules of the host language:

```
constant :: Show a => a -> Expr a
(.+.)    :: Expr Int -> Expr Int -> Expr Int
(.==.)   :: Eq a  => Expr a -> Expr a -> Expr Bool

constant x                = Expr (ConstExpr (show x))
(.+.)  (Expr x) (Expr y)  = Expr (BinExpr OpPlus x y)
(.==.) (Expr x) (Expr y)  = Expr (BinExpr OpEq x y)
```

Note that the definitions don't change except for packing and unpacking the `Expr` constructor. However, the type signature is needed to enforce a *less* general type than would be inferred by the type inferencer!

By making the `Expr` type an abstract data type, we ensure that only the primitive functions can manipulate the unsafe `PrimExpr` type. If we now use the combinators to construct an ill typed expression, `constant 42 .==. constant "World"`, the *Haskell type checker* will complain at compile time that the type `Expr Int` of the first operand doesn't match with the type `Expr String` of the second operand.

Typing expressions through phantom types immediately extends to values built using Haskell primitives. The example function `sum` for instance now has the type: `sum :: Int -> Expr Int`. Later we show how multiple phantom type variables can be used to define a type safe encoding of attribute selection in records.

5.5 Relational algebra

Before we give a precise translation from monad comprehensions to relational algebra, we need a good definition of the relation algebra. Surprisingly, it is hard to find a compact description that fits our needs and we will give a short definition of the algebra starting from basic set theory.

5.5.1 Functions

In this section, we first introduce some basic concrete mathematics definitions. For two sets A and B, a (partial) *function* f from A to B, denoted $f :: A \to B$, is a set of pairs (a, b) where $a \in A$ and $b \in B$. Every element a occurs at most once as the first component of a pair in f:

$$(a, b) \in f \ \wedge \ (a, c) \in f \ \Rightarrow \ b = c$$

Since each element a occurs at most once, we denote its corresponding element b as $f(a)$, i.e. $(a, b) \in f \Leftrightarrow f(a) = b$.

The domain of a function is defined as $dom(f) = \{ \ a \mid (a, b) \in f \ \}$. Dually, the codomain is defined as $codom(f) = \{ \ b \mid (a, b) \in f \ \}$. A function is called *finite* if its domain is a finite set. A *total* function $f :: A \to B$ is a function where every element in A occurs as a first component of a pair in f:

$$a \in A \ \Rightarrow \ a \in dom(f)$$

A restriction of a function f to a set $A' \subseteq A$ is defined as: $f_{|A'} = \{ \ (a, f(a)) \mid a \in A' \ \}$. Note that $dom(f_{|A}) = A$.

The *composition* of a function $f :: B \to C$ and a function $g :: A \to B$ is defined as:

$$(f \cdot g) = \{\ (a, c) \mid (a, b) \in g\ \wedge\ (b, c) \in f\ \}$$

The *extension* of a function $f :: A \to B$ with a function $g :: A \to B$ is defined as:

$$(f \odot g) = \{\ (a, b) \mid (a, b) \in g\ \vee\ ((a, b) \in f\ \wedge\ a \notin dom(g))\ \}$$

The *identity* function I_A is defined as: $I_A = \{\ (a, a) \mid a \in A\ \}$.

5.5.2 Relations

Let $\mathcal{A}$ be a set of names, called *attributes* a. $\mathcal{D}$ is a set of *domains* D. Domains are arbitrary, non-empty sets. Every attribute has an associated domain, i.e. there is a total function $domain :: \mathcal{A} \to \mathcal{D}$ that gives the domain of a certain attribute. For example, the attribute `age` probably has the natural numbers as its domain, $domain(\texttt{age}) = \mathbb{N}$.[1]

Tuples are the mathematical counterpart of database records. A *tuple* t is a finite function from $\mathcal{A}$ to $\mathcal{D}$, where every attribute is mapped to a value in its corresponding domain:

$$t(a) \in domain(a)$$

For example, the `persons` relation/database could contain a tuple/record like: $\{(\texttt{name}, daan), (\texttt{age}, 29)\}$.

The domain of a tuple is also called its *scheme*, $scheme(t) = dom(t)$. The *signature* of a tuple t is a function from each attribute to its domain. Let $\mathcal{T}$ the set of all tuples and $\mathcal{S}$ the set of all signatures. The function $signature :: \mathcal{T} \to \mathcal{S}$ is defined as:

$$signature(t) = \{\ (a, domain(a)) \mid a \in scheme(t)\ \}$$

A *relation* r is a finite set of tuples. Just as tuples, a relation has a signature. We assume a (now overloaded) function $signature :: \mathcal{R} \to \mathcal{S}$ that gives the signature of a relation. The *scheme* of a relation is the domain of its signature. It is required that each tuple in a relation has the same signature as the relation, that is:

$$\forall t.t \in r\ \Rightarrow\ signature(r) = signature(t)$$

[1] This is slightly more restrictive than the relation algebra defined by Codd, where attributes only have a fixed domain within a scheme.

Note that this implies that the scheme of a relation r equals the scheme of any tuple $t \in r$:

$$\begin{aligned} scheme(r) &= dom(signature(r)) \\ &= dom(signature(t)) \quad \text{where } t \in r \\ &= dom(\{\ (a, domain(a)) \mid a \in scheme(t)\ \}) \\ &= \{\ a \mid a \in scheme(t)\ \} \\ &= scheme(t) \end{aligned}$$

5.5.3 Relational expressions

The relational algebra is constructed using:

- Constant relations (T);
- Projection (π), rename (ρ) and restriction (σ);
- Union ($\cup$), difference ($-$) and cartesian product ($\times$).

We will discuss each of these operations in more detail in the following paragraphs.

Constant relations

We assume a universe of *constant relations*. A constant relation is written as T. In practical terms, these constant relations are an existing database with name T. Although we call it *constant* because it has the same value within an expression, such a relation may well change over time – as databases do in practice!

Projection and rename

Projection and rename are actually instances of a more general operation which we call *extension*. This operation is used to good effect within the Haskell embedding, simplifying the implementation considerably.

The extension operator $\delta_f(r)$ takes a finite function $f :: \mathcal{A} \to scheme(r)$ and a relation r as argument. It is defined as: $\delta_f(r) = \{\ t \cdot f \mid t \in r\ \}$. The scheme of δ is

derived as follows:

$$\begin{aligned} scheme(\delta_f(r)) &= scheme(\{\ t \cdot f \mid t \in r\ \}) \\ &= scheme(t \cdot f) \quad \text{where } t \in r \\ &= dom(t \cdot f) \\ &= dom(f) \end{aligned}$$

The projection operator $\pi_S(r)$ takes a scheme S and a relation r as argument. It is defined as: $\pi_S(r) = \delta_{(I_S)}(r)$. Note that $\pi_S(r)$ is only well defined when $S \subseteq scheme(r)$. The scheme of a projection is $scheme(\pi_S(r)) = scheme(\delta_{(I_S)}(r)) = dom(I_S) = S$.

The rename operator $\rho_f(r)$ takes a finite function f and a relation r, where $f :: \mathcal{A} \to scheme(r)$. It is defined as: $\rho_f(r) = \delta_g(r)$ where g is defined as:

$$g = f \cup I_{(scheme(r) - dom(f) - codom(f))}$$

The scheme of ρ is:

$$\begin{aligned} scheme(\rho_f(r)) &= scheme(\delta_g(r)) \\ &= dom(g) \\ &= dom(f \cup I_{(scheme(r) - dom(f) - codom(f))}) \\ &= dom(f) \cup (scheme(r) - dom(f) - codom(f)) \end{aligned}$$

5.5.4 Restriction

Before we can discuss restriction, we first take a look at *formulas*. Formulas are built using constants, attributes (in the role of variables) and functions.

We can perform substitution on a formula F in the context of a tuple t by replacing the attributes in the formula with their corresponding values from that tuple. This operation is written as $F[t]$ and associates to the left. We therefore have the following composition law: $F[t \cdot f] \ \Rightarrow\ F[f][t]$.

The scheme of a formula consists of all free attributes in the formula. Note that $F[t] = F[t_{|S}]$ where $scheme(F) \subseteq S$. When F is used in a restriction, the value $F[t]$ should be in the set of booleans $\mathcal{B}$.

The restriction operator σ is defined as: $\sigma_F(r) = \{\ t \mid t \in r\ \wedge\ F[t]\ \}$. The scheme of a restriction stays the same:

$$\begin{aligned} scheme(\sigma_F(r)) &= dom(t) \quad \text{where } t \in r \\ &= scheme(r) \end{aligned}$$

5.5.5 Union,Difference and Cartesian product

The union and difference on two relations r and s with $scheme(r) = scheme(s)$, are defined as set union and set difference respectively:

$$scheme(r \cup s) = scheme(r)$$
$$scheme(r - s) = scheme(r)$$

The cartesian product of two relations r and s where $scheme(r) \cap scheme(s) = \varnothing$ is defined as: $r \times s = \{ t \cup u \mid t \in r, u \in s \}$. The scheme of $\times$ is: $scheme(r \times s) = scheme(r) \cup scheme(s)$.

5.6 Embedding queries

Finally, we are able to describe the embedding of database queries within Haskell. We use the same three steps as in the embedding of the simple restriction expressions in the previous sections.

5.6.1 Step 1: Abstract syntax

The abstract syntax is modelled directly after the relational algebra.

```
type TableName  = String
type Attribute  = String
type Scheme     = [Attribute]
type Assoc      = [(Attribute,Attribute)]

data PrimQuery  = BaseTable TableName Scheme
                | Project   Assoc PrimQuery
                | Restrict  PrimExpr PrimQuery
                | Binary    RelOp PrimQuery PrimQuery
                | Empty

data RelOp      = Times
                | Union
                | Intersect
                | Divide
                | Difference
                deriving (Show)
```

The `Project` operator actually corresponds to the extension operator δ since it takes an association `Assoc` as argument and thus performs renaming too. We can

easily define some utility functions on these primitive queries, for example `scheme`, `assocFromScheme` and `compose`:

```
scheme :: PrimQuery -> Scheme
scheme query
  = case query of
      (Empty)              -> []
      (BaseTable nm attrs) -> attrs
      (Project assoc q)    -> map fst assoc
      (Restrict expr q)    -> scheme q
      (Binary op q1 q2)    -> case op of
                                Times       -> attr1 ++ attr2
                                Union       -> attr1
                                Intersect   -> attr1 \\ attr2
                                Divide      -> attr1
                                Difference  -> attr1
                              where
                                attr1         = scheme q1
                                attr2         = scheme q2

assocFromScheme :: Scheme -> Assoc
assocFromScheme scheme
  = map (\attr -> (attr,attr)) scheme

compose :: Assoc -> Assoc -> Assoc
compose assoc1 assoc2
  = map assoc2 (\(a1,a2) ->
                  (a1, case lookup a2 assoc1 of
                         Just a3 -> a3
                         Nothing -> error "partial compose")
```

Optimizing queries

In the same way, we can define more elaborate functions to perform useful optimizations. Currently, the library removes dead attributes and relations, merges projections and pushes restrictions into sub expressions.

```
optimize   :: PrimQuery -> PrimQuery
optimize
  = mergeProject . removeEmpty . removeDead . pushRestrict
```

Since each of these passes are fairly straightforward, we will only describe the `pushRestrict` optimization in more detail. Pushing restrictions 'down' can improve the efficiency of the final query. Take for example the following equivalent queries:

$$\sigma_{\texttt{price}<500}(\texttt{Boards} \times \texttt{Riders}) \quad \equiv \quad (\sigma_{\texttt{price}<500}\texttt{Boards}) \times \texttt{Riders}$$

The second query is likely to be more efficient since the first query has to build a potentially large intermediate cartesian product of both databases.

There are a whole set of 'laws' that enable this optimization. For example, we can prove that a projection followed by a restriction is the same as the restriction followed by the projection:

$$\begin{aligned}
\sigma_F(\delta_f(r)) &= \{\ t \mid t \in \delta_f(r)\ \wedge\ F[t]\ \} \\
&= \{\ t \mid t \in \{\ t_r \cdot f \mid t_r \in r\ \}\ \wedge\ F[t]\ \} \\
&= \{\ t_r \cdot f \mid t_r \in r\ \wedge\ F[t_r \cdot f]\ \} \\
&\Rightarrow \{\ t_r \cdot f \mid t_r \in r\ \wedge\ F[f][t_r]\ \} \\
&= \{\ t \cdot f \mid t \in \{\ t_r \mid t_r \in r\ \wedge\ F[f][t_r]\ \}\ \} \\
&= \{\ t \cdot f \mid t \in \sigma_{F[f]}(r)\ \} \\
&= \delta_f(\sigma_{F[f]}(r))
\end{aligned}$$

Another law that we need enables us to push a restriction into a branch of binary expression.

$$\begin{aligned}
\sigma_F(r \times s) &= \{\ t \mid t \in (r \times s)\ \wedge\ F[t]\ \} \\
&= \{\ t \mid t \in \{\ t_r\ \cup\ t_s \mid t_r \in r\ \wedge\ t_s \in s\ \}\ \wedge\ F[t]\ \} \\
&= \{\ t_r\ \cup\ t_s \mid t_r \in r\ \wedge\ t_s \in s\ \wedge\ F[t_r\ \cup\ t_s]\ \} \\
&= \{\ t_r\ \cup\ t_s \mid t_r \in r\ \wedge\ t_s \in s\ \wedge\ F[t_r]\ \}\ \textbf{iff}\ \mathit{scheme}(F) \cap \mathit{scheme}(s) = \varnothing \\
&= \{\ t\ \cup\ t_s \mid t \in \{\ t_r \mid t_r \in r\ \wedge\ F[t_r]\ \}\ \wedge\ t_s \in s\ \} \\
&= \{\ t\ \cup\ t_s \mid t \in \sigma_F(r)\ \wedge\ t_s \in s\ \} \\
&= \sigma_F(r) \times s
\end{aligned}$$

Of course, the same holds for the other branch if $\mathit{scheme}(F) \cap \mathit{scheme}(r) = \varnothing$. In the combinator library, we used these laws to implement the `pushRestrict` function:

```
pushRestrict :: PrimQuery -> PrimQuery
pushRestrict (Restrict x (Project assoc query))
  = Project assoc (pushRestrict (Restrict expr query))
  where
    expr  = substAttr assoc x

pushRestrict (Restrict x (Binary op query1 query2))
  | noneIn1   = Binary op query1 (pushRestrict (Restrict x query2))
  | noneIn2   = Binary op (pushRestrict (Restrict x query2)) query1
  -- otherwise fall through
  where
    attrs     = schemeOfExpr x
    noneIn1   = null (attrs `intersect` scheme query1)
    noneIn2   = null (attrs `intersect` scheme query2)

...
```

Generating SQL

Translating relational algebra expressions into SQL queries is not entirely straightforward. With older SQL dialects, we need to be very careful about name conflicts. Fortunately, the SQL/92 standard provides a rename operation on attributes and there is a straightforward translation from relational algebra into SQL/92 queries (Date, 1995).

The good part of having a solid intermediate form is that we can easily provide different back ends for different SQL dialects or even entirely different query languages like ASN.

Another advantage is that we can extend our relational algebra with new operators that are hard to express using the basic relational algebra. An example of these are *relational comparisons* (Date, 1995). It is extremely awkward to express comparisons between entire relations with just the relational algebra. Suppose for example that each `rider` can have multiple sponsors with a separate table `sponsored` that maps sponsors to riders. Suppose we want to know pairs of riders such that both are sponsored by exactly the same brands. This query is 'easily' expressed with relational comparisons, where formulas can contain relational expressions themselves:

$$\sigma_F(\delta_{(\texttt{name1=name})}\texttt{riders} \times \delta_{(\texttt{name2=name})}\texttt{riders})$$
where
$$F := (\pi_{(\texttt{brand})}(\sigma_{(\texttt{name=name1})}\texttt{sponsored})) = \pi_{(\texttt{brand})}(\sigma_{(\texttt{name=name2})}\texttt{sponsored}))$$

The equivalent query in the basic relational algebra requires about five times as many operations and is (very) difficult to understand.

SQL doesn't provide relational comparisons either and one has to resort to cumbersome negated existential quantifiers. The above query would become:

```
SELECT A.name AS name1, B.name AS name2
FROM riders AS A, riders AS B
WHERE NOT EXISTS
         (SELECT brand AS brand1 FROM sponsored
          WHERE name = name1
            AND NOT EXISTS (SELECT brand AS brand2 FROM sponsored
                            WHERE name = name2
                            AND brand1 = brand2))
```

Mind-boggling! Fortunately, queries like this can automatically be derived from relational comparisons. With a combinator library, we can easily extend the abstract syntax to contain relational comparisons and compile these expressions automatically into the basic relational operators or an equivalent SQL expression. One possible implementation extends the algebra with general relational restrictions:

```
data PrimQuery    = ...
                  | RelRestrict PrimRelExpr

data PrimRelExpr  = RelQuery PrimQuery
                  | RelBin   RelBinOp PrimRelExpr PrimRelExpr

data RelBinOp     = RelEq | RelSub | RelSubEq | ...
```

Just as with the formula and query embedding, we should now define some friendly combinators to write these expressions and extend the code generator to generate the corresponding SQL queries.

5.6.2 Step 2: Abstract syntax embedding

We could proceed as in our earlier example and define some friendly combinators for building relational expressions. However, there is a serious drawback to using relational expressions directly as our programming language. In the relational algebra, attributes are only specified by name – there is no separate binding mechanism to distinguish attributes from different tables. For example, when we take the cartesian product of a relation with itself we are forced to rename every attribute to avoid ambiguity. Indeed, the only reason why the join operator ($\bowtie$) exists, is to capture a common combination of renaming, selection, projection and cartesian products.

Besides covering only a specific combination, it is also notoriously hard to type check join expressions (Buneman and Ohori, 1996) and we haven't found a way to embed those typing rules in Haskell. The join operation is problematic in a polymorphic setting, since the result type of a join expression depends on the common attributes of the arguments. Buneman and Ohori (1996) describe a restricted polymorphic type system for join expressions by adding specialized type constraints just for this operation.

SQL solves the renaming problem by using *qualified* attributes. There is a binding mechanism to assign names to relational expressions, `riders AS X`, and qualified attributes to refer to specific attributes in a certain relation, `X.name`. For example, a query that returns pairs of riders that are sponsored by the same brand is written as:

```
SELECT X.name Y.name
FROM riders AS X, riders AS Y
WHERE X.brand = Y.brand
  AND X.name <> Y.name
```

We will use the same approach in Haskell where monad comprehensions are used to introduce a custom binding mechanism. Instead of identifying attributes just by

name, both a relation and name is used. The above query is formulated in Haskell as:

```
do{ x <- table riders
  ; y <- table riders
  ; restrict (x!brand .==. y!brand)
  ; restrict (x!name  .<>. y!name)
  ; project (name1 = x!name, name2 = y!name)
  }
```

Under the hood, we still generate relational algebra expressions but all the renaming is done automatically within the combinators. The following sections explain in detail how the transformation from monad comprehensions to relational expressions works.

Monad comprehensions

A monad (Wadler, 1992a; Wadler, 1992b) is defined by two operations, *bind* (`>>=`) and `return`. The bind operation combines monadic operations and the return operation lifts values into the monad. The `do` notation is syntactic sugar that gets translated into the basic monadic operations:

$$\begin{aligned} \texttt{do}\{x \texttt{ <- } E\texttt{; } F\} &\equiv E \texttt{ >>= } (\backslash x \texttt{ -> do}\{F\}) \\ \texttt{do}\{E\texttt{; } F\} &\equiv E \texttt{ >>= } (\texttt{\_ -> do}\{F\}) \\ \texttt{do}\{E\} &\equiv E \end{aligned}$$

To understand the implementation of the query monad in Haskell better, we also give a denotational semantics that describes the correspondence between a monad comprehension and a relational algebra expression.

Each monadic rule ($\mathcal{M}$) in the semantics returns a tuple containing the relational algebra expression r that is being build and the returned Haskell value. Each rule takes an environment of currently bound variables and the current relation as arguments.

