The University of New South Wales

School of Computer Science and Engineering

Binding Haskell to Object-Oriented Component Systems via Reflection

André T. H. Pang

B. Sc (Computer Science & Psychology) Honours Thesis

June 2003

Supervisor: Dr. Manuel Chakravarty
Assessor: Dr. Kai Engelhardt

Abstract

A universal framework is described to interface Haskell to a component system or object-oriented programming language with reflection capabilities. Techniques are shown which enable modelling an object-oriented class hierarchy, including multiple inheritance and class objects, in Haskell. Template Haskell is used as a basis to write an interface generator, which uses the reflective capabilities of the targeted component system to automatically derive and create a Haskell interface for the components' APIs.

A low-level communications library is designed which enables sending messages to components, and a high-level messaging API is layered on top of the library to present a convenient, elegant interface for a Haskell programmer to interact with components. Techniques are given to enable exception marshalling between the component system and Haskell, to perform memory management of foreign components in the Haskell system, and to write a component in Haskell. A Haskell to Objective-C binding named MochΛ has been written, to show that these ideas are practical and can be implemented.

Acknowledgements

The problem with acknowledgements is that

$$\exists a \Rightarrow person(a) \wedge unacknowledged(a)$$

Nevertheless, I am indebted to far too many people to not express not my gratitude at all. This thesis was more than a one-man show.

Immense thanks must go to my comrades in the Programming Languages and Systems research group at UNSW, especially Sean Seefried, Gabi Keller and Don Stewart, who always found the time to help me with obscure questions, on topics ranging from computer science to the meaning of life. (Try as they might, I still believe they're two different things.)

Thanks to all the helpful people in the Haskell community, who produce the wonderful compilers and tools that we use, and all the knowledgeable, friendly folk on the mailing lists and #haskell for answering questions which I had.

To Jeffrey Lim, whose friendship and heart is only rivalled by his incredible programming abilities. Any good knowledge I have about program design, I learnt from you. Thanks.

To my supervisor, Manuel Chakravarty, who introduced me to the wonderful world of functional programming, and guided me through the maze of computer science and University in a way that nobody else could have done—all while being a fantastic friend. I hope this thesis makes you proud.

To my family, who always support me and understand me no matter how much sleep I've not had. Thank you for always being there.

To Michelle, for just being who you are.

Contents

1 Introduction & Background **1**
- 1.1 Haskell . . . 1
- 1.2 Object-Oriented Programming . . . 1
- 1.3 Component Systems . . . 2
- 1.4 Motivation and Overview . . . 3
- 1.5 Overloaded Terminology . . . 4
  - 1.5.1 Classes and Types . . . 4
  - 1.5.2 Type Classes and Overloading . . . 4
  - 1.5.3 Parameterised Types *vs.* Templates . . . 5

2 Haskell vs. Component Systems **6**
- 2.1 Integrating Haskell and Component Systems . . . 6
- 2.2 Modelling an Object-Oriented Class Hierarchy . . . 7
- 2.3 Communication with Components . . . 7
- 2.4 Exception Marshalling . . . 8
- 2.5 Memory Management . . . 8
- 2.6 Summary of problems . . . 8

3 Modelling a Class Hierarchy in Haskell **10**
  - 3.0.1 Phantom types . . . 10
  - 3.0.2 Representing multiple inheritance via type classes . . . 11
  - 3.0.3 Moving on from phantom types . . . 13
- 3.1 Automatic class hierarchy generation . . . 14
  - 3.1.1 Meta-programming the interface generator . . . 15
  - 3.1.2 Autogeneration of Interface Definitions . . . 17
  - 3.1.3 Reflecting on Interface Definitions . . . 17
- 3.2 Upcasting and Downcasting Objects in the Class Hierarchy . . . 18
  - 3.2.1 Upcasting and downcasting with fromx and tox . . . 19
  - 3.2.2 Convenient casting: upcast and downcast . . . 19
  - 3.2.3 Statically type-checking casting . . . 22
  - 3.2.4 Dynamically type-checking casting . . . 22
- 3.3 Class Objects . . . 22
- 3.4 Representing Components and Objects in Haskell . . . 25

4 Component Communication **28**
- 4.1 Sending Messages: the Low-Level Interface . . . 28
  - 4.1.1 The Foreign Function Interface . . . 28
  - 4.1.2 The FFI: Extend or Embrace? . . . 28

4.1.3 C as the Proxy Language 30
4.1.4 Messages, Receivers & Message Expressions 30
4.1.5 The mailman arrives 32
4.2 Creating a friendlier mailperson 33
4.2.1 Setting the Message Arguments 34
4.2.2 Sending Variable Arguments in a Message 34
4.2.3 Overloading the message reply type 40
4.2.4 Implementing and using sendMessage 40
4.3 Direct Messaging: Statically type-checked message sending . . . 42
4.4 Implementing Direct Messaging 43
4.5 Transparent marshalling 49
4.6 Monadic binding and object-oriented syntax 49
4.7 Receiving messages 50