The semantic rule for `return` does nothing – it returns the current relation and returned value unmodified. The *bind* rule applies the monadic scheme to the sub expression E which returns a new relation and some value which is bound in the environment when translating F.

$$
\begin{array}{lll}
\mathcal{M}[\![\ \texttt{return}\ x\ ]\!]\ \mathcal{E}\ r & := & (r, x) \\
\mathcal{M}[\![\ \texttt{do}\{x\ \texttt{<-}\ E;\ F\}\ ]\!]\ \mathcal{E}\ r & := & \mathcal{M}[\![\ F\ ]\!]\ \ \mathcal{E}'\ \ r' \\
 & & \textbf{with} \\
 & & \quad (r', f) \quad = \mathcal{M}[\![\ E\ ]\!]\ \mathcal{E}\ r \\
 & & \quad \mathcal{E}' \quad\ \ = \mathcal{E} \odot \{(x, f)\}
\end{array}
$$

The corresponding implementation in Haskell is straightforward. First we define a data type for our `Query` monad:

```
data Query a     = Query (QueryState -> (a,QueryState))
type QueryState  = (Int,PrimQuery)
```

The state of the monad contains not only the current relational algebra expression r in the form of a `PrimQuery` but also an integer that is used for creating unique names.

The environment $\mathcal{E}$ in the semantics is no longer explicitly present in the Haskell implementation but it is *implicit* in the lambda-bound variables. Since Haskell is sometimes called a domain specific language for denotational semantics (Hudak, 1998), it is not surprising that the semantic rules translate almost literally into Haskell (modulo the implicit environment):

```
instance Monad Query where
  return x         = Query (\r -> (x,r))
  (Query e) >>= m  = Query (\r -> let (x,r')    = e r
                                      (Query f) = m x
                                  in  (f r'))
```

There are two other primitive operations give access to the state of the monad. The `updateQuery` function applies a function to the current relation and the `unique` function returns a unique integer.

```
updatePrimQuery :: (PrimQuery -> PrimQuery) -> Query PrimQuery
updatePrimQuery f
  = Query (\(i,qt) -> (qt,(i,f qt)))

unique :: Query Int
unique
  = Query (\(i,qt) -> (i,(i+1,qt)))
```

Constants

Existing databases ('constants') are opened with the `table` combinator. The effect is to take the cartesian product with the current relation. Remember that the

cartesian product $r \times s$ is only valid when $scheme(r) \cap scheme(s) = \varnothing$, and thus, the `table` operation needs to rename all the attributes into unique names in order to avoid name clashes. In the semantic rules, the a^* notation means a uniquely renamed attributed a.

$$\begin{aligned} \mathcal{M}[\![\texttt{table}\ T]\!]\ \mathcal{E}\ r &:= (r \times \delta_f(T), f^{-1}) \\ &\quad \textbf{with} \\ &\quad\ f = \{\ (a^*, a) \mid a \in scheme(T)\ \} \end{aligned}$$

As we can see from the rules, a unique renaming f is created. The returned value contains the inverse of this function – a mapping from the original attribute names into the unique ones. We call this mapping an association. This means that Haskell values bound by a table expression are not actual relations but associations that can be applied onto the hidden relational algebra expression to access an attribute. The attribute selector `(!)` uses this association to map an association/attribute pair into an unambiguous and unique attribute name. The scheme $\mathcal{F}$ translates expressions into relational formulas.

$$\begin{aligned} \mathcal{F}[\![x\ \texttt{!}\ a]\!]\ \mathcal{E} &:= \mathcal{E}[x]\ a \\ \mathcal{F}[\![e_1\ \texttt{.==.}\ e_2]\!]\ \mathcal{E} &:= (\mathcal{F}[\![e_1]\!]\ \mathcal{E}) = (\mathcal{F}[\![e_2]\!]\ \mathcal{E}) \end{aligned}$$

Before we can translate these rules in Haskell we first define data types for tables, associations, and attributes:

```
data Rel    = Rel Assoc
data Attr   = Attr Attribute
data Table  = Table TableName Scheme
```

Later we will see how we can use phantom types (again!) to add a typed layer around the code that we write now. The `Rel` type is used for associations. It is called after relations since that is how the user of the library will think of it – only the implementor is interested in the association that it holds. Again, the following Haskell implementation is almost a direct translation of the semantic rules.

```
(!) :: Rel -> Attr -> Expr a
(Rel assoc) ! (Attr attr)
  = case lookup attr assoc of
      Just realname -> Expr (AttrExpr realname)
      Nothing       -> error ("unknown attribute " ++ show attr)

table :: Table -> Query Rel
table (Table name scheme)
  = do{ assoc <- uniqueAssoc scheme
      ; updatePrimQuery
          (\q -> Times q (Project assoc (BaseTable name scheme)))
```

```
      ; return (Rel (inverse assoc))
      }

uniqueAssoc :: Scheme -> Query Assoc
uniqueAssoc scheme
  = do{ i <- unique
      ; return (map (\attr -> (attr ++ show i,attr)) scheme)
      }

inverse :: [(a,b)] -> [(b,a)]
inverse xs
  = map (\(a,b) -> (b,a)) xs
```

Other operations

Other operations are binary operations like `union` and `difference`, and projections and restrictions. Except for the renaming, the rules are straightforward:

$$\mathcal{M}[\![\,\texttt{restrict}\ e\,]\!]\ \mathcal{E}\ r \quad := \ (\sigma_{(\mathcal{F}[\![\,e\,]\!]\ \mathcal{E})}(r), \varnothing)$$

$$\begin{array}{lll}\mathcal{M}[\![\,\texttt{project}\ p\,]\!]\ \mathcal{E}\ r & := & (\delta_h(r), f^{-1}) \\ & \textbf{with} & \\ & f & = \{\ (a^*, a) \mid a \in dom(p)\ \} \\ & g & = p \cdot f \\ & h & = I_{scheme(r)}\ \cup\ g\end{array}$$

$$\begin{array}{lll}\mathcal{M}[\![\,\texttt{union}\ m_1\ m_2\,]\!]\ \mathcal{E}\ r & := & ((\delta_{g_1}(r_1)\ \cup\ \delta_{g_2}(r_2)) \times r, f^{-1}) \\ & \textbf{with} & \\ & (r_1, f_1) & = \mathcal{M}[\![\,m_1\,]\!]\ \mathcal{E}\ r \\ & (r_2, f_2) & = \mathcal{M}[\![\,m_2\,]\!]\ \mathcal{E}\ r \\ & f & = \{\ (a^*, a) \mid a \in dom(f_1\ \cup\ f_2)\ \} \\ & g_1 & = \{\ (a, f_1(b)) \mid (a, b) \in f\ \} \\ & g_2 & = \{\ (a, f_2(b)) \mid (a, b) \in f\ \}\end{array}$$

The translation to Haskell is straighforward and we will only show the implementation for `project` here:

```
project :: Assoc -> Query Rel
project assoc
  = do{ assocF <- uniqueAssoc (map fst assoc)
      ; updatePrimQuery (\q ->
      ;   Project
            (assocFromScheme (scheme q) ++ compose assoc assocF) q)
      ; return (Rel (inverse assocF))
      }
```

Liveness of attributes

It might come as a surprise to the reader that both `project` and `union` actually *extend* the scheme with new attributes instead of replacing them. This is because the user might still refer to an older attribute. Here is a (contrived) example:

```
do{ x <- table boards
  ; y <- project (reducedPrice = x!price)
  ; project (price = y!reducedPrice, brand = x!brand)
  }
```

In general, we can not predict the life-time of a bound lambda expression that arises from the monadic style. This means that the current translation will produce lots of dead attributes – attributes that are present in each sub-relation but discarded in the final projection. In practice, it is therefore fairly essential to perform the `removeDead` optimization that removes these dead attributes. Arrow-style combinators have the property that the life-times of their bound variables are made explicit (Hughes, 2000) and would remove the need for a seperate `removeDead` pass. However, arrow-style combinators are somewhat harder to use in practice than monadic style combinators.

Proof of liveness

Actually, the first implementations of Haskell/DB wrongly discarded certain live attributes. Since the problem only showed up in contrived or highly complex queries the bug went undiscovered and was only spotted after writing down the intended formal correspondence between monad comprehensions and relational queries. The bug showed up when we tried to prove that 'live' attributes are always part of the scheme of the relation. Of course, we were unable to do so in our first semantics since the translation was plainly wrong! This lead to the current semantics from which the current implementation is derived. Moreover, the implementation also improved a lot with respect to modularity and clearness of expression by using the semantics as a template.

The proof of liveness centers around the attribute selector (`!`):

$$\mathcal{F}[\![\, x \texttt{ ! } a \,]\!]\, \mathcal{E} \quad := \quad (\mathcal{E}(x))(a)$$

This translation is only valid when the following conditions hold:

- $x \in \mathcal{E}$. This condition always holds in the Haskell implementation since the environment is implicit in the lambda bound variables and thus checked at compile time.

- $\mathcal{E}(x) :: \mathcal{A} \to \mathcal{A}$. This condition is enforced by the Haskell type system.
- $a \in dom(\mathcal{E}(x))$. This condition is also enforced by the Haskell type system as described in the next section.

The second condition garantees the value x has the correct type – a renaming from attributes to attributes. However, in order to ensure that no arbitrary renamings are allowed, we will strengthen the condition and only allow renamings that are constructed by the basic combinators. We call these renamings *associations*. We write $assoc(f)$ when the renaming f is an assocation. By using an abstract type for assocations (`Rel`) we can ensure within the Haskell implementation that no arbitrary renamings can be used with attribute selection. We write $\overline{\mathcal{E}}$ to denote the environment constrained to association values:

$$\overline{\mathcal{E}} = \{\ (x, f) \mid (x, f) \in \mathcal{E} \wedge assoc(f)\ \}$$

The substituted attribute names, $\mathcal{E}(x)(a)$, will eventually be used from a restriction context:

$$\mathcal{M}[\![\ \texttt{restrict}\ e\ ]\!]\ \mathcal{E}\ r \quad := \quad (\sigma_{(\mathcal{F}[\![\ e\ ]\!]\ \mathcal{E})}(r), \varnothing)$$

The substituted formula is only correct when all possible substituted attributes will be part of the scheme of the relation, that is:

$$codom(\bigcup codom(\overline{\mathcal{E}})) \subseteq scheme(r)$$

This clearly holds for the initially empty environment and all the rules that don't use the $\mathcal{M}$ scheme. That only leaves the rule for monadic bind:

$$\begin{array}{lll} \mathcal{M}[\![\ \texttt{do}\{x\ \texttt{<-}\ E\texttt{;}\ F\}\ ]\!]\ \mathcal{E}\ r & := & \mathcal{M}[\![\ F\ ]\!]\ \ \mathcal{E}'\ \ r' \\ & & \textbf{with} \\ & & \quad (r', f) \quad = \mathcal{M}[\![\ E\ ]\!]\ \mathcal{E}\ r \\ & & \quad \mathcal{E}' \qquad = \mathcal{E} \odot\ \{(x, f)\} \end{array}$$

By induction, we need to ensure that in the application of the monadic rule the condition holds for the newly constructed environment:

$$codom(\bigcup codom(\overline{\mathcal{E}'})) \subseteq scheme(r')$$

There are two cases to consider. The first is the case where f is an assocation:

$$\begin{aligned} & codom(\bigcup codom(\overline{\mathcal{E}'})) \\ = {} & codom(\bigcup codom(\overline{\mathcal{E} \odot\ \{(x, f)\}})) \\ \subseteq {} & codom(\bigcup codom(\overline{\mathcal{E}} \cup \{(x, f)\}) \\ \subseteq {} & codom(\bigcup codom(\overline{\mathcal{E}})) \cup codom(f) \end{aligned}$$

And secondly, the case where f is not an assocation:

$$\begin{aligned} & codom(\bigcup codom(\overline{\mathcal{E}'})) \\ = & \; codom(\bigcup codom(\overline{\mathcal{E} \odot \{(x, f)\}})) \\ = & \; codom(\bigcup codom(\overline{\mathcal{E}})) \end{aligned}$$

For each rule $\mathcal{M}[\![\ e\ ]\!]\ \mathcal{E}\ r = (r', f)$ we now have to show that $codom(\bigcup codom(\overline{\mathcal{E}}\ )) \cup codom(f) \subseteq scheme(r')$ when $assoc(f)$ holds, or that $codom(\bigcup codom(\overline{\mathcal{E}})) \subseteq scheme(r')$ when f is not an assocation.

The proof is done by straightforward induction. Here is the case for `project`.

$$\begin{aligned} \mathcal{M}[\![\ \texttt{project}\ p\ ]\!]\ \mathcal{E}\ r \quad & := \ (\delta_h(r), f^{-1}) \\ & \quad \textbf{with} \\ & \quad\ \ f \quad = \{\ (a^*, a) \mid a \in dom(p)\ \} \\ & \quad\ \ g \quad = p \cdot f \\ & \quad\ \ h \quad = I_{scheme(r)}\ \cup\ g \end{aligned}$$

Since f^{-1} is an association, we have:

$$\begin{aligned} & codom(\bigcup codom(\overline{\mathcal{E}}))\ \cup\ codom(f^{-1}) \\ = & \; codom(\bigcup codom(\overline{\mathcal{E}}))\ \cup\ dom(f) \\ = & \; codom(\bigcup codom(\overline{\mathcal{E}}))\ \cup\ dom(p \cdot f) \\ = & \; codom(\bigcup codom(\overline{\mathcal{E}}))\ \cup\ dom(g) \\ \subseteq & \; scheme(r)\ \cup\ dom(g) \\ = & \; scheme(\delta_h(r)) \quad \square \end{aligned}$$

The other cases are equally structured. Just as in the previous chapter, proving properties like liveness give a lot more confidence in the actual implementation. Moreover, the exact semantics have been invaluable in a clear implementation of the combinators.

5.6.3 Step 3: Type embedding

We have already made the expression language type safe by using phantom types. The same trick is used to add a typed layer to the comprehensions. Central to the discussion is the attribute selection operator:

```
(!) :: Rel -> Attr -> Expr a
```

Given a relation and an attribute name, the operator returns the attribute value expression. Given that any attribute always has a well defined type, we parameterize an attribute by its type to return an expression of the same type:

```
data Attr a = Attr Attribute

(!) :: Rel -> Attr a -> Expr a
```

The type of an attribute is now connected to the type of the resulting expression. That was easy! Unfortunately, the type system does not prevent us yet from selecting a non-existing attribute from the relation. The solution is to parameterize the `Rel` type by its "scheme". Similarly, we parameterize the `Attr` type again by both the scheme of the relation and the type of the attribute:

```
data Rel r    = Rel Assoc
data Table r  = Table TableName Scheme
data Attr r a = Attr Attribute
```

The `Rel` and `Table` both retain their associated scheme since we need the actual values to create the primitive query. The types are just to assure that this will always succeed!

The selection operator `(!)` now expresses in its type that given a relation with scheme `r` that has an attribute of type `a`, it returns a value expression of type `a`.

```
(!) :: Rel r -> Attr r a -> Expr a
```

The type signature is almost right, but not yet what we want. We would like to ensure that the scheme `r` contains *at least* the attribute but the scheme can of course contain many other attributes. A better type signature, although not possible within Haskell, would be:

```
(!) :: (name ∈ r) => Rel r -> Attr name a -> Expr a
```

There are different (partial) solutions to this problem but a particularly nice solution are the *Typed Record Extensions* (TREX) of Gaster and Jones (1996).

TREX

TREX extends the Haskell language with extensible records. As an experimental system, the feature is currently only available by the Hugs implementation of Haskell. A record is an association list of field-value pairs. For example:

```
(x = 3, even = False) :: Rec (x :: Int, even :: Bool)
```

This record has two fields, `x` of type `Int` and `even` of type `Bool`. Note that the type `(x :: Int, even :: Bool)` has kind `Row`. The special *type* constructor `Rec` takes a type of kind `Row` into a record (with kind `*`), ie. `Rec :: Row -> *`.

A record of type `Rec r` can be *extended* by a field `z` provided that `z` doesn't already occur in `r`. This is indicated by the constraint `r\z`. The type of a function that adds a field `foo` to a record becomes:

```
extendWithFoo :: r\foo => a -> Rec r -> Rec (foo :: a | r)
extendWithFoo a r  = (foo = a | r)
```

Unfortunately, labels are not first class values in TREX, so we cannot write a generic function that extends a given record with a new field:

```
-- WRONG
extendWith field a r  = (field = a | r)
```

Schemes as TREX records

Instead of adding a qualified type that expresses that an scheme should contain at least a certain attribute name, we make the attribute definitions polymorphic in their row. The type signature for attribute selection stays the same as previously defined, but now with a different kind for `r`!

```
(!) :: Rel r -> Attr r a -> Expr a
```

The lack of first class labels means that we have to repeat a lot of code that only differs in the name of some labels. This shows up mostly for the definitions of attributes. For every attribute *attr* we define an attribute definition with the following type:

```
attr :: r\attr => Attr (attr :: Expr a | r ) a
```

This type means that the attribute *attr* can be applied to any *row* (scheme) with at least field *attr* with type `Expr a` and possibly other fields `r`. Similarly, for every base table with scheme r we have a definition with type `Table r`. For the example database we have:

```
boards   :: Table (brand :: Expr String, model :: Expr String
```

```
                , price :: Expr Int, freeride :: Expr Bool)
boards   = Table "boards"
                 (attributes [("brand","model","price","freeride")])

brand    :: r\brand => Attr (brand :: Expr String | r) String
brand    = Attr "brand"

model    :: r\model => Attr (model :: Expr String | r) String
model    = Attr "model"

price    :: r\price => Attr (price :: Expr Int | r) Bool
price    = Attr "price"

freeride :: r\freeride => Attr (freeride :: Expr Bool | r) Bool
freeride = Attr "freeride"
```

Note that the type definitions are required to give each definition a less general type than the type inferencer would infer. The definitions of the constants (attributes and tables) are unsafe since they have to be defined by the user for each particular database. This is where the connection is weak again since attributes are represented by simple strings. However, we have written a tool, called DB/Direct, that queries the system tables and *automatically* generates the suitable database definition in Haskell. This tool is written with Haskell/DB itself! This means that the Haskell sources are dependent on the database and a type safe system should automatically re-generate and re-compile the client programs whenever the database definition changes.

Kind annotations

The type checker still complains about the library as it stands. Both the `Attr` and `Rel` data types take a type of kind `Row` instead of `*`. However, the kind inferencer assigns the kind `Rel,Attr :: * -> *` for these datatypes and doesn't accept a type signature that applies such datatype to a row – `Attr (price :: Int | r)`.

A *kind signature* should be added to the definition of these datatypes. Unfortunately, Haskell doesn't provide a way to do that explicitly. One solution is to add a *phantom constructor* – a constructor that is never used but just added to force the type inferencer to assign a specific kind to a *phantom type*.

```
data Rel r    = Rel Assoc
              | RelKind (Rec r)

data Attr r a = Attr Attribute
              | AttrKind (Rec r)
```

```
data Table r  = Table TableName Scheme
              | TableKind (Rec r)
```

We use the special type constructor `Rec :: Row -> *` to assign a `Row` kind to the phantom type variable.

Another way to guide the kind inferencer is the addition of a class constraint. First we define a class with a single *phantom function* that assigns the correct kind to the class parameters:

```
class RecKind r where
  recKind :: Rec r

instance RecKind r
```

The single instance declaration allows the type inferencer to instantiate any type variable to the `RecKind` class but only if the kind can be instantiated into `Row -> *`. By adding a class constraint to the data type declaration, the kind inferencer will assign the correct kinds to the phantom type variables.

```
data RecKind r => Attr r a   = Attr Attribute
```

Since the `recKind` function is never used in the program, the dictionary for the `RecKind` class is also never used and passed as a hidden parameter. The last solution can be considered better than a phantom constructor since a `data` declaration can sometimes be changed into a more efficient `newtype` declaration.

```
newtype RecKind r => Attr r a   = Attr Attribute
```

The best solution and supported in the latest release of the Glasgow Haskell compiler, is to add proper kind signatures to Haskell and hopefully these will also be supported by the next standard Haskell definition.

Typing projections

The type signature for projections becomes with TREX:

```
project :: Rec r -> Query (Rel r)
```

A projection takes a record whose labels correspond with the attributes of the resulting relation. For example:

```
cheap :: Query (Rel (model :: Expr String))
cheap =
  do{ x <- boards
    ; restrict (x!price .<. constant 500)
    ; project (model = x!model)
    }
```

Unfortunately, there is nothing that prevents us from writing an arbitrary expression as the label value, instead of an attribute expression:

```
project (model = "wrong") :: Query (Rel (model :: String))
```

This is a point where we can no longer model the type system of relational queries in Haskell. But fortunately, the problem is not so bad as it seems: sooner or later, the query is used in a context that expects the relation to contain attribute expressions and the type checker will complain at the use site of this expression.