5 Exception Marshalling 54

6 Memory Management 56

7 MochΛ: Haskell & Cocoa 59
7.1 MochΛ 59
7.1.1 The Cocoa classes 59
7.1.2 Communication with Objective-C 60
7.1.3 Building Cocoa applications 60

8 Discussion & Conclusion 61
8.1 Summary of ideas 61
8.2 Conclusion 61
8.3 Future Work 62

A An explanation of getClassObject 63

Chapter 1

Introduction & Background

1.1 Haskell

Haskell is a purely functional programming language. It has many features which make it desirable to use, such as strong, static typing with a type inference engine, parameterised data types, ad-hoc and parametric polymorphism, lazy evaluation, higher-order functions, and built-in memory management. Haskell offers full-fledged IO via monads, and a standardised *Foreign Function Interface* [12] (*FFI*) to interact with the operating system and other languages.

1.2 Object-Oriented Programming

The object-oriented programming paradigm is based around the idea of an *object*. An object is a collection of data and functions which operate on that data, merged into a single entity. The object's internal data (known as *instance variables* or *fields*) can typically only be manipulated through the functions which the object exposes (called *methods*). This ensures that details of an object's implementation are hidden behind the interface offered by the object's methods. Thus, objects provide a means of *data encapsulation*. Calling a method of an object is usually known as *invoking* a method, or *sending a message* to the object (where the method to invoke is part of the message's payload).

Objects are categorised into various *classes*. A class is a specification for what fields and methods an object should contain. Using classes, object-oriented systems offer a feature known as *inheritance*: classes can *inherit* or *subclass* other classes. All the fields and methods of the inherited class (also called the *superclass*) become part of the specification of the *subclass*. Objects which are instantiated from a subclass are treated as belonging to the same category as an object instantiated from the superclass; i.e. from the compiler's point of view, a subclass can masquerade as the same *type* of the superclass (but not vice versa). Type inheritance is more formally known as *subtyping*.

Examples of object-oriented programming languages include C++, Java, Smalltalk, and Objective-C.

1.3 Component Systems

Component systems have been evangelised as a way of building systems through self-contained entities known as *components*. Components, like objects, are required to hide their implementation from their interface, and also offer a standard protocol for performing a component's actions and communicating between components.

Components therefore share much similarity with object-oriented programming. Both systems are based around the idea of self-contained entities with data encapsulation, and both systems offer methods that can be invoked by other objects and components, with a standard way of invoking those services. Examples of component systems include CORBA, (D)COM and Enterprise Javabeans (J2EE).

While components and objects are similar, they usually differ in two significant respects:

- Component systems are required to offer ways to locate and communicate with other components in the system. For example, in CORBA, a component known as an Object Request Broker (ORB) can locate a component which performs the services that another component requires, and establish communication channels between those two services.
- Objects are typically much more fine-grained than components. The range of services or functionality that an object offers is usually smaller than a component.

Even with these differences, it is common to see object-oriented programming languages being used as a basis to write components because it is easy to create an object which functions as a component. This is usually achieved by using a *software adapter*, which transforms the component system's method calling convention into one which the object can understand.

Indeed, sometimes the line between object and component systems are even more blurred than outlined above. A component system can be built from a programming language feature commonly known as *reflection* or *introspection*. Reflection enables programs, at run-time, to query detailed information about objects, such as what class and superclass an object belongs to and what methods an object offers. Reflection also usually allows a running program to create instances of or invoke methods on objects, even if the appropriate class, object and method names are not known until runtime. Reflection can therefore serve as a foundation for building a facility to locate other objects in the system, so that a programming language itself can, with little effort, be extended to become a complete component system. Enterprise Javabeans is an example of a component system which is implemented in this manner.

For the purposes of this thesis, object-oriented languages which have reflective capabilities will be treated in the same manner as component systems. Both systems are similar enough that they present the same challenges in integrating them with Haskell.

1.4 Motivation and Overview

Haskell has nowhere near the popularity of object-oriented programming languages and component systems. Indeed, object-oriented languages and components serve as the foundation for many modern systems, and many frameworks, libraries and platforms use object-oriented class hierarchies to structure themselves. There are a number of benefits to both the Haskell and object-oriented worlds if a binding could be created which enables Haskell to communicate with and act as an object or component:

- Many important APIs are object-oriented. For example, the Objective-C Cocoa framework, the Microsoft Foundation Classes, and the GTK+/QT toolkits are standard APIs which are used to develop GUI applications on the Mac OS X, Windows and UNIX platforms. All these APIs are object-oriented, so a way for Haskell to use these frameworks in a convenient, elegant manner would enable Haskell to build sophisticated GUI applications.

- Component systems such as CORBA, (D)COM/OLE, Enterprise Javabeans, Bonobo and KParts are evolving into *dé facto* ways to provide extra functionality to their respective environments. Each component system usually provides multiple language bindings, but a Haskell binding to such systems is rare because of the difficulties faced in bridging Haskell to the component systems.