Still, the implementation of `project` has to overcome some obstacles that are related to the above problem. In contrast to the earlier definition that was passed an explicit relation association, the `project` function now needs to reconstruct the association `Assoc` from a TREX record: the labels and its values.

```
project :: ShowRecRow r => Rec r -> Query (Rel r)
project rec
  = do{ let assoc = zip (labels rec) (values rec)
      ; assocF <- uniqueAssoc (map fst assoc)
      ; updatePrimQuery (\q ->
      ;   Project
            (assocFromScheme (scheme q) ++ compose assoc assocF) q)
      ; return (Rel (inverse assocF))
      }
```

The function `labels` returns the labels of a TREX record and the function `values` returns the values of the record. A problem with these functions is that it potentially destroys referential transparency. TREX therefore defines a canonical ordering on the labels. Another problem is that the polymorphic values have to be converted into `String`s. The TREX implementation contains the special `ShowRecRow` class for this purpose:

```
class ShowRecRow r where
  showRecRow :: Rec r -> [(String, ShowS)]
```

The function `showRecRow`, together with `eqRecRow`, are the only generic functions on records. The class `ShowRecRow` is known to the type checker, which assures that a row `r` that has a `ShowRecRow r` constraint is never extended with values that are not in the `Show` class.

It is easy now to define the `labels` and `values` functions:

```
labels, values :: ShowRecRow r => Rec r -> [String]
labels r = map fst (showRecRow r)
values r = map (\(l,v) -> v "") (showRecRow r)
```

Of course, since the values may be arbitrary expressions (as long as they are part of the `Show` class), the resulting value strings may be illegal attribute names, but either the result is never used (and thus never evaluated under a lazy evaluation strategy) or the type checker will complain at the use site.

Type checked queries

Finally, the Haskell typechecker can check the consistency of our queries. For example:

```
cheap :: Query (Rel (model :: Expr String))
cheap =
  do{ x <- boards
    ; restrict (x!price .<. constant 500)
    ; project (model = x!model)
    }
```

As it stands, the query is correct. But when we would use an illegal attribute, like `x!name`, the type checker would complain that it can't unify the types of `x` and `name`:

```
Type checking
ERROR "SAMPLE.HS" (line 63): Type error in application
*** Expression    : x ! name
*** Term          : x
*** Type          : Rel
                     (brand :: Expr String, model :: Expr String
                     , price :: Expr Int, freeride :: Expr Bool)
*** Does not match: Rel
                     (name :: a
                     , brand :: Expr String, model :: Expr String
                     , price :: Expr Int, freeride :: Expr Bool | b)
*** Because       : rows are not compatible
```

5.7 Exam marks

In this final section we explore how different combinator libraries can be combined to program a simple web server. Any commercial exploitation of the web today uses server-side scripts that connect to a database and deliver HTML pages composed from dynamic data that is obtained from querying the database using information in the client's request. The following example is a is a simple server-side web script that generates an HTML page for a database of exam marks and student names.

The database is accessed via simple web page with a text entry and a submit button. The HTML contains a form element that submits the query to the `getMark` script on the server:

```
<HTML>
<HEAD> <TITLE>Find my mark</TITLE> </HEAD>
<BODY>
  <FORM ACTION="getMark.asp" METHOD="post">
    My name is:
    <INPUT TYPE=text NAME="name">
    <INPUT TYPE="submit" VALUE="Show my mark">
  </FORM>
</BODY>
</HTML>
```

5.7.1 Visual Basic

Even the simplest Visual Basic solution uses no less than four different languages. Visual Basic for the business logic and glue, SQL for the query, and HTML with ASP directives to generate the result page.

1. In ASP pages, scripts are separated from the rest of the document by `<%` and `%>` tags. The prelude script declares all variables, construct the query and retrieves the results from the students database. The ASP `Request` object contains the information passed by the client to the server. The `Form` collection contains all the form-variables passed using a `POST` query. Hence `Request.Form("name")` returns the value that the user typed into the `name` textfield of the above HTML page.

   ```
   <%
   Q =     "SELECT student.name, student.mark"
   Q = Q & " FROM Students AS student"
   Q = Q & " WHERE "student.name = "
   Q = Q & Request.Form("name")
   ```

```
Set RS = CreateObject("ADO.Recordset")
RS.Open Q "CS101"
%>
```

2. The body contains the actual HTML that is returned to the client, with a table containing the student's name and mark. The <%= and %> tags enclose Visual Basic expressions that are included in the output text. Thus the snippet:

```
<TR>
  <TD><%=RS("name")%></TD>
  <TD><%=RS("mark")%></TD>
</TR>
```

creates a table row that contains the name and the mark of the student who made the request:

```
<HTML>
<HEAD> <TITLE>Marks</TITLE> </HEAD>
<BODY>
  <TABLE BORDER="1">
    <TR>
      <TH>Name</TH>
      <TH>Mark</TH>
    <TR>
    <%Do While Not RS.EOF%>
    <TR>
      <TD><%=RS("name")%></TD>
      <TD><%=RS("mark")%></TD>
    <TR>
    <%RS.MoveNext%>
    <%Loop%>
  </TABLE>
</BODY>
</HTML>
```

3. The clean-up phase disconnects the databases and releases the recordset:

```
<%
RS.Close
set RS = Nothing
%>
```

5.7.2 Haskell

The Haskell version of our example web page is more coherent than the Visual Basic version. Instead of four different languages, we need only need Haskell embedded in a minimal ASP page (Meijer, 2000):

```
<%@ LANGUAGE=HaskellScript %>
<%
module Main where

import Asp
import HtmlWizard

main :: IO ()
main = wrapper $ \request ->
  do{ name <- request # lookup "name"
    ; r <- runQuery (queryMark name) "CS101"
    ; return (markPage r)
    }
```

The function `queryMark` is the analog of code in the prelude part of the Visual Basic page, except here it is defined as a separate function parameterized on the name of the student:

```
type Student = Row(name :: Expr String, mark :: Expr Char)

queryMark :: String -> Query (Rel Student)
queryMark n =
  do{ student <- table students
    ; restrict (student!name .==. constant n)
    ; project ( name = student!name, mark = student!mark)
    }
```

Function `markPage` makes a nice HTML page from the result of performing the query:

```
markPage :: [Row Student] -> HTML
markPage xs =
  page "Marks"
    [ table ( headers = [ "Name", "Mark"]
            , rows = [[x!.name, x!.mark] | x <- xs ] )
    ]
%>
```

The Haskell program is more concise and more modular than the Visual Basic version. Functions `queryMark` and `markPage` can be tested separately, and perhaps even more important, we can easily reuse the complete program to run in a traditional CGI-based environment, by importing the `CGI` module instead of `Asp` (in a language such as Standard ML we would have parameterized over the server interface).

5.8 Status and conclusions

The main lesson of this chapter is a new design principle for embedding domain specific languages where embedded programs are compiled on-the-fly and executed by submitting the target code to a server component for execution. We have shown how to embed SQL into Haskell using this principle, but there are numerous other possible application domains where embedded compilers are the implementation technology of choice; many UNIX services are accessible using a completely text-based protocol over sockets.

Our ultimate goal for a Domain Specific Embedded Compiler is to provide hard compile-time guarantees for type safety and syntactic correctness of the generated target program. Syntactic correctness of target programs can be garanteed by hiding the construction of programs behind abstract data types. Phantom types, polymorphic types whose type parameter is only used at compile-time but whose values never carry any value of the parameter type, are a very elegant mechanism to impose the Haskell type system on the embedded language.

Our final example shows how Domain Specific Embedded Compilers can make server-side web scripting more productive. Because we can leverage on the abstraction mechanisms of Haskell (higher-order functions, module system), compared to the VB solution, the Haskell program is of higher quality, and easier to change and maintain.

Both the Haskell/DB and the DB/Direct packages are available on the web at `http://www.haskell.org/haskellDB`.

Chapter 6

The Lazy Virtual Machine

6.1 Introduction

This chapter describes the Lazy Virtual Machine (LVM). Just like the JVM (Lindholm and Yellin, 1999), it defines a portable instruction set and file format. However, it is specifically designed to execute languages with non-strict (or lazy) semantics. Is there need for such system? After all, there are many compilers and interpreters for lazy languages, for example, GHC developed at the Glasgow university, the HUGS interpreter by Mark Jones, NHC from York university (UK), the HBC compiler developed at Chalmers, and Clean from Nijmegen. One may think that with this diversity of systems there is no need anymore for other compilers, as most implementation issues have been resolved.

However, the current compilers and interpreters have become large systems that are hard to adapt – it has become difficult to experiment with new type systems, language constructions, compiler transformations, profiling, or debugging tools. In particular, the work on embedded languages as described in the previous two chapters gave rise to experimentation with various extensions to the Haskell language. Eventually, this lead to the development of the LVM:

- A small portable system that can be easily adapted to support different (experimental) languages and type systems.
- A simple and robust instruction set that is an easy target for compiler frontends.
- Efficient interpretation or JIT compilation is possible.
- A toolkit that translates an *untyped*, rich intermediate language (λ_{core}) to LVM instructions. Note that we use untyped expressions in order to experiment

with extensions that are hard to type at this level, examples include *type indexed records* (Shields, 2001) and dependent types (Augustsson, 1998).

The LVM is currently implemented on top of the OCaml runtime system (Leroy, 1990; Leroy, 1995). The system runs on many platforms, including Windows, various Unix's, MacOSX and 64-bit platforms like the DEC alpha. The LVM is used as a backend for the experimental HX system (Shields and Peyton Jones, 2001) and the Helium compiler – this compiler implements a very large subset of Haskell and is currently used to teach first year students Haskell at Utrecht University.

The design and implementation of the LVM is as simple and modular as possible. However, simplicity does not imply that it is a toy system; the implementation is full fledged including support for exotic features as (asynchronous) exceptions, concurrency, a foreign function interface, generational garbage collection, and execution traces.

This chapter will focus on the translation of the intermediate language to LVM instructions, and on the operational semantics of the instructions themselves. Many items of this chapter have been described before, and the main contribution of this chapter is the *design* of a 'real world' instruction set, operational semantics, and translation scheme as a whole. More specifically:

- We define a naive and straightforward translation scheme from the low-level λ_{lvm} language to LVM instructions. Instead of defining many *optimized* translation schemes (Peyton Jones, 1986; Johnsson, 1984; Plasmeijer and van Eekelen, 1993), we define a small set of rewrite rules on instructions that achieve the same effect. The correctness of the rewrite rules is relatively easy to prove with the operational semantics. In contrast, an optimized translation scheme is much harder to prove correct, as one has to show a correspondence between the operational semantics of the λ_{lvm} language and the generated instructions. Furthermore, the rewrite rules are most of the time even more effective than optimized translation rules, as the rewrite rules sometimes find optimization opportunities between instructions that are unrelated at the language level.
- The low-level λ_{lvm} language has an operational reading and directly reflects the capabilities of the abstract machine. As such, we are able to reason about denotationally equivalent expressions that have a different operational behaviour.
- We define simple operational semantics for the instructions. A state is determined by just three items: the current code, the heap, and the stack. Besides being simpler than many other instruction sets (Peyton Jones, 1992; Peyton Jones, 1986; Johnsson, 1984; Plasmeijer and van Eekelen, 1993), the instructions also map directly onto C instructions, reducing the number of bugs in an implementation and improving our understanding of the relationship between abstract machine and concrete implementation.

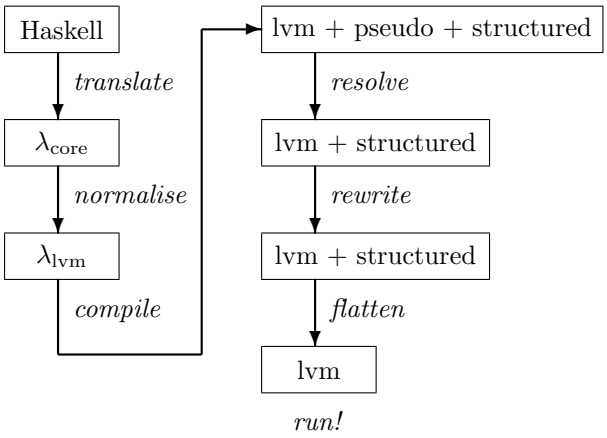

Figure 6.1: Compilation scheme

- As the instructions can be so closely related to an actual implementation, we can reason about implementation techniques that are normally only described informally, or explained via pictures. Examples are exception handling, returning constructors in registers, and the reason why `seq` frames are a necessary addition to the STG machine (Peyton Jones, 1992).

6.2 An overview

The compilation of a high level functional language to LVM instructions goes via a number of intermediate languages: λ_{core}, λ_{lvm}, and extended LVM instructions. The process is sketched in figure 6.1. The actual compilation steps involved are:

- Translate the source language to λ_{core}; an enriched lambda calculus that corresponds closely to the intermediate core language of the GHC compiler (Peyton Jones, 1992; Peyton Jones and Santos, 1998).
- *Normalise*: translate the λ_{core} language to the λ_{lvm} language, a more restricted form of λ_{core} that maps conveniently to LVM instructions.
- *Compile*: translate λ_{lvm} to LVM instructions that contain *pseudo* instructions. These pseudo instructions are used in the next phases to calculate correct stack and code offsets, but should be removed completely when generating the final instruction stream.
- *Resolve*: use and remove pseudo instructions that resolve stack offsets of local variables and arguments.

- *Rewrite*: optimize the instruction stream using rewrite rules.
- *Flatten*: remove all structure from the instructions and generate a flat stream of instructions that can efficiently be interpreted. This phase uses (and removes) the last pseudo instructions by calculating code offsets for jumps.
- *Execute*: execute the instruction streams according to the state transition rules.

The next sections describe all these steps in detail, but to give a feel of what each of these steps do, we describe each step in the context of a small example. We start with the following Haskell program:

$$
\begin{array}{l}
\mathit{main} \;=\; \mathit{const}\ \mathit{inc}\ (\lambda\, x \rightarrow x)\ 42 \\
\quad \textbf{where} \\
\quad\quad \mathit{inc}\ x \;=\; x \;+\; 1 \\
\quad\quad \mathit{const}\ x\ y \;=\; x
\end{array}
$$

The front-end language, in this case Haskell, is translated into an enriched lambda calculus, called λ_{core}. In this example, only the Haskell specific `where` binding is translated into a `let` binding but in general more transformation can be necessary; pattern matching compilation for example. Here is the translated program in λ_{core}:

$$
\begin{array}{l}
\mathit{main} \;=\; \textbf{let}\ \mathit{inc}\ x \;=\; x \;+\; 1 \\
\qquad\qquad\quad\ \ \mathit{const}\ x\ y \;=\; x \\
\qquad\qquad\ \ \textbf{in}\ \ \mathit{const}\ \mathit{inc}\ (\lambda\, x \rightarrow x)\ 42
\end{array}
$$

The *Normalise* step translates λ_{core} to a restricted form, called λ_{lvm}. The λ_{lvm} language is specifically designed to reflect the capabilities of the abstract machine and maps easily onto LVM instructions. In our example, all functions are lifted to top-level and binary application is translated into vector application:

$$
\begin{array}{lcl}
\mathit{id}\ x & = & x \\
\mathit{inc}\ x & = & x \;+\; 1 \\
\mathit{const}\ x\ y & = & x \\
\mathit{main} & = & \mathit{const}\ \mathit{inc}\ \mathit{id}\ 42
\end{array}
$$

Finally, the program is in a form where it can be compiled to LVM instructions, here are the instructions for the *id* function:

$$
\mathit{id} \;\mapsto\; \textbf{instr}(\mathsf{ArgChk}(1); \mathsf{Atom}(\mathsf{Param}(x); \mathsf{PushVar}(x); \mathsf{NewAp}(1)); \mathsf{Enter})
$$

However, the compiled program does not just consist of LVM instructions, but also contains *pseudo* instructions like Param and Atom. These instructions are used during the next phases to calculate stack and code offsets. The stack offsets are determined in the *resolve* step:

$$id \;\mapsto\; \textbf{instr}(\mathsf{ArgChk}(1); \mathsf{PushVar}(0); \mathsf{NewAp}(1); \mathsf{Slide}(1,1); \mathsf{Enter})$$

As we can see, the pseudo instructions Param and Atom have dissappeared, and variable references are replaced by a stack offset. After the stack offsets have been resolved, the instruction stream is rewritten according to simple rewrite rules. This step is strictly used to optimize the instruction stream. It is interesting to see that just few rewrite rules suffice to completely replace complicated translation schemes (Peyton Jones and Wadler, 1993; Johnsson, 1984; Plasmeijer and van Eekelen, 1993). The optimized instruction stream is:

$$id \;\mapsto\; \textbf{instr}(\mathsf{ArgChk}(1); \mathsf{Enter})$$

The final step flattens the instruction stream by calculating code offsets and it removes the remaining pseudo instructions. The complete code for our example becomes[1]:

$$\begin{array}{lll}
id & \mapsto & \textbf{instr}(\mathsf{ArgChk}(1); \mathsf{Enter}) \\
inc & \mapsto & \textbf{instr}(\mathsf{ArgChk}(1); \mathsf{PushInt}(1); \mathsf{PushVar}(1); \mathsf{AddInt}; \mathsf{Return}) \\
const & \mapsto & \textbf{instr}(\mathsf{ArgChk}(2); \mathsf{PushVar}(0); \mathsf{Slide}(1,2); \mathsf{Enter}) \\
main & \mapsto & \textbf{instr}(\mathsf{ArgChk}(0); \mathsf{PushInt}(42); \mathsf{PushCode}(id); \\
 & & \quad \mathsf{PushCode}(inc); \mathsf{EnterCode}(const))
\end{array}$$

The program can now be executed by the abstract LVM machine. The state of the machine is determined by the current code, the stack, and the heap. The initial state of the machine is always the Enter instruction with the value *main* on the stack. In our example, the initial heap *hp* just consists of the compiled functions. Note that we write $hp[p \mapsto x]$ if the heap *hp* contains a pointer p that points to value x. Here is a complete execution trace of our example:

	Code	Stack	Heap
$\Longrightarrow$	Enter	$[main]$	$hp[main \mapsto \textbf{instr}(...)]$
$\Longrightarrow$	ArgChk(0); PushInt(42); ...	$[main]$	hp
$\Longrightarrow$	PushInt(42); PushCode(id); ...	[]	hp
$\Longrightarrow$	PushCode(id); PushCode(inc); ...	$[42]$	hp
$\Longrightarrow$	PushCode(inc); EnterCode($const$)	$[id, 42]$	hp
$\Longrightarrow$	EnterCode($const$)	$[inc, id, 42]$	$hp[const \mapsto \textbf{instr}(...)]$
$\Longrightarrow$	PushVar(0); Slide(1, 2); ...	$[inc, id, 42]$	hp
$\Longrightarrow$	Slide(1, 2); Enter	$[inc, inc, id, 42]$	hp
$\Longrightarrow$	Enter	$[inc, 42]$	$hp[inc \mapsto \textbf{instr}(...)]$
$\Longrightarrow$	ArgChk(1); PushInt(1); ...	$[inc, 42]$	hp
$\Longrightarrow$	PushInt(1); AddInt; ...	$[42]$	hp
$\Longrightarrow$	AddInt; Return	$[1, 42]$	hp
$\Longrightarrow$	Return	$[43]$	hp
$\Longrightarrow$	terminate with an integer	$[43]$	hp

[1] Actually, some functions are compiled into more efficient code, but for clarity we use the unoptimized version.

The next section describes the abstract machine and instructions in detail. Section 6.4 describes the λ_{lvm} and λ_{core} languages. Section 6.5 describes the translation from λ_{lvm} to LVM instructions, together with the rewrite rules. The chapter closes with an assessment of a real implementation of the LVM and is followed by conclusions and two appendices that describe the format of binary LVM files.

6.3 The abstract machine

In this section we look at the operational semantics of the LVM instructions. The semantics of the LVM is given by state transitions (Plotkin, 1981). The state of the LVM is determined by a triple: the current instructions is, the stack st, and the heap hp.

The instruction sequence is consists of instructions and arguments of instructions. The empty sequence is written as [] and an initial instruction with arguments x and y, is written as $\mathsf{Instr}(x, y) : is$. Arguments and instructions have the same size and the previous expression is equivalent to $\mathsf{Instr} : x : y : is$.

The stack st is a sequence of values. The empty stack is written as [] and a non-empty stack with an initial value x as $x : st$. The n^{th} value on the stack is written as $st[n]$ where $st[0]$ is the top of the stack. Besides values, the stack also contains a chain of stack *markers*, each taking two stack slots. A marker is associated with the value next on the stack. A marker together with its value is called a *frame*. There exist three kinds of markers, update markers (**upd**), continuation markers (**cont**), and catch markers (**catch**).

The heap hp is a map from pointers p to heap values. We write $hp[p \mapsto x]$ if the heap hp contains a pointer p that points to value x. The extension of the heap with a fresh pointer p to value x is written as $hp \circ [p \mapsto x]$. The update of an existing pointer p with value x is written as $hp \bullet [p \mapsto x]$.

Heap values are tagged and have varying sizes. Although we give a short description, the exact meaning of each heap value becomes clear during the description of the instruction set. There exist six kinds of heap values:

$\mathbf{instr}(is)$	A sequence of instructions is.
$\mathbf{ap}(x_1, ..., x_n)$	An updateable application block.
$\mathbf{nap}(x_1, ..., x_n)$	A non-updateable application block.
$\mathbf{con}_t(x_1, ..., x_n)$	A constructor with tag t and arguments x_1 to x_n.
$\mathbf{inv}_n$	An invalid block of size n.
$\mathbf{raise}(x)$	An exception block, raises exception x when entered.

The *initial heap* contains all global values. All instruction arguments that refer to a global value are fixed by the runtime loader to contain the proper heap pointer. The special value inv will point to an invalid block of size 0: $inv \mapsto \mathbf{inv}_0$

The initial state of the abstract machine consists of: the Enter instruction, a stack that just contains (a pointer to) the function *main*, and the initial heap.

Since the state of the abstract machine is so simple, it maps directly onto current hardware. Instruction streams can be modelled with a simple instruction pointer

as instructions are never modified. The stack can be modelled with an array and a stack pointer. Only the heap requires extensive runtime support, but that seems unavoidable in any garbage collected language.

6.3.1 Basic instructions

We first introduce a minimal set of instructions that support a minimal subset of λ_{lvm}. New instructions are introduced as the need arises by taking new language features into account. We first treat a minimal useful subset of the λ_{lvm} language, just consisting of top-level values with (partial) function applications.

All `let`-bound local variables and function parameters reside on the stack. Three instructions manipulate the stack: PushVar pushes a copy of a value that resides on the stack (i.e. a local variable or parameter), PushCode pushes a pointer to a top-level value, and Slide slides out unused values.

	Code	Stack	Heap
	PushCode(f) : is	st	hp
$\Longrightarrow$	is	$f : st$	hp
	PushVar(ofs) : is	st	hp
$\Longrightarrow$	is	$st[ofs] : st$	hp
	Slide(n, m) : is	$x_1 : ... : x_n : ... : x_{n+m} : st$	hp
$\Longrightarrow$	is	$x_1 : ... : x_n : st$	hp

The parameters of a function are pushed on the stack in a right-to-left order. This is dual to most imperative languages that use left-to-right order, like Java and ML. The most notable exception is the C language that uses a right-to-left calling convention in order to support functions with a variable number of arguments. However, for any higher-order language that allows partial applications, it is necessary to use this calling convention. The following example illustrates why partial applications force a right-to-left order.

$$
\begin{aligned}
&\mathit{id}\ x &&= x\\
&\mathit{const}\ x\ y &&= x\\
&\mathit{apply}\ f\ x &&= f\ x\\
&\mathit{main} &&= \mathit{apply}\ (\mathit{const}\ \mathit{id})\ \mathit{const}
\end{aligned}
$$

With a right-to-left order, everything works well – inside *apply*, the argument x is pushed (which is *const*) and f is entered. This is actually the expression *const id* that pushes *id* and enters *const* with a proper stack: $\mathit{id} : \mathit{const} : []$, where parameter x is *id* and parameter y is *const*. If a left-to-right order is used, the partial application *const id* somehow has to insert its argument between arguments already residing on the stack. This can not be done without whole-program analysis and might even be impossible to do in general.

Partial applications combined with polymorphism also lead to the famous *argument check*. In a higher-order, polymorphic language it is not always possible to determine at a call site if a function is partially applied or not. In the previous example, this is the case for the parameter f in the *apply* function. For this reason, each function checks the number of arguments itself with the ArgChk instruction, which is always the first instruction of a top-level value. If there are enough arguments on the stack, execution continues. If there are not enough arguments on the stack, we immediately return with a functional value as a result.

	Code	Stack	Heap
$n \leq m$	$\mathsf{ArgChk}(n) : is$	$f : x_1 : ... : x_m : st$	hp
$\Longrightarrow$	is	$x_1 : ... : x_m : st$	hp
(1) $n > m$	$\mathsf{ArgChk}(n) : is$	$f : x_1 : ... : x_m : []$	hp
$\Longrightarrow$	$[]$	$f : x_1 : ... : x_m : []$	hp

(1) termination with a functional value.

Due to polymorphism, it is not always possible to determine at a call site which particular function is called. Therefore, the Enter instruction is able to enter any kind of value that resides on top of the stack. Right now, we only have instruction values but we will later add more values that can be entered.

	Code	Stack	Heap
	$\mathsf{Enter} : is$	$f : st$	$hp[f \mapsto \mathbf{instr}(is_f)]$
$\Longrightarrow$	is_f	$f : st$	hp

Note that we *enter* a function instead of *calling* it. Every function application in λ_{lvm} is a *tail* call and, since we don't need to return, there is no need to push a return address either. It is necessary however to remove any local variables and parameters of the calling function that are still on the stack with the Slide instruction. Besides keeping the stack from growing, it is essential for our definition of the ArgChk instruction – if the local variables or parameters are not squeezed out, they are misinterpreted by the argument check as if they are extra parameters! This subtle requirement was first observed by Mountjoy in the context of the STG machine (Mountjoy, 1998).

Here are some examples of functions that can be compiled with the current instruction set:

$$\begin{aligned} id\ x &= x \\ swap\ x\ f &= f\ x \\ main &= swap\ id\ id \end{aligned}$$

The final value of this program is the functional value id. With the compilation scheme from section 6.5 we get the following initial heap:

$$id \mapsto \mathbf{instr}(\mathsf{ArgChk}(1); \mathsf{Enter})$$

$$
\begin{array}{rcl}
\mathit{swap} & \mapsto & \mathbf{instr}(\mathsf{ArgChk}(2); \mathsf{PushVar}(0); \mathsf{PushVar}(2); \mathsf{Slide}(2,2); \mathsf{Enter}) \\
\mathit{main} & \mapsto & \mathbf{instr}(\mathsf{ArgChk}(0); \mathsf{PushCode}(\mathit{id}); \mathsf{PushCode}(\mathit{id}); \\
 & & \quad \mathsf{PushCode}(\mathit{swap}); \mathsf{Enter})
\end{array}
$$

In the above program, it is clear that the *swap* function is called with sufficient arguments. This special case can be optimized with the EnterCode instruction. If a known function is called with sufficient arguments, the argument check of the called function can be skipped. This is called the 'direct entry point' convention in the STG machine (Peyton Jones, 1992). The EnterCode instruction performs this optimization and enters a known function with sufficient arguments.

	Code	Stack	Heap
	$\mathsf{EnterCode}(f) : \mathit{is}$	st	$\mathit{hp}[f \mapsto \mathbf{instr}(\mathsf{ArgChk}(n) : \mathit{is}_f)]$
$\Longrightarrow$	is_f	st	hp

This instruction is essentially what a C compiler would use to implement tail calls: a jump! In contrast, the Enter instruction performs an indirect jump based on the kind of value that is entered – object oriented people would probably call this a 'virtual method tail-call'.

6.3.2 Local definitions

In this section we extend the instruction set to deal with local `let` and `letrec` bindings. A `let` binding is non-strict and delays evaluation of its right-hand side. The NewNap instruction allocates a (non-updateable) application node in the heap that contains the function to be called and its arguments.

	Code	Stack	Heap
	$\mathsf{NewNap}(n) : \mathit{is}$	$x_1 : \ldots : x_n : \mathit{st}$	hp
$\Longrightarrow$	is	$p : \mathit{st}$	$\mathit{hp} \circ [p \mapsto \mathbf{nap}(x_1, \ldots, x_n)]$

Now that we have introduced a new heap value, we need to extend the Enter instruction to deal with this new value. When the Enter instruction sees a (non-updateable) application node, the values are moved to the stack, and the top of the stack is entered again.

	Code	Stack	Heap
	$\mathsf{Enter} : \mathit{is}$	$p : \mathit{st}$	$\mathit{hp}[p \mapsto \mathbf{nap}(x_1, \ldots, x_n)]$
$\Longrightarrow$	$\mathsf{Enter} : \mathit{is}$	$x_1 : \ldots : x_n : \mathit{st}$	hp

6.3.3 Sharing

Although the NewNap instruction delays the evaluation of an expression, it is not lazy since it doesn't *share* the result. Take for example the following program:

$$main \;=\; \texttt{let}\ x = \mathit{nfib}\ 10\ \texttt{in}\ x + x$$

The expression *nfib* 10 is calculated twice if the `let` binding uses the NewNap instruction. To share the computation, we use graph reduction instead of simple tree reduction. The NewAp instruction allocates an *updateable* application node in the heap. When this node is evaluated it is *updated* with its evaluated value, thus sharing the result of the computation. The Enter instruction puts a special update marker on the stack as a reminder that the node has to be updated with its evaluated value. The ArgChk instruction looks for these update frames – if there are insufficient arguments, the updateable application node is overwritten with a non-updateable one. At the moment, the only weak-head-normal-form values are functional values, but in the following sections we will see how updateable application nodes can also be overwritten by integers for example.

	Code	Stack	Heap
	NewAp$(n) : is$	$x_1 : ... : x_n : st$	hp
$\Longrightarrow$	is	$p : st$	$hp \circ [p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
	Enter $: is$	$p : st$	$hp[p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
$\Longrightarrow$	Enter $: is$	$x_1 : ... : x_n : \mathbf{upd} : p : st$	hp
$n > m$	ArgChk$(n) : is$	$f : x_1 : ... : x_m : \mathbf{upd} : p : st$	hp
$\Longrightarrow$	ArgChk$(n) : is$	$f : x_1 : ... : x_m : st$	$hp \bullet [p \mapsto \mathbf{nap}(f, x_1, ..., x_m)]$

The argument check instruction suddenly looks expensive: previously, the number of arguments on the stack was equal to the depth of the stack, but now it seems the argument check has to search the stack for an update marker to determine the number of arguments! Fortunately, we can use some conventional compiler technology to overcome this inefficiency.

An implementation uses a frame pointer *fp* that points to the top frame on the stack. Now we also see why a marker takes up two stack slots: one slot is the real marker while the second is a link back to the previous stack frame. When a frame is pushed, the current frame pointer is saved in the marker and the frame pointer is updated to point to the new top frame. When a frame is popped, the frame pointer is updated with the back-link. The argument check can now substract the frame pointer from the stack pointer to obtain the number of arguments on the stack.

Not only local values should be shared but top-level values that take no arguments should be shared too. These values are called *constant applicative forms* or caf's. The initial heap contains an **ap** node for each caf. In the previous example, *main* takes no arguments and its initial heap nodes are:

$$\begin{array}{rl} main & \mapsto \mathbf{ap}(main') \\ main' & \mapsto \mathbf{instr}(\mathsf{ArgChk}(0); ...) \end{array}$$

Note that we can't generate an EnterCode instruction for function calls with zero arguments: their values point to either an **ap** node, or to any other value with which they are updated! We can certainly not jump to their instructions directly.

6.3.4 Recursive values

Recursive values are constructed in two steps: dummy values are allocated first, and later initialized. This allows the values to refer to each other. The AllocAp instruction allocates an application node without initializing its fields. Later the Pack(N)Ap instruction initializes the fields.

	Code	Stack	Heap
	$\mathsf{AllocAp}(n) : is$	st	hp
$\Longrightarrow$	is	$p : st$	$hp \circ [p \mapsto \mathbf{inv}_n]$
$p = st[ofs - n]$	$\mathsf{PackAp}(ofs, n) : is$	$x_1 : ... : x_n : st$	hp
$\Longrightarrow$	is	st	$hp \bullet [p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
$p = st[ofs - n]$	$\mathsf{PackNap}(ofs, n) : is$	$x_1 : ... : x_n : st$	hp
$\Longrightarrow$	is	st	$hp \bullet [p \mapsto \mathbf{nap}(x_1, ..., x_n)]$

6.3.5 Algebraic data types

The LVM supports open ended algebraic data types. Constructor blocks are allocated just like application blocks. The AllocCon and PackCon are used to construct recursive constructor definitions.

	Code	Stack	Heap
	$\mathsf{NewCon}(t, n) : is$	$x_1 : ... : x_n : st$	hp
$\Longrightarrow$	is	$p : st$	$hp \circ [p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
	$\mathsf{AllocCon}(t, n) : is$	st	hp
$\Longrightarrow$	is	$p : st$	$hp \circ [p \mapsto \mathbf{con}_t(...)]$
$p = st[ofs - n]$	$\mathsf{PackCon}(ofs, n) : is$	$x_1 : ... : x_n : st$	$hp[p \mapsto \mathbf{con}_t(...)]$
$\Longrightarrow$	is	st	$hp \bullet [p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$

When the Enter instruction sees a constructor value, it behaves like the Return instruction:

	Code	Stack	Heap
	Enter : *is*	p : *st*	$hp[p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
$\Longrightarrow$	Return : *is*	p : *st*	hp
	Return : *is*	p : **upd** : u : *st*	$hp[p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
$\Longrightarrow$	Return : *is*	p : *st*	$hp \bullet [u \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
(1)	Return : *is*	p : []	hp
$\Longrightarrow$	[]	p : []	hp

(1) Termination with a constructor value.

The Return instruction is used when the final value is known to be a constructor. Just like the ArgChk instruction, the Return instruction looks for frames on the stack where an update frame causes the value to be updated with the constructor value. When the stack is empty, execution stops with the constructor value as the result.

Note that we have two instructions that look at the stack configuration, ArgChk and Return, and one instruction that looks at the type of an heap value, Enter.

6.3.6 Strict evaluation

Before describing how values of algebraic data types are matched, we first look at their evaluation. The `let!` binding strictly evaluates its right-hand side before evaluating the body. A *continuation* marker is pushed on the stack before the evaluation of the right-hand side. When the right-hand side is evaluated, execution resumes at the instructions in the continuation frame.

	Code	Stack	Heap
	PushCont(n) : *is*	*st*	*hp*
$\Longrightarrow$	*is*	**cont** : drop n *is* : *st*	*hp*
	Return : *is*	p : **cont** : is' : *st*	*hp*
$\Longrightarrow$	is'	p : *st*	*hp*
$n > m$	ArgChk(n) : *is*	f : x_1 : ... : x_m : **cont** : is' : *st*	*hp*
$\Longrightarrow$	is'	p : *st*	$hp \circ [p \mapsto \mathbf{nap}(f, x_1, ..., x_m)]$

Continuation frames resemble conventional calling conventions closely – a C compiler pushes a return address before calling a function. The STG machine (Peyton Jones, 1992) also uses plain return adresses instead of continuation frames. This seems impossible at first sight – The argument check builds a partial application block if there are insufficient arguments, which is checked by looking at the top frame. If only a plain return address is pushed instead of a frame, the number of arguments can't be determined as the return address is misinterpreted as just another argument! However, the STG machine only evaluates expressions that are scrutinized by a `case` expression. These expressions can never have a functional type, and the STG machine therefore never reaches this machine configuration – quite a subtly dependency on the host language. Indeed, the STG machine has spe-

cial **seq** frames to support the polymorphic *seq* function of Haskell that *can* take functional values as its argument. The **seq** frames are treated like our continuation frames. As we always use explicit continuation frames, the *seq* function can be expressed directly in the λ_{lvm} language: $seq\ x\ y = \texttt{let!}\ z = x\ \texttt{in}\ y$

6.3.7 Matching

Once a value is evaluated to weak head normal form, it can be matched. The MatchCon instruction matches on algebraic datatypes.

	Code	Stack	Heap
$\exists i.\ t = t_i$	$\mathsf{MatchCon}(n, o, t_1, o_1, ..., t_n, o_n) : is$	$p : st$	$hp[p \mapsto \mathbf{con}_t(x_1, ..., x_m)]$
$\Longrightarrow$	drop $o_i\ is$	$x_1 : ... : x_m : st$	hp
$\forall i.\ t \neq t_i$	$\mathsf{MatchCon}(n, o, t_1, o_1, ..., t_n, o_n) : is$	$p : st$	$hp[p \mapsto \mathbf{con}_t(x_1, ..., x_m)]$
$\Longrightarrow$	drop $o\ is$	$p : st$	hp

The MatchCon instruction pops the argument p when a constructor matches. This opens up the possibility of an important optimization. Many constructors are allocated in the heap and immediately deconstructed with a match. The ReturnCon instruction tries to avoid many of these allocations. ReturnCon behaves denotationally exactly like a NewCon followed by a Return:

$$\mathsf{ReturnCon}(t, n)\ \Rightarrow\ \mathsf{NewCon}(t, n); \mathsf{Return}$$

However, there exist a more efficient implementation that sometimes avoids an expensive heap allocation. This is called the 'return in registers' convention in the STG machine (Peyton Jones, 1992).

	Code	Stack	Heap
(1)	$\mathsf{ReturnCon}(t, n) : is$	$x_1 : ... : x_n : \mathbf{cont} : is' : st$	hp
$\Longrightarrow$	drop $o_i\ is''$	$x_1 : ... : x_n : st$	hp
	$\mathsf{ReturnCon}(t, n) : is$	st	hp
$\Longrightarrow$	$\mathsf{NewCon}(t, n) : \mathsf{Return} : is$	$x_1 : ... : x_n : st$	hp

(1) $is' = \mathsf{MatchCon}(n, o, t_1, o_1, ..., t_n, o_n) : is'' \wedge \exists\, i.\ t = t_i$

In the special but common case that a constructor returns immediately into a MatchCon instruction, the ReturnCon instruction avoids the allocation of the constructor in the heap. In all other cases, it behaves like a NewCon/Return pair. This happens for example when there is an update frame before the continuation or when the constructor is not immediately matched after being evaluated. The 'return in register' convention is no longer used in GHC as it lead to too much complexity in the generated code. For the LVM this doesn't seem the case and the LVM interpreter uses this optimization on every constructor that is returned.

6.3.8 Synchronous exceptions

Any robust programming language needs to handle exceptional situations. The LVM instruction set supports exception handling at a fundamental level for two reasons. The first reason is efficiency – since exceptional situations are exceptional, normal execution shouldn't be penalized. Furthermore, LVM instructions, like division, can raise exceptions themselves and thus, the LVM needs a standard mechanism for raising exceptions.

The Catch instruction installs an exception handler. The instruction pushes a **catch** frame on the stack. When an exception is raised, execution is continued at the exception handler. When no exception is raised, the **catch** frame is simply ignored by other instructions that look for stack frames, i.e. ArgChk and Return.

	Code	Stack	Heap
	Catch : *is*	$h : st$	*hp*
$\Longrightarrow$	*is*	**catch** : $h : st$	*hp*
$n > m$	ArgChk(n) : *is*	$x_1 : ... : x_m$: **catch** : $h : st$	*hp*
$\Longrightarrow$	ArgChk(n) : *is*	$x_1 : ... : x_m : st$	*hp*
	Return : *is*	x : **catch** : $h : st$	*hp*
$\Longrightarrow$	Return : *is*	$x : st$	*hp*

Note that a **catch** frame should immediately follow another frame or the end of the stack. If this is not the case, the Return instruction could end up in an undefined configuration. In practice, an implementation can actually deal quite easily with **catch** frames that don't follow another frame directly. When the Return instruction pops the **catch** frame, it also pops any values up to the next frame on the stack.

An exception is raised explicitly with the Raise instruction. It unwinds the stack until it finds a **catch** frame. Execution is continued at the exception handler with the exception as its argument.

	Code	Stack	Heap
	Raise : *is*	x : **catch** : $h : st$	*hp*
$\Longrightarrow$	Enter : *is*	$h : x : st$	*hp*
(1)	Raise : *is*	x : []	*hp*
$\Longrightarrow$	[]	x : []	*hp*
	Raise : *is*	x : **upd** : $p : st$	*hp*
$\Longrightarrow$	Raise : *is*	$x : st$	$hp \bullet [p \mapsto \textbf{raise}(x)]$
	Raise : *is*	x : **cont** : $is' : st$	*hp*
$\Longrightarrow$	Raise : *is*	$x : st$	*hp*
	Raise : *is*	$x : y : st$	*hp*
$\Longrightarrow$	Raise : *is*	$x : st$	*hp*

(1) Termination with an exceptional value.

Again, we assume that there is another frame immediately following the Catch frame. Otherwise, the Raise instruction has to pop any values following the Catch frame to prevent that they are treated as extra arguments by the Enter instruction. Note that having explicit continuation frames helps out in practice to find more information about the exception at runtime. In our LVM implementation we have special functions that inspect the stack when an exception occurs – the update and continuation markers give the execution trace that lead to the exception. This proved especially useful in the Helium compiler that is used mainly for educational purposes where good error messages are highly important.

When the Raise instruction encounters an update frame it updates the value with a **raise** block – indeed, if a value raises an exception once, it will always raise that exception when evaluated, and should be updated with that exception. When a **raise** block is entered, it raises the exception again.

	Code	Stack	Heap
	Enter : is	$p : st$	$hp[p \mapsto \mathbf{raise}(x)]$
$\Longrightarrow$	Raise : is	$x : st$	hp

An example of an exceptional situation is a stack overflow. In the LVM the ArgChk instruction conservatively checks for thousand available stack slots. If there are fewer, a stack overflow exception is raised.

	Code	Stack	Heap
$free(st) < 1000$	ArgChk(n) : is	st	hp
$\Longrightarrow$	Raise : is	$p : st$	$hp \circ [p \mapsto stackoverflow]$

The *stackoverflow* heap block is just a constructor of the predefined *Exception* data type and contains for example the source and line number of the function where the stack overflowed. Together with the execution trace, this normally pins down an unbounded recursion.

6.3.9 Blackholing

Certain forms of infinite loops can be detected at runtime. In particular, if we enter an updateable application node, we should not re-enter that node again during the update. To prevent this kind of infinite loop, we can overwrite an application node with a **raise** node when we enter it. When the value is finally updated, the **raise** node is overwritten again with the computed value. Whenever the value is re-entered during the update, a *black hole* exception is raised automatically. Here is the refined Enter rule for application nodes.

	Code	Stack	Heap
	Enter : *is*	$p : st$	$hp[p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
$\Longrightarrow$	Enter : *is*	$x_1 : ... : x_n : \mathbf{upd} : p : st$	$hp \bullet [p \mapsto \mathbf{raise}(blackhole)]$

However, it is fairly expensive to overwrite all application nodes whenever they are entered. A more efficient technique is to delay the overwrite, called lazy blackholing. With this technique, execution is sometimes stopped to do lazy blackholing, where every value in an update frame on the stack is overwritten with a blackhole. Since this kind of infinite loop always grows the stack, a good moment to do this is when the stack needs to be extended, but it can also be done during garbage collection or when a thread yields. We can describe this technique formally by a generic rule that allows us to execute a Blackhole instruction at any time. This instruction saves the current stack pointer and then walks the stack, updating any update frames with a blackhole.

	Code	Stack	Heap
	is	*st*	*hp*
$\Longrightarrow$	Blackhole(st) : *is*	*st*	*hp*
	Blackhole(st) : *is*	[]	*hp*
$\Longrightarrow$	*is*	*st*	*hp*
	Blackhole(st') : *is*	**upd** : p : *st*	*hp*
$\Longrightarrow$	Blackhole(st') : *is*	*st*	$hp \bullet [p \mapsto \mathbf{raise}(blackhole)]$
	Blackhole(st') : *is*	_ : *st*	*hp*
$\Longrightarrow$	Blackhole(st') : *is*	*st*	*hp*

6.3.10 Garbage collection

Another generic rule that can be applied at any time is the garbage collection rule. This rule models a garbage collector as part of the abstract machine.

	Code	Stack	Heap
(1)	*is*	*st*	*hp*
$\Longrightarrow$	*is*	*st*	hp'

(1) Where hp' is constrained to the *reachable* pointers:
$hp' = [\, p \mapsto x \mid p \mapsto x \in hp \ \wedge \ p \in reachable(st, hp) \,]$

The *reachable(st,hp)* predicate returns all pointers that can be reached from the stack *st* in the heap *hp*. Note that the reachable set includes all pointers can potentially be used later, and it is a superset of the *live* pointers that encompasses the set the pointers that are actually used later on. As such it is a conservative estimate of liveness.

However, we should always try to keep the reachable set as small as possible to

avoid space leaks. For one thing, we can see that it is actually a good strategy to perform lazy blackholing just before a garbage collection as it makes the reachable set potentially smaller. This was first observed by Jones (Jones, 1994). Other techniques are stack stubbing where we overwrite values in the stack with dummy values if we can statically determine that the values are never referenced again. This happens often in alternatives of a `match` statement:

	Code	Stack	Heap
	$\mathsf{Stub}(n) : is$	$x_0 : ... : x_{n-1} : st$	hp
$\Longrightarrow$	is	$x_0 : ... : inv : st$	hp

The garbage collection rule is also essential to prove the correctness of the rewrite rules presented in the next section. As an illustrative example, we show that the important rule for avoiding allocation of application nodes is correct:

$$\mathsf{NewAp}(n); \mathsf{Slide}(1, m); \mathsf{Enter} \Rightarrow \mathsf{Slide}(n, m); \mathsf{Enter}$$

We can prove that this transformation is correct by showing that it leads to the same machine configuration at runtime. Here, we need the garbage collection rule to discard the intermediate application node.

	Code	Stack	Heap
	$\mathsf{NewAp}(n); \mathsf{Slide}(1, m); \mathsf{Enter}$	$x_1 : ... : x_n : ... : x_{n+m} : st$	hp
$\Longrightarrow$	$\mathsf{Slide}(1, m); \mathsf{Enter}$	$p : ... : x_{n+m} : st$	$hp \circ [p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
$\Longrightarrow$	Enter	$p : st$	$hp \circ [p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
$\Longrightarrow$	Enter	$x_1 : ... : x_n : st$	$hp \circ [p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
	{garbage collection}		
$\Longrightarrow$	Enter	$x_1 : ... : x_n : st$	hp
$\Longleftarrow$	$\mathsf{Slide}(n, m); \mathsf{Enter}$	$x_1 : ... : x_n : ... : x_{n+m} : st$	hp

6.4 The LVM language

Now that we have described the instruction set in detail, it is time to look more closely at how we can map a functional language onto these instruction. We define a low level language called λ_{lvm}, that closely relates to the LVM instructions. The abstract syntax for λ_{lvm} is given in figure 6.3.10. Allthough the form of expressions is restrictive, any enriched lambda calculus expression can be translated into a λ_{lvm} expression.

Just like the STG language (Peyton Jones, 1992), we attach an operational reading to λ_{lvm}: `let` and `letrec` bind expressions to variables, `let!` evaluates expressions to weak head normal form and `match` distinguishes evaluated values.

The λ_{lvm} language does not contain lambda-expressions or local function definitions – we assume that all functions have been lambda-lifted to toplevel (Johnsson, 1985). This means that λ_{lvm} functions contain no free variables and a program consists just of a set of combinator definitions.

The `let!` expression is a strict version of `let`. It evaluates its right hand side to weak-head-normal-form before evaluating the body of the expression. The usual `case` expression of lazy languages is easily translated into a `let!` and `match` pair:

`case` *e* `of` *alts*
⇒
`let!` *x* `=` *e* `in` `match` *x* `with` *alts*

The `let!` binding is also used to translate strict languages to λ_{lvm}. The `letrec` binding of O'Caml and ML can only be used with recursive *functions*, which are lifted to toplevel, and present no problem.

Primitive expressions

Primitive expressions are functions that are not expressed in λ_{lvm} itself. They may consist of instructions, like integer addition, statically linked functionality like the `sin` function, or user imported foreign functions. All these variants are accomodated with the *prim* expression. This has proved very convenient in the implementation of the compiler, as it can treat these expressions uniformly up to code generation time.

Separate primitive declarations describe the different kinds. The syntax is exactly the same as foreign import declarations – here are some examples:

```
foreign instruction  "AddInt" addInt :: Int -> Int -> Int
foreign import       "sin"    sin    :: Double -> Double
```

Program	*program*	→	$\{$*top* $\{$*var*$\}^*$ `=` *expr*`;`$\}^*$
Expression	*expr*	→	`let!` *var* `=` *expr* `in` *expr*
		|	`match` *var* `with` `{` $\{$*pat* `->` *expr*`;`$\}^+$ `}`
		|	$prim^n$ $\{atom\}^n$
		|	*let* `in` *expr*
		|	*atom*
Let	*let*	→	`letrec` `{` $\{$*var* `=` *atom*`;`$\}^+$ `}`
		|	`let` *var* `=` *atom*
Atomic	*atom*	→	*let* `in` *atom*
		|	*id* $\{atom\}^*$
		|	con_t^n $\{atom\}^n$
		|	*literal*
Pattern	*pat*	→	*var*
		|	con_t^n $\{var\}^n$
		|	*literal*
Literal	*literal*	→	*int* | *float* | *bytes*
Identifier	*id*	→	*var* | *top*
Variable	*var*	→	local identifier (x)
Global	*top*	→	top level identifier (f)
Constructor	con_t^n	→	constructor with tag t and arity n
Primitive	$prim^n$	→	instruction or foreign function of arity n
Integer	*int*	→	integer (i)
Float	*float*	→	floating point number
Bytes	*bytes*	→	a sequence of bytes (packed string)
Notation	$\{p\}^*$	→	zero or more p
	$\{p\}^+$	→	one or more p
	$\{p\}^n$	→	exactly n occurrences of p

Figure 6.2: Abstract syntax of the λ_{lvm} language

Note that instructions are also described as foreign functions – they just have an extremely efficient calling convention and encoding!

Atomic expressions

The distinction between atomic and normal expressions is more than a syntactic convenience. During execution, the instructions that are generated for an atomic expression will always succeed and terminate[2]. In contrast, a `let!`, `match` or *prim* expression can raise an exception or go into an infinite loop. This is the reason why `let` expressions can only contain atomic expressions on their right-hand side. In contrast with the STG language, `let` expressions can contain nested `let` expressions on their right-hand side (as they are also atomic).

The STG paper (Peyton Jones, 1992) recommends special compilation techniques to avoid the creation of nested `let` bindings. With the LVM language, this can be avoided as nested `let` bindings can be compiled directly. Consider the following Haskell expression:

$$f\ =\ \texttt{let}\ x\ =\ [1,2]\ \texttt{in}\ e$$

This is translated to:

$$f\ =\ \texttt{let}\ x\ =\ (\texttt{let}\ y\ =\ \mathit{Cons}\ 2\ \mathit{Nil}\ \texttt{in}\ \mathit{Cons}\ 1\ y)\ \texttt{in}\ e$$

The STG machine can not deal with nested `let` bindings and will implicitly lift it to top-level, as in:

$$f_y\ =\ \mathit{Cons}\ 2\ \mathit{Nil}$$
$$f\ =\ \texttt{let}\ x\ =\ \mathit{Cons}\ 1\ f_y\ \texttt{in}\ e$$

However, it is hard to garbage collect a top level binding without arguments and it is not recommended to lift bindings to top level in general (Peyton Jones *et al.*, 1996). The translation recommended in the STG paper therefore, is to float the `let` binding one level up:

$$f\ =\ \texttt{let}\ y\ =\ \mathit{Cons}\ 2\ \mathit{Nil}\ \texttt{in let}\ x\ =\ \mathit{Cons}\ 1\ y\ \texttt{in}\ e$$

Both programs are denotationally equivalent but operationally different. Under the compilation scheme presented in the next section, the first program slides out the y value from the stack and therefore uses slightly less stack space with slightly more work. In contrast to the STG machine, both programs will construct the *Cons* 2 *Nil* node, even when x is never demanded. Since everything in the LVM language is explicit and has a formal operational reading, we are able to explicitly express all three variants and reason about their operational behaviour.

[2]Modulo fatal situations like heap exhaustion.

Strictness and speculative evaluation

In general, we can not float other constructs like `let!` or `match` since they might fail or perform an unbounded amount of computation. It is possible when a strictness analyser determines that the value is demanded later, but in that case it is easier to transform the `let` binding into a `let!` binding, which *can* have full expressions at its right-hand side.

If the strictness analyser can not prove that a value is demanded but if we are reasonably sure that the expression uses a bounded amount of computation, we could *speculatively* evaluate the expression. The value is computed eagerly but if it fails or uses too much resources, in terms of time or space, it is suspended. Currently, this is still an area of research (Peyton Jones and Ennals, 2003) but we plan to add the atomic `let$` construct for speculative bindings. Again, the operational semantics described later in this chapter allows us to reason very specifically about the operational behaviour of eager evaluation.

6.4.1 Translating λ_{core} to λ_{lvm}

It is convenient to use an intermediate language that is less restrictive than λ_{lvm} in a compiler. We define λ_{core} as an enriched lambda calculus with lambda expressions, free variables, no distinction between atomic expresions and normal expression, binary application, and unsaturated constructors and primitives.

The λ_{core} language can be mapped to the λ_{lvm} by applying the following transformations:

- Replace binary application with vector application.

  $(... ((id\ e_1)\ e_2) ...)\ e_n \quad \Rightarrow \quad id\ e_1 ...\ e_n$

- Saturate all applications to constructors and primitives.

  $con_t^n\ e_1 ...\ e_m \quad \mid (m < n) \Rightarrow \quad \backslash x_{(m+1)} ...\ x_n\ .\ con_t^n\ e_1 ...\ e_m\ x_{(m+1)} ...\ x_n$

- Introduce a `let` expression for all anonymous lambda expressions.

  $\backslash x_1 ...\ x_n\ .\ e \quad \Rightarrow \quad \texttt{let}\ x\ x_1 ...\ x_n\ =\ e\ \texttt{in}\ x$

- Introduce a `let` expression for all non-atomic arguments.

  $e\ (\texttt{match}\ x\ \texttt{with}\ alts) \quad \Rightarrow \quad \texttt{let}\ y\ =\ (\texttt{match}\ x\ \texttt{with}\ alts)\ \texttt{in}\ e\ y$

- Introduce a `let` expression for all applications to terms that are not variables or constructors:

  $$e\ x_1\ ...\ x_n \quad \Rightarrow \quad \texttt{let}\ x\ =\ e\ \texttt{in}\ x\ x_1\ ...\ x_n$$

- Pass all free variables in non-atomic expressions as explicit arguments. This corresponds essentially to lambda-lifting (Johnsson, 1985; Peyton Jones and Lester, 1991; Hughes, 1984), and leads to an environment-less machine.

  $$f\ x\ =\ \texttt{let}\ y\ =\ (\texttt{let!}\ z\ =\ 1/x\ \texttt{in}\ z)\ \texttt{in}\ y$$
  $$\Rightarrow$$
  $$f\ x\ =\ \texttt{let}\ y\ x\ =\ (\texttt{let!}\ z\ =\ 1/x\ \texttt{in}\ z)\ \texttt{in}\ y\ x$$

- Lift all local functions and non-atomic right-hand sides of `let` bindings to top-level.

  $$f\ x\ =\ \texttt{let}\ y\ x\ =\ (\texttt{let!}\ z\ =\ 1/x\ \texttt{in}\ z)\ \texttt{in}\ y\ x$$
  $$\Rightarrow$$
  $$f_y\ x\ =\ \texttt{let!}\ z\ =\ 1/x\ \texttt{in}\ z$$
  $$f\ x\ =\ f_y\ x$$

6.5 Compilation scheme

The compilation scheme translates λ_{lvm} into LVM instructions. In order to make the translation as clear as possible, the compilation scheme uses a few pseudo instructions to delay offset computations. This allows us to move the complexity of computing stack offsets of local variables to a separate *resolve* phase. The pseudo instructions are:

- $\mathsf{Param}(x)$ declares a local variable x that resides on the stack as an argument. This instruction allows the *resolve* phase to calculate the correct stack offset for x.
- $\mathsf{Var}(x)$ declares a local variable x that is bound to the current top of the stack.
- $\mathsf{Eval}(is)$. After executing instructions is, execution is continued at the next instruction. It is translated during code generation into $(\mathsf{PushCont}(ofs); is)$. Eval is introduced to delay the computation of the code offset *ofs* which is only known at code generation time.
- $\mathsf{Atom}(is)$. This instruction is used for translating expressions that result in a single value on the stack. During *resolve* it is translated into the instructions $(is; \mathsf{Slide}(1, m))$ where m intermediate values are slided out of the stack. Atom is used to delay the computation of the correct value for m which is only known during the *resolve* phase.
- $\mathsf{Init}(is)$. This instruction is used for translating the initialization of `letrec` bindings. The instructions *is* don't compute any value on the stack. During *resolve* it is translated into the instructions $(is; \mathsf{Slide}(0, m))$ where m intermediate values are slided out of the stack.

6.5.1 Program

A LVM program is translated with the $\mathcal{P}$ scheme.

$$\begin{aligned}&\mathcal{P}[\![\, f_1\ args_1 = e_1\ ;\ \ldots ;\ f_n\ args_n = e_n\ ;\,]\!] \quad \Rightarrow\\ &\quad \textbf{let}\ index(f_i) = i\\ &\quad \textbf{let}\ arity(f_i) = |args_i|\\ &\quad \textbf{let}\ code(f_i) = \mathcal{T}[\![\, args_i = e_i\,]\!]\end{aligned}$$

The $\mathcal{P}$ scheme translates a program into three functions, *code* gives the code for a function, *arity* returns the number of parameters and *index* returns the index used in binary LVM files.

Each top level value is translated with the $\mathcal{T}$ scheme. The $\mathcal{T}$ scheme emits the pseudo instruction Param for each argument in order to resolve the stack offsets of

each argument during the *resolve* phase. As signified by the Atom instruction, a single value is computed on top of the stack that is subsequently entered by the Enter instruction.

$$\mathcal{T}[\![\, x_1 \ldots x_n = e \,]\!] \quad \Rightarrow$$
$$\quad \mathsf{ArgChk}(n);\ \mathsf{Atom}(\mathsf{Param}(x_n);\ \ldots;\ \mathsf{Param}(x_1);\ \mathcal{E}[\![\, e \,]\!]);\ \mathsf{Enter}$$

Each top level value first checks the number of arguments with an argument check instruction. Quite often however, the compiler *can* determine if there are sufficient arguments when the function is called. The rewrite rules that are given later in this chapter will emit an EnterCode instruction if a function call is saturated. This instruction enters a function just beyond the ArgChk instruction since we know that the check will succeed. For this reason, every supercombinator *always* has to start with the ArgChk instruction or otherwise the EnterCode instruction will enter the function at the wrong location!

6.5.2 Expressions

Expressions are translated with the $\mathcal{E}$ scheme.

$$\mathcal{E}[\![\, let\ \texttt{in}\ e \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{L}[\![\, let \,]\!]; \mathcal{E}[\![\, e \,]\!]$$
$$\mathcal{E}[\![\, \texttt{let!}\ x = e\ \texttt{in}\ e' \,]\!] \quad \Rightarrow$$
$$\quad \mathsf{Eval}(\mathsf{Atom}(\mathcal{E}[\![\, e \,]\!]);\ \mathsf{Enter});\ \mathsf{Var}(x);\ \mathcal{E}[\![\, e' \,]\!]$$
$$\mathcal{E}[\![\, \texttt{match}\ x\ \texttt{with}\ \{\ alts\ \} \,]\!]$$
$$\quad \mathsf{PushVar}(x);\ \mathcal{M}[\![\, alts \,]\!]$$
$$\mathcal{E}[\![\, prim^n\ a_1 \ldots a_n \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{A}[\![\, a_n \,]\!];\ \ldots;\ \mathcal{A}[\![\, a_1 \,]\!];\ \mathsf{Call}(prim, n)$$
$$\mathcal{E}[\![\, a \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{A}[\![\, a \,]\!]$$

Let bindings are translated with the $\mathcal{L}$ scheme. A strict binding first evaluates its right hand side, leaving the result on the stack and continues with the evaluation of the body. A `match` statement pushes the value to be matched and uses the $\mathcal{M}$ scheme to translate the alternatives. A primitive call is handled by the Call instruction. Atomic expressions are translated with the $\mathcal{A}$ scheme.

6.5.3 Atomic expressions

The $\mathcal{A}$ scheme wraps the instructions in an Atom pseudo instruction to slide out any intermediate local variables arising from nested `let` expressions.

$$\mathcal{A}[\![\, a \,]\!] \quad \Rightarrow$$
$$\quad \mathsf{Atom}(\mathcal{A}'[\![\, a \,]\!])$$

The $\mathcal{A}'$ scheme translates atomic expressions without entering them, effectively delaying their computation.

$$\mathcal{A}'[\![\, let\ \mathtt{in}\ a \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{L}[\![\, let \,]\!]; \mathcal{A}'[\![\, a \,]\!];$$
$$\mathcal{A}'[\![\, x\ a_1\ ...\ a_n \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{A}[\![\, a_n \,]\!];\ \ldots\ ;\ \mathcal{A}[\![\, a_1 \,]\!];\ \mathsf{PushVar}(x);\ \mathsf{NewAp}(n+1);$$
$$\mathcal{A}'[\![\, f\ a_1\ ...\ a_n \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{A}[\![\, a_n \,]\!];\ \ldots\ ;\ \mathcal{A}[\![\, a_1 \,]\!];\ \mathsf{PushCode}(f);\ \mathsf{NewAp}(n+1);$$
$$\mathcal{A}'[\![\, con_t^n\ a_1\ ...\ a_n \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{A}[\![\, a_n \,]\!];\ \ldots\ ;\ \mathcal{A}[\![\, a_1 \,]\!];\ \mathsf{NewCon}(t, n);$$
$$\mathcal{A}'[\![\, i \,]\!] \quad \Rightarrow$$
$$\quad \mathsf{PushInt}(i);$$

Note that this simple translation scheme is quite inefficient – it allocates an application node for every function call. Take for example the following expression:

$$swap\ f\ x\ y\ =\ f\ y\ x$$

Using the simple translation scheme, *swap* is translated into:

```
ArgChk(3); Atom(
  Param(y); Param(x); Param(f);
  Atom(PushVar(x); NewAp(1));
  Atom(PushVar(y); NewAp(1));
  Atom(PushVar(f); NewAp(1));
  NewAp(3))
Enter
```

After *resolve*, this instruction stream becomes:

```
ArgChk(3);
PushVar(1); NewAp(1); Slide(1, 0);
PushVar(3); NewAp(1); Slide(1, 0);
PushVar(2); NewAp(1); Slide(1, 0);
NewAp(3); Slide(1, 3);
Enter
```

Instead of just pushing the arguments on the stack and entering the function f, the code first builds an application node with application nodes for each variable, which is subsequently entered, unpacked and, only than, the function f is entered!

Fortunately, we can use some simple rewrite rules on the instruction stream to remove these inefficiencies. Using the rewrite rules from section 6.5.8, the instruction stream becomes much more efficient:

$$\mathsf{ArgChk}(3); \mathsf{PushVar}(1); \mathsf{PushVar}(3); \mathsf{PushVar}(2); \mathsf{Slide}(3,3); \mathsf{Enter}$$

We made the compilation scheme as simple and straightforward as possible and let the compiler do its optimizations on the LVM language and the instruction streams. It is quite easy to prove that transformations on the LVM language and instruction stream are correct. For example, the above transformation is simply a matter of applying the operational semantics described in section 6.3. In contrast, proving that the compilation scheme is correct is much harder – we have to show a correspondence between the operational semantics of the LVM language and the translated instructions. By making the compilation scheme naive, we hope that it becomes at least 'obviously' correct.

6.5.4 Let expressions

$$\begin{array}{l}
\mathcal{L}[\![\, \texttt{let}\ x = a \,]\!] \quad \Rightarrow \\
\qquad \mathcal{A}[\![\, a \,]\!]; \mathsf{Var}(x) \\
\mathcal{L}[\![\, \texttt{letrec}\ \{\ x_1 = a_1;\ ...;\ x_n = a_n;\ \}\,]\!] \quad \Rightarrow \\
\qquad \mathsf{Atom}(\mathcal{U}[\![\, a_1 \,]\!]);\ \mathsf{Var}(x_1);\ ...;\ \mathsf{Atom}(\mathcal{U}[\![\, a_n \,]\!]);\ \mathsf{Var}(x_n); \\
\qquad \mathsf{Init}(\mathcal{I}[\![\, x_1 = a_1 \,]\!]);\ ...;\ \mathsf{Init}(\mathcal{I}[\![\, x_n = a_n \,]\!])
\end{array}$$

The rule for `letrec` first allocates uninitialized values for its bindings using the $\mathcal{U}$ scheme and binds the stack slots to its local variables using the Var pseudo instruction. Later, the values are initialized using the $\mathcal{I}$ scheme. The rule for `let` is not concerned with recursive bindings and immediately allocates a value.

The $\mathcal{U}$ scheme allocates an uninitialized application- or constructor node that later initialized. This allows the different bindings in a `letrec` expression to refer to each other.

$$\begin{array}{l}
\mathcal{U}[\![\, \mathit{let}\ \texttt{in}\ a \,]\!] \quad \Rightarrow \\
\qquad \mathcal{U}[\![\, a \,]\!] \\
\mathcal{U}[\![\, \mathit{id}\ a_1\ ...\ a_n \,]\!] \quad \Rightarrow \\
\qquad \mathsf{AllocAp}(n+1); \\
\mathcal{U}[\![\, \mathit{con}_t^n\ a_1\ ...\ a_n \,]\!] \quad \Rightarrow \\
\qquad \mathsf{AllocCon}(t,n);
\end{array}$$

Later, the $\mathcal{I}$ scheme is used to initialize each node with the proper values.

$$\mathcal{I}[\![\, x = \mathit{let}\ \texttt{in}\ a \,]\!] \quad \Rightarrow$$

$$\mathcal{L}[\![\, let \,]\!]; \mathcal{I}[\![\, x = a \,]\!]$$
$$\mathcal{I}[\![\, x = x'\ a_1\ ...\ a_n \,]\!] \quad \Rightarrow$$
$$\mathcal{A}[\![\, a_n \,]\!];\ \ldots\ ;\ \mathcal{A}[\![\, a_1 \,]\!];\ \mathsf{PushVar}(x');\ \mathsf{PackAp}(x, n+1);$$
$$\mathcal{I}[\![\, x = f\ a_1\ ...\ a_n \,]\!] \quad \Rightarrow$$
$$\mathcal{A}[\![\, a_n \,]\!];\ \ldots\ ;\ \mathcal{A}[\![\, a_1 \,]\!];\ \mathsf{PushCode}(f);\ \mathsf{PackAp}(x, n+1);$$
$$\mathcal{I}[\![\, x = con_t^n\ a_1\ ...\ a_n \,]\!] \quad \Rightarrow$$
$$\mathcal{A}[\![\, a_n \,]\!];\ \ldots\ ;\ \mathcal{A}[\![\, a_1 \,]\!];\ \mathsf{PackCon}(x, n);$$

6.5.5 Matching

A `match` is translated with the $\mathcal{M}$ scheme.

$$\mathcal{M}[\![\, pat_1 \texttt{ -> } e_1 \texttt{ ; } ... \texttt{ ; } pat_n \texttt{ -> } e_n \,]\!] \quad \mid \exists\, i.\ pat_i \text{ is a constructor pattern } \Rightarrow$$
$$\mathsf{MatchCon}(\mathcal{P}[\![\, pat_1 \texttt{ -> } e_1 \,]\!],\ ...,\ \mathcal{P}[\![\, pat_n \texttt{ -> } e_n \,]\!])$$
$$\mathcal{M}[\![\, pat_1 \texttt{ -> } e_1 \texttt{ ; } ... \texttt{ ; } pat_n \texttt{ -> } e_n \,]\!] \quad \mid \exists\, i.\ pat_i \text{ is an integer pattern } \Rightarrow$$
$$\mathsf{MatchInt}(\mathcal{P}[\![\, pat_1 \texttt{ -> } e_1 \,]\!],\ ...,\ \mathcal{P}[\![\, pat_n \texttt{ -> } e_n \,]\!])$$

The patterns result in a list of tuples, the first element containing the value to be matched and the second the instructions to be executed. The code generation phase will arrange the code correctly.

Each pattern is compiled with the $\mathcal{P}$ scheme.

$$\mathcal{P}[\![\, con_t^n\ x_1\ ...\ x_n \texttt{ -> } e \,]\!] \quad \Rightarrow$$
$$\langle t,\ \mathsf{Atom}(\mathsf{Param}(x_n);\ ...;\ \mathsf{Param}(x_1);\ \mathcal{E}[\![\, e \,]\!])\rangle$$
$$\mathcal{P}[\![\, i \texttt{ -> } e \,]\!] \quad \Rightarrow$$
$$\langle i,\ \mathsf{Atom}(\mathcal{E}[\![\, e \,]\!])\rangle$$
$$\mathcal{P}[\![\, x \texttt{ -> } e \,]\!] \quad \Rightarrow$$
$$\langle x,\ \mathsf{Atom}(\mathsf{Param}(x);\ \mathcal{E}[\![\, e \,]\!])\rangle$$

Note that the Param instruction is used to bind the values of a matched constructor. As we saw in section 6.3.7, it is quite important that a `match` automatically unpacks the constructor as it allows us to return the constructor on the stack sometimes without allocation (the *return in registers* convention).

6.5.6 Optimized schemes

Although we tried to make the compilation scheme as straightforward as possible, some transformations are hard to apply during a different phase. For example, the following rule discards a stack push of a value that has just been evaluated to

be matched. However, it can only do so if the bound variable is not used in the alternatives. This is a good example of where we need both high level information (is x used in the alternatives?) and low-level information (we can skip a PushVar instruction).

$$\mathcal{E}[\![\,\texttt{let!}\ x = e\ \texttt{in}\ \texttt{match}\ x\ \texttt{with}\ alts\,]\!] \quad \mid x \notin fv(alts) \Rightarrow$$
$$\quad \mathsf{Eval}(\mathsf{Atom}(\mathcal{E}[\![\,e\,]\!]); \mathsf{Enter});\ \mathcal{M}[\![\,alts\,]\!]$$

Another important optimization removes superfluous continuation frames. This is especially important for efficient arithmetic. For example:

$$\begin{aligned} &\mathit{discriminant}\ a\ b\ c = \texttt{let!}\ ac = a * c\ \texttt{in} \\ &\qquad\qquad\qquad\qquad\quad \texttt{let!}\ ac4 = 4 * ac\ \texttt{in} \\ &\qquad\qquad\qquad\qquad\quad \texttt{let!}\ b2 = b * b\ \texttt{in}\ b2 + ac4 \end{aligned}$$

If we suppose that a, b and c are already in weak head normal form and that $*$ and $+$ expand to the primitive MulInt and AddInt instructions, we would get the following instruction sequence (after some rewriting):

$$\begin{aligned} &\mathsf{ArgChk}(3) \\ &\mathsf{Eval}(\mathsf{PushVar}(c); \mathsf{PushVar}(a); \mathsf{MulInt}; \mathsf{Slide}(1,0); \mathsf{Enter}) \\ &\mathsf{Var}(ac); \\ &\mathsf{Eval}(\mathsf{PushVar}(ac); \mathsf{PushInt}(4); \mathsf{MulInt}; \mathsf{Slide}(1,0); \mathsf{Enter}) \\ &... \end{aligned}$$

However, the result of MulInt is already in weak head normal form and entering the value will only return immediately to the continuation frame pushed by Eval. A much better instruction sequence is possible:

$$\begin{aligned} &\mathsf{ArgChk}(3) \\ &\mathsf{PushVar}(c); \mathsf{PushVar}(a); \mathsf{MulInt}; \\ &\mathsf{Var}(ac); \\ &\mathsf{PushVar}(ac); \mathsf{PushInt}(4); \mathsf{MulInt}; \\ &... \end{aligned}$$

In general, when an expression is evaluated that is already in weak head normal form, we don't need to evaluate it again.

$$\mathcal{E}[\![\,\texttt{let!}\ x = e\ \texttt{in}\ e'\,]\!] \quad \mid \mathit{whnf}(e) \Rightarrow$$
$$\quad \mathsf{Atom}(\mathcal{E}[\![\,e\,]\!]); \mathsf{Var}(x); \mathcal{E}[\![\,e'\,]\!]$$

The *whnf* predicate determines whether the expression e puts a value in weak head normal form on the stack. We assume that every primitive operation *prim* has an associated type t which is annotated with a (`!`) when the result is always in weak head normal form. The function *whnf* can be conservatively defined as:

$$
\begin{array}{lcl}
whnf\ (let\ \texttt{in}\ e) & = & whnf(e) \\
whnf\ (\texttt{let!}\ x\ \texttt{=}\ e\ \texttt{in}\ e') & = & whnf(e') \\
whnf\ (\texttt{match}\ x\ \texttt{with}\ alts) & = & whnfAlts\ alts \\
whnf\ (x\ a_1\ ...\ a_n) & = & False \\
whnf\ (con_t^n\ a_1\ ...\ a_n) & = & True \\
whnf\ (i) & = & True \\
whnf\ (prim^n\ a_1\ ...\ a_n) & \mid\ prim\ ::\ t_1 \rightarrow ... \rightarrow t_n \rightarrow t\,\texttt{!} & = True \\
 & \mid\ otherwise & = False
\end{array}
$$

$$
\begin{array}{lcl}
whnfAlts\ (\{\ alt_1\ \texttt{;}\ ...\ \texttt{;}\ alt_n\ \}) & = & whnfAlt(alt_1)\ \&\ ...\ \&\ whnfAlt(alt_n) \\
whnfAlt\ (pat\ \texttt{->}\ e) & = & whnf(e)
\end{array}
$$

6.5.7 Resolve stack offsets

The *resolve* phase resolves all offsets of local variables and removes the Param, Var, Init and Atom pseudo instructions. Guided by these pseudo instructions, the algorithm simulates the stack and calculates the correct offsets for each variable.

The resolve monad

We use a monadic formulation of the algorithm. The monad type is defined as:

$$\textbf{newtype}\ M\ a\ =\ M\ (\langle Env,\ Depth\rangle\ \rightarrow\ \langle a,\ Depth\rangle)$$

The monad uses an enviroment, *Env* that maintains the mapping from local variables to their stack location. The monad also has a state *Depth* that contains the current depth of the (simulated) stack.

The monadic functions are defined as usual (Hutton and Meijer, 1996):

$$
\begin{array}{l}
return\ x\ = \\
\quad M\ (\backslash\langle env,\ depth\rangle \rightarrow \langle x,\ depth\rangle) \\
(M\ m)\ \texttt{>>=}\ f\ = \\
\quad M\ (\backslash\langle env,\ depth\rangle \rightarrow \\
\quad\ \textbf{let}\ \langle x,\ depth'\rangle\ =\ m\ \langle env,\ depth\rangle \\
\quad\quad (M\ fm)\ =\ f\ x \\
\quad\ \textbf{in}\ fm\ \langle env,\ depth'\rangle
\end{array}
$$

The *push* and *pop* non-proper morphisms simulate stack movements.

$$
\begin{array}{l}
pop\ n\ = \\
\quad M\ (\backslash\langle env,\ depth\rangle \rightarrow \langle (),\ depth - n\rangle) \\
push\ n\ = \\
\quad M\ (\backslash\langle env,\ depth\rangle \rightarrow \langle (),\ depth + n\rangle)
\end{array}
$$

The *depth* function returns the current stack depth.

```
depth =
    M (\⟨env, depth⟩ → ⟨depth, depth⟩ )
```

Variables are bound using *bind* and the function *offset* returns their current offset relative to the top of the stack.

```
offset x =
    M (\⟨env, depth⟩ → ⟨depth − env[x], depth⟩)
bind x (M m) =
    M (\⟨env, depth⟩ → m ⟨env ⊕ { x ↦ depth}, depth⟩)
```

Addressing variables relative to the top of the stack removes the need for a separate *base pointer*, which is still used in some C compilers to aid debuggers.

The algorithm

An instruction stream is resolved by the *resolves* function.

```
resolves (Param(x) : instrs) =
    do{ push 1; bind x (resolves instrs) }
resolves (Var(x) : instrs) =
    bind x (do{ resolves instrs })
resolves (instr : instrs) =
    do{ is ← resolve instr
        iss ← resolves instrs
        return (is ++ iss) }
```

Individual instructions are resolved by the *resolve* function. Note that we allow ourselves some freedom by reusing the PushVar instruction such that it can contain either an argument name or resolved stack offset.

```
resolve PushVar(x) =
    do{ ofs ← offset x;
        push 1;
        return [PushVar(ofs)] }
resolve PackAp(x, n) =
    do{ ofs ← offset x;
        pop n;
        return [PackAp(ofs, n)] }
resolve PackCon(x, n) =
    do{ ofs ← offset x;
        pop n;
        return [PackCon(ofs, n)] }
```

```
resolve Eval(is) =
    do{ push 3;
        is' ← resolves is;
        pop 3;
        return [Eval(is')] }
resolve Atom(is) =
    do{ resolveSlide 1 is }
resolve Init(is) =
    do{ resolveSlide 0 is }
resolve (MatchCon(alts)) =
    do{ pop 1;
        alts' ← sequence (map resolveAlt alts);
        return [MatchCon(alts')] }
resolve instr =
    do{ effect instr; return [instr] }
```

The *resolveSlide n is* function slides out any dead values on the stack, only preserving the top n stack values.

```
resolveSlide n is =
    do{ d0 ← depth;
        is' ← resolves is;
        d1 ← depth;
        let m = d1 − d0 − n
        pop m;
        return (is' ++ [Slide(n, m)]) }
```

Alternatives are resolved with *resolveAlt*. Note that every alternative should return with the same stack depth.

```
resolveAlt ⟨t, is⟩ =
    do{ is' ← resolves is; return ⟨t, is⟩ }
```

Most instructions are not transformed but they do have an effect on the stack. The *effect* function simulates this effect in the resolve monad.

```
effect PushCode(f)   = push 1
effect AllocAp(n)    = push 1
effect AllocCon(t,n) = push 1
effect NewAp(n)      = do{ pop n; push 1 }
effect NewCon(t,n)   = do{ pop n; push 1 }
effect AddInt        = do{ pop 2; push 1 }
...
effect instr         = return ()
```

Using pseudo instructions together with this simple algorithm, we have cleanly

separated stack offset resolving from the translation scheme from the LVM language to instructions.

6.5.8 Rewrite rules

The rewrite rules transform a sequence of instructions into a more efficient sequence of instructions with the same semantic effect. As described in section 6.5.3, the rewrite rules describe important optimizations since the compilation scheme is quite naïve.

There are two essential rewrite rules that push instructions following a match into the branches of the match. This is needed since the branches are not able to jump to those instructions. The transformation is safe, since every alternative leaves the stack at the same depth.

$$\begin{aligned}&\mathsf{MatchCon}(alt_1, ..., alt_n); instrs \quad \Rightarrow \\ &\quad \mathsf{MatchCon}(alt_1; instrs\ , ...,\ alt_n; instrs) \\ &\mathsf{MatchInt}(alt_1, ..., alt_n); instrs \quad \Rightarrow \\ &\quad \mathsf{MatchInt}(alt_1; instrs\ , ...,\ alt_n; instrs)\end{aligned}$$

This transformation duplicates code, but fortunately, the *instrs* are always a Slide followed by an Enter, as we can see from the translation schemes. Moreover, subsequent transformations are more effective when these instructions are directly in scope.

The first optimizing rules transform partial and saturated applications. The first rule emits NewNap instructions for a known partial application – this instruction will not push an expensive update frame. The second rule uses EnterCode for saturated applications to a known top level function. This instruction behaves just like Enter except that an implementation can safely skip the expensive argument check for the entered function (a *direct entry point*).

$$\begin{aligned}&\mathsf{PushCode}(f); \mathsf{NewAp}(n) \quad \mid arity(f) > (n-1) \Rightarrow \\ &\quad \mathsf{PushCode}(f); \mathsf{NewNap}(n) \\ &\mathsf{PushCode}(f); \mathsf{Slide}(n, m); \mathsf{Enter} \quad \mid arity(f) = (n-1)\ \&\ arity(f) \neq 0 \Rightarrow \\ &\quad \mathsf{Slide}(n-1, m); \mathsf{EnterCode}(f)\end{aligned}$$

If an application node is entered immediately after building it, we can safely enter the application directly without building the application node at all!

$$\begin{aligned}&\mathsf{NewAp}(n); \mathsf{Slide}(1, m); \mathsf{Enter} \quad \Rightarrow \\ &\quad \mathsf{Slide}(n, m); \mathsf{Enter} \\ &\mathsf{NewNap}(n); \mathsf{Slide}(1, m); \mathsf{Enter} \quad \Rightarrow \\ &\quad \mathsf{Slide}(n, m); \mathsf{Enter}\end{aligned}$$

These two simple rewrite rules remove the need for the usual translation schemes: one for expressions that will be entered and one for `let`-bound expressions (Peyton Jones, 1986).

The following rule moves the Slide instruction up in order to prevent space leaks while calling external functions.

$$\begin{aligned}&\mathsf{Call}(prim, n); \mathsf{Slide}(1, m); \mathsf{Enter} \quad \Rightarrow \\ &\quad \mathsf{Slide}(n, m); \mathsf{Call}(prim, n); \mathsf{Enter}\end{aligned}$$

Expressions of the form (`let!` $x = e$ `in` x) lead to code that pushes variable x and subsequently discard the original binding. We can instead discard the push and leave the original binding in place.

$$\begin{aligned}&\mathsf{PushVar}(0); \mathsf{Slide}(1, m) \quad \mid m \geq 1 \Rightarrow \\ &\quad \mathsf{Slide}(1, m-1)\end{aligned}$$

The previous rule naturally generalizes to a sequence of n pushes:

$$\begin{aligned}&\mathsf{PushVar}_1(n-1); ...; \mathsf{PushVar}_n(n-1); \mathsf{Slide}(n, m) \quad \mid m \geq n \Rightarrow \\ &\quad \mathsf{Slide}(n, m-n)\end{aligned}$$

If a value is entered that is already in weak head normal form, we can directly use the Return instruction. We assume that all primitive functions have a type that ends with a (`!`) when they return a strict result. This is the case for many primitive operations and instructions.

$$\begin{aligned}&\mathsf{Call}(prim, n); \mathsf{Enter} \quad \mid prim :: t_1 \rightarrow ... \rightarrow t_n \rightarrow t\,! \Rightarrow \\ &\quad \mathsf{Call}(prim, n); \mathsf{Return}\end{aligned}$$

Commonly, a constructor or literal is returned. The LVM has the special ReturnCon and ReturnInt instructions that can potentially execute without extra heap allocation resulting from building a new constructor. Instead of building a new constructor that is immediately entered, the constructor is kept on the stack (see section 6.3.7). This is the 'return in registers' convention as described in the STG machine paper (Peyton Jones, 1992).

$$\begin{aligned}&\mathsf{NewCon}(t, n); \mathsf{Slide}(1, m); \mathsf{Enter} \quad \Rightarrow \\ &\quad \mathsf{Slide}(n, m); \mathsf{ReturnCon}(t, n) \\ &\mathsf{PushInt}(i); \mathsf{Slide}(1, m); \mathsf{Enter} \quad \Rightarrow \\ &\quad \mathsf{Slide}(0, m); \mathsf{ReturnInt}(i)\end{aligned}$$

An LVM interpreter can cheaply test a variable to see if it is already in a weak head normal form. The EvalVar instruction can use this in order to avoid creating a continuation frame that is immediately popped.

$$\begin{aligned}&\mathsf{Eval}(\mathsf{PushVar}(ofs); \mathsf{Slide}(1, 0); \mathsf{Enter}) \quad \Rightarrow \\ &\quad \mathsf{EvalVar}(ofs - 3)\end{aligned}$$

Quite often, we can merge slides, arising from instructions pushed into an alternative.

$$\begin{array}{l}\mathsf{Slide}(n0, m0); \mathsf{Slide}(n1, m1) \quad \mid n1 \leq n0 \Rightarrow \\ \quad \mathsf{Slide}(n1, m0 + m1 - (n0 - n1))\end{array}$$

The last rules deal with instructions that have no effect and primitive instructions. By treating instructions like AddInt as a primitive call, the compiler can be simplified since it doesn't need special code to deal with built-in operations. In a sense, these instructions are just like external calls except that they have a very efficient calling convention and encoding.

$$\begin{array}{l}\mathsf{Call}(prim, n) \quad \mid prim = \texttt{instr}\ instr :: t_1 \to ... \to t_n \to t \Rightarrow \\ \quad instr \\ \mathsf{NewAp}(n) \quad \mid n \leq 1 \Rightarrow \\ \quad - \\ \mathsf{Slide}(n, 0) \quad \Rightarrow \\ \quad -\end{array}$$

Together, the above rules have proven to be quite effective in optimizing the instructions generated by the naive translation scheme. Careful study of the generated code shows that hardly any improvements on this level are attainable. The simple translation scheme in combination with these rewrite rules also make the compiler much simpler. Furthermore, the rewrite rules even seem to perform *better* than optimized translation schemes as the rewrite rules sometimes find optimization opportunities between instructions that are unrelated at the language level.

6.5.9 Code generation

The code generation phase resolves the code offsets relative to program counter.

$$\begin{array}{l}codegens\ is\ = \\ \quad concat\ (map\ codegen\ is) \\ codegen\ \mathsf{Eval}(is)\ = \\ \quad \textbf{let}\ is'\ =\ codegens\ is \\ \quad \textbf{in}\ [\mathsf{PushCont}(size\ is')]\ +\!\!+\ is' \\ codegen\ \mathsf{PushCode}(f)\ = \\ \quad [\mathsf{PushCode}(index(f))] \\ codegen\ \mathsf{EnterCode}(f)\ = \\ \quad [\mathsf{EnterCode}(index(f))] \\ codegen\ \mathsf{MatchCon}(alts)\ = \\ \quad \textbf{let}\ iss\ =\ map\ (codegen\ .\ snd)\ alts \\ \qquad tags\ =\ map\ fst\ alts \\ \qquad ofss\ =\ scanl\ (+)\ 0\ (map\ size\ iss) \\ \quad \textbf{in}\ [\mathsf{MatchCon}(length\ alts, 0, zip\ tags\ ofss)]\ +\!\!+\ concat\ iss\end{array}$$

$$
\begin{array}{l}
codegen\ instr\ = \\
\quad [instr]
\end{array}
$$

For simplicity, the rule for MatchCon($alts$) assumes that there are no (default) variable patterns inside $alts$. The actual rule used in the compiler is inconvenient to formulate but still straightforward. After the code resolve phase, we are done and the compiler can write a binary LVM file that can be loaded by the interpreter.

6.5.10 More optimization: superfluous stack movements

An important optimization is to reduce the number of superfluous stack movements. Due to the close relation of λ_{lvm} with the LVM instruction set, it is possible to perform this optimization at the language level instead of at the instruction level.

As an example of unnecessary stack pushes we look at a definition of the S combinator.

$$combS\ f\ g\ x = \texttt{let}\ z = g\ x\ \texttt{in}\ f\ x\ z$$

After translating, resolving, and rewriting this program, it is compiled into:

ArgChk(3);
PushVar(2 (x)); PushVar(2 (g)); NewAp(2);
PushVar(0 (z)); PushVar(4 (x)); PushVar(3 (f));
Slide(3, 4); Enter

However, the variable z is pushed on the stack immediately after building it and later discarded with the Slide instruction. Better code can be obtained by inlining the definition of z.

$$combS\ f\ g\ x = f\ x\ (g\ x)$$

This program uses the application node immediately and discards the superfluous PushVar instruction.

ArgChk(3);
PushVar(2 (x)); PushVar(2 (g)); NewAp(2);
PushVar(3 (x)); PushVar(2 (f));
Slide(3, 3); Enter

Et voilá, we can optimize stack movements (and remove dead variables) by using a standard inliner. The inliner for the LVM language can be much simpler than a full fledged inliner (Peyton Jones and Marlow, 1999) since we will not instantiate across lambda expressions but only perform local substitutions. This property makes it also easier to analyse whether work or code is ever duplicated.

It is always beneficial to inline *trivial* expressions since they duplicate neither work, nor code. Trivial expressions consist of:

- literals (*literal*),
- variables (x),
- constructors with no arguments (con_t^0).

For other expressions, we need to determine how often the binder *occurs*. The occurrence analysis can be as simple as counting the number of syntactic occurrences. If code duplication is not perceived as a problem, we can refine the analysis by taking the maximum of the occurrences inside alternatives instead of the sum. If a binder occurs only once, we can safely inline it (since lambda expressions are not part of the LVM language). When a binder has no occurrences, the binding can be removed entirely.

Inlining strict bindings

We look again at the example program *discriminant* from section 6.5.6:

$$\begin{aligned} &\mathit{discriminant}\ a\ b\ c = \texttt{let!}\ ac = a * c\ \texttt{in} \\ &\qquad\qquad\qquad\qquad\quad \texttt{let!}\ ac4 = 4 * ac\ \texttt{in} \\ &\qquad\qquad\qquad\qquad\quad \texttt{let!}\ b2 = b * b\ \texttt{in}\ b2 + ac4 \end{aligned}$$

The optimized instruction sequence was:

$$\begin{aligned} &\mathsf{ArgChk}(3) \\ &\mathsf{PushVar}(c); \mathsf{PushVar}(a); \mathsf{MulInt}; \\ &\mathsf{Var}(ac); \\ &\mathsf{PushVar}(ac); \mathsf{PushInt}(4); \mathsf{MulInt}; \\ &\ldots \end{aligned}$$

This example can be optimized a little bit more since it still pushes variable ac allthough it already resides on the stack. An optimal instruction sequence would be:

$$\begin{aligned} &\mathsf{ArgChk}(3) \\ &\mathsf{PushVar}(c); \mathsf{PushVar}(a); \mathsf{MulInt}; \\ &\mathsf{Var}(ac); \\ &\mathsf{PushInt}(4); \mathsf{MulInt}; \\ &\ldots \end{aligned}$$

Unfortunately, our simple inliner will not inline the binding for ac since `let!` bindings can not be inlined in general. However, we can define some side conditions under which the inlining of `let!` bindings is possible.

First, we extend the grammar and allow `let!` expressions as atomic expressions – of course, this is in general unsafe and should only be used 'internally'. Together with the grammar extension, the translation scheme for atomic expressions is also extended:

$$\begin{aligned}
&\mathcal{A}'[\![\,\texttt{let!}\ x = e\ \texttt{in}\ e'\,]\!] \qquad \mid \mathit{whnf}(e) \ \Rightarrow \\
&\qquad \mathsf{Atom}(\mathcal{E}[\![\,e\,]\!]); \mathsf{Var}(x); \mathcal{A}'[\![\,e'\,]\!]; \\
&\mathcal{A}'[\![\,\texttt{let!}\ x = e\ \texttt{in}\ e'\,]\!] \quad \Rightarrow \\
&\qquad \mathsf{Eval}(\mathsf{Atom}(\mathcal{E}[\![\,e\,]\!]); \mathsf{Enter}); \mathsf{Var}(x); \mathcal{A}'[\![\,e'\,]\!];
\end{aligned}$$

When an evaluated expression is both *pure* and *total*, we can transform `let!` bindings into `let` bindings. The standard inliner can now inline `let!` expressions via those `let` bindings.

$$\texttt{let!}\ x = e\ \texttt{in}\ e' \qquad \mid \mathit{pure}(e)\ \&\ \mathit{total}(e) \ \Rightarrow \quad \texttt{let}\ x = (\texttt{let!}\ x = e\ \texttt{in}\ x)\ \texttt{in}\ e'$$

The $\mathit{pure}(e)$ predicate ensures that the expression has no side effect and the $\mathit{total}(e)$ predicate ensures that the expression can not fail or loop. These conditions can probably only be approximated in practice but they can be determined exactly for many common primitive expressions like comparison and bitwise operations. However it fails for operations that can raise exceptions – like addition, multiplication and division. Note that the `let!` binding inside the `let` is still needed in order to emit an Eval instruction during compilation.

The above approach works for expressions that are both pure and total but many times we don't know enough about the expression to ensure those predicates. Other strict expressions can be inlined only if the following conditions hold:

1. the inliner never duplicates code (to avoid duplication of an impure expression).
2. the binding is used once.
3. the binding is used before any other primitive function, `let!`, or `match` construct.

The first two conditions are intrisic properties of the inliner. The last condition, is formalized with the *firstuse* predicate.

$$\texttt{let!}\ x = e\ \texttt{in}\ e' \qquad \mid \mathit{once}\ x\ e'\ \&\ \mathit{firstuse}\ x\ e' \ \Rightarrow \quad \texttt{let}\ x = (\texttt{let!}\ x = e\ \texttt{in}\ x)\ \texttt{in}\ e'$$

Note that the *firstuse* predicate is extremely dependent on the exact translation scheme that is used – after a strict binding is inlined as a `let`, we must be sure that no other strict binding gets inlined beyond the previous one.

The *firstuse* predicate is defined in terms of the *first* function that has as its second argument a possible continuation that starts out as *False*. As soon as a primitive operation, `let!`, or `match` is encountered, the continuation is set to *False* again to avoid inlining a binding beyond that construct.

$$
\begin{array}{ll}
\mathit{firstuse}\ x\ e & = \mathit{first}\ x\ \mathit{False}\ e \\
 & \\
\mathit{first}\ x\ c\ x & = \mathit{True} \\
\mathit{first}\ x\ c\ (y\ a_1\ ...\ a_n) & = \mathit{firsts}\ x\ c\ [y, a_1, ..., a_n] \\
\mathit{first}\ x\ c\ (\mathit{con}_t^n\ a_1\ ...\ a_n) & = \mathit{firsts}\ x\ c\ [a_1, ..., a_n] \\
\mathit{first}\ x\ c\ (\mathit{prim}^n\ a_1\ ...\ a_n) & = \mathit{firsts}\ x\ \mathit{False}\ [a_1, ..., a_n] \\
\mathit{first}\ x\ c\ (\texttt{match}\ y\ \texttt{with}\ \mathit{alts}) & = \mathit{False} \\
\mathit{first}\ x\ c\ (\texttt{let!}\ y\ \texttt{=}\ e\ \texttt{in}\ e') & = \mathit{first}\ x\ \mathit{False}\ e \\
\mathit{first}\ x\ c\ (\texttt{let}\ y\ \texttt{=}\ e\ \texttt{in}\ e') & = \mathit{firsts}\ x\ c\ [e, e'] \\
\mathit{first}\ x\ c\ (\texttt{letrec}\ \texttt{\{}\ y_1\ \texttt{=}\ e_1\ \texttt{;}\ ...\ \texttt{;}\ y_n\ \texttt{=}\ e_n\ \texttt{\}}\ \texttt{in}\ e') & = \mathit{firsts}\ x\ c\ [e_1, ..., e_n, e'] \\
\mathit{first}\ x\ c\ \mathit{other} & = c \\
 & \\
\mathit{firsts}\ x\ c\ \mathit{es} & = \mathit{foldl}\ (\mathit{first}\ x)\ c\ \mathit{es}
\end{array}
$$

In combination with the rewrite rules, the described optimizations remove all superfluous stack movements – there is no need for complex reorderings of bindings. Note that we could achieve this by distinguishing between normal expressions and atomic expressions, and allowing atomic expressions as arguments. This constrasts with the STG language for example, that only allows variables as arguments. In our case, λ_{lvm} directly reflects the capabilities of the abstract machine.

6.6 Assessment

We have implemented a LVM interpreter on top of the O'Caml runtime system (Leroy, 1995), which is well known for its portability and the efficient bytecode interpreter. By taking advantage of this excellent system, we were able to build an LVM interpreter in a relatively short time frame.

There is also a core compiler that translates λ_{core} programs into LVM files using the compilation rules described in section 6.5. The compiler is still very naive and doesn't perform any 'essential' optimizations like simplification, inlining or strictness analysis. Even though we tried to keep the LVM instruction set and compilation scheme as simple as possible, the total line count of the core compiler is still about 7000 lines of Haskell which is a bit disappointing. On the other hand, the core compiler has a very modular structure and it is easy to use as the backend for a real compiler or as a platform to experiment with new transformation algorithms. It is currently used as a backend to the Helium compiler and the experimental HX system (Shields and Peyton Jones, 2001).

To assess the performance of the interpreted LVM instruction set, we ran some preliminary benchmarks. Since each benchmark is rather small the results should be interpreted with care. However, we believe that the benchmarks will at least give an indication whether the performance of the an LVM interpreter is acceptable in practice. The following three programs were tested.

nfib 27 Calculates the 27th *nfib* number.

```
nfib :: Int -> Int
nfib 0  = 1
nfib 1  = 1
nfib n  = 1 + nfib (n-1) + nfib (n-2)
```

queens 9 Finds the number of ways to put 9 queens on a 9×9 checkboard where no queen threatens another.

```
queens n     = length (qqueens n n)

qqueens k 0  = [[]]
qqueens k n  = [ (x:xs) | xs <- qqueens k (n-1)
                        , x  <- [1..k], safe x 1 xs ]

safe x d []     = True
safe x d (y:ys) = x /= y && x+d /= y
                 && x-d /= y && safe x (d+1) ys
```

sieve 1000 Calculates the 1000th prime number using the sieve of Erasthones.

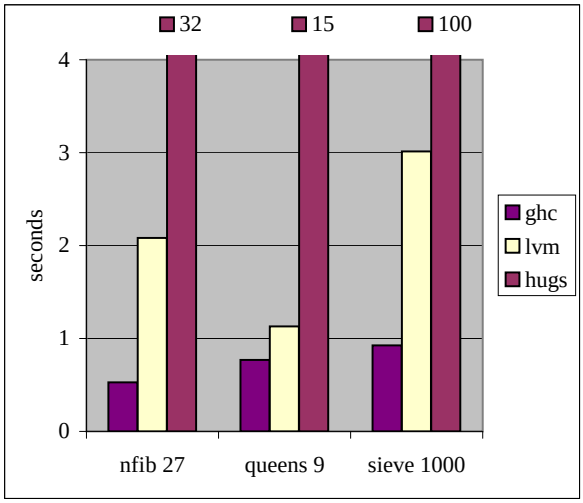

Figure 6.3: Benchmarks

```
sieve n  = last (take n (ssieve [3,5..]))
         where
          ssieve (x:xs)  = x:ssieve (filter (noDiv x) xs)
          noDiv x y      = (mod x y /= 0)
```

Each program was translated with the Hugs interpreter (May 1999), the GHC compiler (5.02) and the LVM core compiler. GHC was run without the `-O` flag but it still does simplification and inlining. Since the core compiler can not parse full Haskell, each program was manually desugared into enriched lambda expressions before compilation. All programs were run on a 266Mhz PentiumII PC with 128Mb RAM.

Figure 6.6 shows the running times of each program. Note that the running times of the programs run with Hugs are outside the scale of the y-axis. Perhaps not surprisingly, the LVM performs about 15 to 30 times better on these programs than Hugs. What is more surprising is that the interpreted, non-inlined, unsimplified LVM programs run just 3 times as slow as GHC compiled programs. The `queens` benchmark is even just 25% faster when compiled with GHC. Of course, the programs are too small to be used as realistic benchmarks but the results still give us confidence that the interpreter approach can be successful in practice.

We also measured how the LVM performs if the core compiler would have a simple strictness analyser and inliner. We naively hand-optimized the programs for the LVM, trying to emulate a simple strictness analyser and inliner. Here is for example the optimized source for *nfib*:

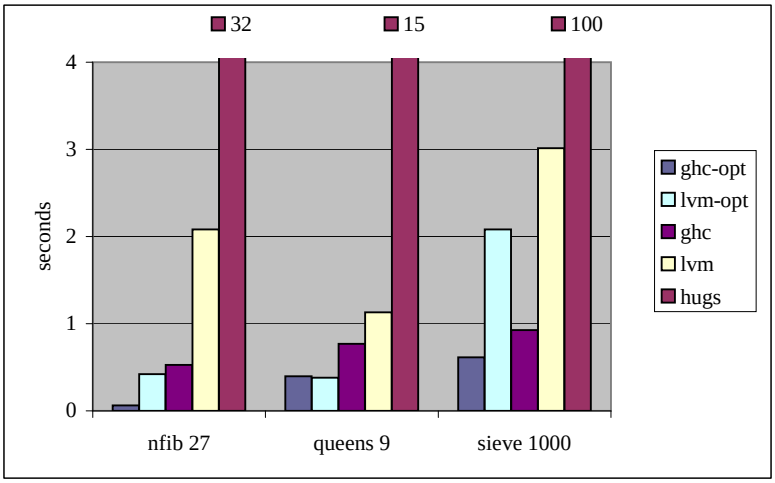

Figure 6.4: Benchmarks

```
nfib :: Int -> Int
nfib n = match n with
           0 -> 1
           1 -> 1
           n -> let! n2  = primSubInt n 2 in
                let! nf2 = nfib n2 in
                let! n1  = primSubInt n 1 in
                let! nf1 = nfib n1 in
                let! m   = primAddInt nf1 nf2 in
                primAddInt 1 m
```

Figure 6.6 shows the benchmarks with the optimized compilers. The *ghc-opt* programs are compiled with GHC with the `-O` flag while the *lvm-opt* programs are the hand-optimized sources compiled for the LVM. Optimized GHC is *much* faster on the *nfib* and *sieve* benchmarks but, surprisingly, the *queens* benchmark runs faster with the optimized LVM. We don't know for sure why the *queens* program performs so well, it might be a cache effect or it might be linked to the 'return in registers' convention that can avoid heap allocation – maybe the LVM avoids an expensive allocation in a critical part of the algorithm.

Bibliography

Lennart Augustsson. *Cayenne – a language with dependent types*. In International Conference on Functional Programming, pages 239–250, 1998.

Arthur Baars. *Syntax macros*. available at http://www.cs.uu.nl/people/arthurb/macros.html, 2002.

David Beazley. *SWIG: An easy to use tool for integrating scripting languages with C and C++* . In 4th annual Tcl/Tk workshop, Monterey, CA, July 1996. http://www.swig.org/papers/Tcl96/tcl96.html.

Richard Bird. *Introduction to Functional Programming using Haskell (2nd edition)*. Prentice Hall, 1998. ISBN 0-13-484346-0.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. *Lava: Hardware design in Haskell*. In proceedings of the ICFP, Baltimore, Maryland, 1998.

Don Box. *Essential COM*. Addison-Wesley, December 1997. ISBN 0-201-63446-5.

Kraig Brockschmidt. *Inside OLE (second edition)*. Microsoft Press, 1995.

William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley Computer Publishing, 1998. ISBN 0-471-19713-0.

Peter Buneman and Atsushi Ohori. *Polymorphism and type inference in database programming*. In ACM Transactions on Database Systems, 21(1):30–76, March 1996. http://www.jaist.ac.jp/~ohori/dblang.ps.

Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. *Comprehension syntax*. In ACM SIGMOD Record, 23(1):87–96, March 1996. ftp://ftp.cis.upenn.edu/pub/papers/db-research/sigmod_record94.ps.Z.

Manuel Chakravarty (ed.). *The Haskell 98 Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report*, 2002. http://www.cse.unsw.edu.au/~chak/haskell/ffi.

Manuel Chakravarty. *C → haskell, or yet another interfacing tool.* In Implementation of Functional Languages, 11th. International Workshop (IFL'99), LNCS, 1868. Springer-Verlag, 2000.

C.J. Date. *An Introduction to Database Systems (6th edition).* Addison-Wesley, 1995. ISBN 0-201-82458-2.

Conal Elliott and Paul Hudak. *Functional reactive animation.* In the International Conference on Functional Programming (ICFP), Amsterdam, the Netherlands, June 1997.

Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. *H/Direct: A Binary Foreign Language Interface to Haskell.* In the International Conference on Functional Programming (ICFP), Baltimore,USA, 1998. Also appeared in ACM SIGPLAN Notices 34, 1, (Jan. 1999).
http://www.cs.uu.nl/people/daan/pubs.html.

Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. *Calling hell from heaven and heaven from hell.* In the International Conference on Functional Programming (ICFP), Paris,France, 1999. Also appeared in ACM SIGPLAN Notices 34, 9, (Sep. 1999).
http://www.cs.uu.nl/people/daan/pubs.html.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

Ben R. Gaster and Mark P. Jones. *A polymorphic type system for extensible records and variants.* Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, 1996.

David N. Gray, Hotchkiss John, LaForge Seth, Shalit Andrew, and Weinberg Tony. *Modern languages and microsoft's component object model.* In Communications of the ACM, 41(5):55–65. ACM press, May 1998.

Dominik Gruntz. *Implementing COM objects with the Direct-To-COM compiler.* http://www.oberon.ch/resources/component_pascal/com_example.html, 1998.

Paul Hudak. *Modular domain specific languages and tools.* In Fifth International Conference on Software Reuse (ICSR'98), pages 134–142, Victoria, B.C., Canada, June 1998. IEEE Computer Society Press.
http://www.cs.yale.edu/homes/hudak-paul/hudak-dir/icsr98.ps.

Lorenz Huelsbergen. *A portable C interface for standard ML of New Jersey.* Technical report, AT&T Bell Laboratories, January 1996. http://cm.bell-labs.com/cm/cs/what/smlnj.

John Hughes. *The Design and Implementation of Programming Languages.* PhD thesis, University of Oxford, 1984.

John Hughes. *Why Functional Programming Matters*. Computer Journal, 32(2):98–107, 1989.

John Hughes. *The Design of a Pretty-printing Library*. In J. Jeuring and E. Meijer, editors, Advanced Functional Programming, 925:53–96. Springer Verlag, LNCS, 1995.

John Hughes. *Generalising monads to arrows*. Science of Computer Programming, 37:67–111, 2000. http://www.cs.chalmers.se/~rjmh/Papers/arrows.ps.

Graham Hutton and Erik Meijer. *Monadic parser combinators*. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996. http://www.cs.nott.ac.uk/Department/Staff/gmh/monparsing.ps.

David Jeffery, Tyson Dowd, and Zoltan Somogyi. *MCORBA: a CORBA binding for Mercury*. In Proceedings of the First International Workshop on Practical Aspects of Declarative Languages, LNCS, 1551:211–222, San Antonio, Texas, January 1999. Springer-Verlag.

Thomas Johnsson. *Efficient compilation of lazy evaluation*. In proceedings of the ACM Conference on Compiler Construction, pages 58–69, Montreal, Canada, June 1984.

Thomas Johnsson. *Lambda lifting: Transforming programs to recursive equations*. In proceedings of Functional Programming Languages and Computer Architecture, LNCS, 201:190–203, Nancy, France, September 1985.

Mark Jones. *The implementation of the Gofer functional programming system*. Technical Report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, May 1994.

Peter J. Landing. *The next 700 programming languages*. Communications of the ACM, 9(3):157–164, March 1966.

Konstantin Läufer and Martin Oderski. *An extension of ML with first-class abstract types*. In Workshop on ML and its Applications, 1992.

Daan Leijen and Erik Meijer. *Domain specific embedded compilers*. In Second USENIX Conference on Domain Specific Languages (DSL'99), pages 109–122, Austin, Texas, October 1999. USENIX Association. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).
http://www.cs.uu.nl/people/daan/pubs.html.

Daan Leijen and Erik Meijer. *Parsec: Direct style monadic parser combinators for the real world*. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001. http://www.cs.uu.nl/people/daan/pubs.html.

Daan Leijen, Erik Meijer, and James Hook. *Haskell as an automation controller*. In 3rd International Summerschool on Advanced Functional Programming, 1608, Braga, Portugal, 1999. Springer Lecture Notes in Computer Science (LNCS).

Daan Leijen. *Functional components: COM components in Haskell*. Master's thesis, University of Amsterdam, 1998.

Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language*. Technical Report 117, INRIA, 1990.

Xavier Leroy. *Le système caml special light: modules et compilation efficace en caml*. Technical Report 2721, INRIA, France, November 1995.

Xavier Leroy. *Interfacing C with Objective Caml*. http://caml.inria.fr/ocaml, 1996.

Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Addison-Wesley, April 1999. ISBN 0-201-43294-3.

X/Open Company Ltd. *X/Open Preliminary Specification X/Open DCE: Remote Procedure Call*, 1993.

Simon Marlow. *Writing high-performance server applications in haskell, case study: A haskell web server*. In Haskell Workshop, Montreal, Canada, September 2000.

Erik Meijer, Daan Leijen, and James Hook. *Client-side web scripting with HaskellScript*. In Proceedings of Practical Aspects of Declarative Languages (PADL), 1999.

Erik Meijer. *Server side web scripting in haskell*. Journal of Functional Programming, 10(1):1–18, January 2000.

Microsoft Press. *The COM Reference*, 1992.

Microsoft. *SDK for Java*, 1998. http://www.microsoft.com/java/.

Microsoft. *Developing for Microsoft Agent*, 1998.

Jon Mountjoy. *The spineless tagless g-machine, naturally*. In Proceedings of the third ACM SIGPLAN International Conference on Functional programming, Baltimore, Maryland, 1998.

Object Management Group. *The Common Object Request Broker: Architecture and Specification (revision 1.2)*. Object Management Group, 1993. OMG Document Number 93.12.43.

Simon Peyton Jones and Robert Ennals. *Optimistic evaluation: a fast evaluation strategy for non-strict programs*. submitted to ICFP'03, March 2003.

Simon Peyton Jones and David Lester. *A modular fully-lazy lambda lifter in Haskell*. Software – Practice and Experience, 21(5):479–506, 1991.

Simon Peyton Jones and Simon Marlow. *Secrets of the glasgow haskell compiler inliner*. In Workshop on Implementing Declarative Languages, 1999. revised version submitted to JFP, Feb 2001.

Simon Peyton Jones and Andre Santos. *A transformation-based optimiser for Haskell*. Science of Computer Programming, 32(1–3):3–47, September 1998.

Simon Peyton Jones and Tim Sheard. *Template meta-programming for Haskell*. In Haskell Workshop, pages 1–16, Pittsburg, Pennsylvania, 2002.

Simon L. Peyton Jones and Philip Wadler. *Imperative functional programming*. In POPL, 20:71–84, 1993.

Simon Peyton Jones, Will Partain, and Andre Santos. *Let-floating: moving bindings to give faster programs*. In International Conference on Function Programming (ICFP), Philadelphia, May 1996.

Simon Peyton Jones, Thomas Nordin, and Alastair Reid. *Green Card: a foreign-language interface for Haskell*. In Haskell Workshop, 1997.

Simon Peyton Jones, Erik Meijer, and Daan Leijen. *Scripting COM components from Haskell*. In Fifth International Conference on Software Reuse (ICSR'98), Victoria, B.C., Canada, June 1998. IEEE Computer Society Press.
http://www.cs.uu.nl/people/daan/pubs.html.

Simon Peyton Jones, Simon Marlow, and Conal Elliott. *Stretching the storage manager: weak pointers and stable names in Haskell*. In IFL, 1999.

Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, International series in computer science, 1986. ISBN 0-13-453325-9.

Simon Peyton Jones. *Implementing non-strict languages on stock hardware: The Spineless Tagless G-machine*. Journal of Functional Programming, 2(2):127–202, April 1992.
http://www.research.microsoft.com/~simonpj/Papers/papers.html#compiler.

Marinus Plasmeijer and Marco van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993. ISBN 0-201-41663-8.

Gordon D. Plotkin. *A structural approach to operational semantics*. Technical Report DAI-MI FN-19, Computer Science department, Aarhus University, September 1981.

Dale Rogerson. *Inside COM*. Microsoft Press, 1997.

John R. Rose and Hans Muller. *Integrating the Scheme and C languages*. In Proceedings of the ACM 1992 Conference on Lisp and Functinal Programming, pages 247–259, San Francisco,California, 1992.

M. Serrano. *Bigloo user's manual*. http://www-sop.inria.fr/mimosa/fp/Bigloo, 1999.

Mark Shields and Simon Peyton Jones. *First-class modules for Haskell*. In Ninth International Conference on Foundation of Object-Oriented Languages (FOOL 9), Portland, Oregon, December 2001.

Mark Shields. *Static Types for Dynamic Documents*. PhD thesis, Oregon Graduate Institute, February 2001.

Julian Smart. *The wxWindows GUI library*. http://www.wxwindows.org, 1992.

David Sussman. *ADO programmer's reference (3rd edition)*. Wrox Press inc, August 2000. ISBN 1-861-00463-X.

Doaitse Swierstra and Pablo Azero Alcocer. *Fast, error correcting parser combinators: A short tutorial*. In SOFSEM'99, Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics, LNCS, 1725:111–129. Springer-Verlag, November 1999.
http://www.cs.uu.nl/groups/ST/Software/UU_Parsing.

Doaitse Swierstra, Pablo Azero Alcocer, and Saraiava J. *Designing and implementing combinator languages*. In Advanced Functional Programming, Third international school (AFP'98), LNCS, 1608:150–206. Springer-Verlag, 1999.

Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998. ISBN 0-201-17888-5.

A. Vogel and B. Gray. *Translating DCE IDL in OMG IDL and vice versa*. Technical Report 22, CRC for Distributed Systems Technology, Brisbane, 1995.

A. Vogel, B. Gray, and K. Duddy. *Understanding any IDL, lesson one: DCE and CORBA*. In SDNE, 1996.

Philip Wadler. *Comprehending Monads*. In Mathemetical Structures in Computer Science, 2:461–493, 1992. Special issue of papers from the 6'th Conference on Lisp and Functional Programming.
http://cm.bell-labs.com/cm/cs/who/wadler/topics/monads.html.

Philip Wadler. *The essence of functional programming*. In 19'th Symposium on Principles of Programming Languages, pages 1–14, Albuquerque, New Mexico, January 1992. ACM press.
http://cm.bell-labs.com/cm/cs/who/wadler/topics/monads.html.

Philip Wadler. *A prettier printer*. Talk at *The Fun of Programming*,A symposium in honour of Professor Richard Bird's 60th birthday. Oxford, 24-25 March 2003. (Original paper April 1997, revised March 1998.) http://www.research.avayalabs.com/user/wadler/topics/language-design.html#prettier, 1997.

Malcolm Wallace. *Calling Haskell from C using Green Card*. http://www.cs.york.ac.uk/fp/nhc, 1998.

Samenvatting

In het invloedrijke artikel “Why functional programming matters” beargumenteert John Hughes dat de kracht van declaratieve, niet stricte, hogere orde talen ligt in hun vermogen om verschillende aspecten van een programma orthogonaal te specificeren. Hierbij word vooral gekeken naar de compositie op het laagste niveau, namelijk functies. In dit proefschrift bestuderen we hoe deze talen gebruikt kunnen worden bij de compositie van externe imperatieve functionaliteit en componenten.

Het eerste hoofdstuk van dit proefschrift beschrijft het ontwerp van een zogeheten “foreign function interface” (FFI) voor de pure, hogere orde taal, Haskell. Een FFI zorgt ervoor dat een programma geschreven in een bepaalde programmeertaal, andere programma’s in andere talen kan aanroepen en omgekeerd. Verschillende programmeertalen gebruiken verschillende aanroepconventies en representeren hun gegevens op andere wijze – De FFI zorgt automatisch voor de juiste aanroepconventie en transformeert gegevens naar het juiste formaat. Vooral het transformeren van de gegevens is een complexe zaak.

Omdat een FFI met een grote variëteit aan talen moet omgaan, worden de meeste FFI’s groot, complex, incompleet, en gespecificeerd zonder formele semantiek. Ons ontwerp daarentegen is formeel beschreven en gebaseerd op een standaard protocol taal (IDL). Het is bovendien zorgvuldig gescheiden in twee aparte lagen: een minimaal en primitief mechanisme dat ondersteund dient te worden door de compiler, en een apart programma, H/Direct, dat het primitieve mechanisme gebruikt om automatisch uitgebreide gegevens transformaties te genereren.

Voortbouwend op de FFI beschrijft het tweede hoofdstuk de integratie van Haskell met Microsoft’s Component Object Model (COM). Component modellen beschrijven een systeem protocol voor interactie tussen verschillende software componenten. Hierbij worden programma’s geconstrueerd door verschillende software componenten samen te voegen. Omdat deze protocollen programmeertaal onafhankelijk zijn, is dit ideaal voor exotische talen als Haskell. Als een Haskell programma kan worden verpakt als software component, dan kan deze worden gebruikt zonder kennis van Haskell. Andersom geeft zo’n integratie de Haskell gebruiker toegang tot een grote verzameling “off the shelf” software componenten.

COM is een groot en complex protocol en het vergt normaal veel code om een COM component te bouwen, vaak ondersteund door 'wizards'. Wij laten zien dat hogere orde functies kunnen worden gebruikt om componenten gemakkelijk te kunnen maken zonder ondersteuning van wizards of duplicatie van code. Verder blijkt dat single inheritance interface subtypering kan worden gemodelleerd door parametrisch polymorphisme en zogeheten 'phantom' types. Deze modellering blijkt essentieel om de object geörienteerde natuur van de meeste software componenten te kunnen ondersteunen binnen een functionele taal als Haskell. Zelfs op het meest fundamentele niveau blijken we in staat om vele eigenschappen statisch te kunnen verifiëren met behulp van het rijke type systeem van Haskell: bijvoorbeeld de associatie tussen het type van component instanties en hun klasse, en de associatie tussen virtuele methode tabellen en corresponderende instantie gegevens.

Na het lezen van de voorgaande hoofdstukken over de integratie van Haskell met de imperatieve wereld, kan de lezer zichzelf afvragen of de voordelen wel opwegen tegen de complexiteit. Een potentieel probleem is dat een typische component is ontworpen voor een imperatief model en daardoor zorgt voor een imperatieve stijl van programmeren binnen het declaratieve model van Haskell. In de volgende twee hoofdstukken gaan we hier verder op in. We laten zien dat het mogelijk is om bibliotheken van herbruikbare hogere orde functies te definiëren boven op de imperatieve laag. Deze functies gedragen zich als een domein specifieke taal om een bepaalde verzameling componenten aan te spreken. De stelling van deze hoofdstukken is dat getypeerde hogere orde talen, zoals Haskell, een nieuwe manier van programmeren mogelijk maken waarmee componenten kunnen worden aangestuurd. Een generieke strategie om zulke domein specifieke component talen op te zetten word gegeven aan de hand van een uitgebreid voorbeeld in de context van database servers.

Het werk in de voorgaande hoofdstukken gaf aanleiding to experimentatie met taal uitbreidingen aan Haskell. Helaas vergt Haskell een uitgebreide ondersteuning tijdens executie en compilatie, wat het moeilijk maakt om hiermee te experimenteren. Dit gaf aanleiding tot de ontwikkeling van de 'lazy virtual machine' (LVM). Deze virtuele machine is speciaal opgezet voor executie van niet-stricte hogere orde talen, zoals Haskell. Het doel van de LVM is om een systeem te hebben dat zeer geschikt is om mee te experimenteren door zijn modulaire en uitbreidbare opzet.

We kijken in het bijzonder naar het algehele ontwerp van de LVM instructies, de operationele semantiek, en de vertalings schema's. In plaats van gebruikelijke geoptimaliseerde vertalings schema's, gebruiken we een naïef en simpel schema met een kleine verzameling herschrijfregels op de instructies, die uiteindelijk hetzelde effect bereiken. De korrektheid van de herschrijfregels is relatief gemakkelijk te bewijzen met behulp van de operationele semantiek van de instructies. Een (geoptimaliseerd) vertalingsschema bewijzen is veel moeilijker omdat men de overeenkomst moet aantonen tussen de operationele semantiek van de doeltaal met de gegenereerde instructies. De abstracte machine is sterk gerelateerd aan de mogelijkheden van de huidige hardware. We kunnen daardoor redeneren over implementatietechnieken die normaal alleen informeel beschreven worden, zoals foutafhandeling, het teruggeven van constructoren in registers en de 'black holing' techniek.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE.

1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11