- Enabling Haskell to communicate with component systems opens Haskell to a far greater audience. Authors of components and applications may turn to Haskell to build their systems if they understand that Haskell can easily interplay with component and object-oriented systems.

This thesis tackles the challenging problem of interfacing Haskell with object-oriented and component systems. There are many problems to be solved: how can Haskell model an object-oriented class hierarchy, which includes modelling features such as class inheritance, and multiple inheritance? Can such a model be designed while taking full advantage of Haskell's strong, static typing whenever possible? How can Haskell invoke methods on objects, and send messages to components, again while retaining strong, static typing where appropriate? How can a convenient messaging API be written, so that Haskell programmers who wish to interact with components can do so, in a manner as convenient as one of the component system's natively supported programming languages? How can exceptions be marshalled from the component system to the Haskell environment, and how can Haskell's memory management be extended to support management on foreign components?

This challenges are all defined and tackled in this thesis, and not only have solutions been found to many of the problems, but a Haskell to Objective-C language binding has been written to prove that these ideas are practical and work. This language binding, MochΛ, allows Haskell to interact with the powerful *Cocoa* framework written in Objective-C. Cocoa is the primary framework used to develop applications on Mac OS X, and MochΛ shows that binding Haskell to such a framework has great benefits. By using MochΛ, Haskell can be used to build full-blown graphical applications on Mac OS X, and Haskell

can also be used to create objects which interact with the rest of Cocoa and the operating system.

1.5 Overloaded Terminology

The functional programming and object-oriented programming disciplines are both rich with their own terminology. Some words have conflicting meanings in the two domains: in particular, the definitions of *class*, *template* and *overloading* should be clarified.

1.5.1 Classes and Types

A *class* in object-oriented terminology is a specification of what methods and instance variables that a concrete instantiation of that class—an *object*—should have. Classes can be thought of as types, and class names are usually used as types in object-oriented languages to permit the compiler to type-check expressions. Indeed, the analogy to a class in the Haskell world is the Haskell *data type* (or simply *type*), with the notable exception that Haskell does not permit subtyping.

Classes in Haskell are different entities from classes in object-oriented programming; Haskell's classes, more formally known as *type classes*, serve as a form of *overloading* (also known as *ad-hoc polymorphism*).

1.5.2 Type Classes and Overloading

Type classes allow for a function to have different implementations, depending on the type(s) of the arguments passed to the function. For example, the Haskell function show can be used to display any data type as a string, as long as the data type to be displayed is an *instance* of (belongs to) the Show type class, which requires that the data type has implemented an appropriate show function.

It is important to note that the type signature for the functions in the type class remains the same no matter how many implementations it has. This is achieved by using *constrained type variables* in the type signature for the overloaded function. The constrained type variables represent arguments of any type which are instances of the type class. For example, the type signature for the Haskell $>$ (greater than) function is **Ord** a $\Rightarrow$ a $\rightarrow$ a $\rightarrow$ **Bool**, which signifies that $>$ takes in two arguments of type a which must belong to the type class Ord, and outputs a value of type Bool. A compile-time error will be raised by the type checker if the constraints on the type variables are not met.

Type classes can also be *subclassed*, so that any data type which belongs to the subclass also belongs to the superclass. For example, the Ord type class specifies types which have the property of being ordered, so that one can use the comparison functions $<$ and $>$ on those types to see which of the arguments is greater or lesser. However, for a type to be ordered, it must also be possible to define whether two values are equal: thus, the Ord type class can subclass the Eq type class, so that any instances of Ord are also required to be instances of Eq.

Haskell's type classes are most analogous to *operator overloading* in C++, although Haskell's overloading is not restricted to the standard mathematical

operators. Type classes allow a programmer to define new functions (such as show) which can be overloaded, and specify which implementation of those functions should be chosen for particular types.

Note that this definition of overloading in Haskell is very different to the typical definition of overloading in the object-oriented world, which often refers to *method overloading*. Method overloading enables a method to have multiple type signatures, and which method is called is dependent on which type signature matches the function call.

1.5.3 Parameterised Types *vs.* Templates

Haskell also supports the notion of *parameterised types*: these are type definitions which contain a *type variable*. When the type is instantiated, the type variable is filled in with a concrete Haskell type, such as an Int or a String. For example, a pointer in Haskell has the type Ptr a; when an actual pointer is created, the type variable a must be instantiated with a concrete type, so that a pointer to an Int will have the type Ptr Int, and a pointer to a Float will have the type Ptr Float.

The object-oriented programming language C++ also supports parameterised types through a mechanism called *templates*. As well as parameterised types, C++ templates are used as a form of *compile-time meta-programming*. At the time of compilation, the template generates code as its output which becomes part of the final program.

The Glasgow Haskell Compiler (*GHC*) also offers meta-programming facilities via *Template Haskell*. What is important to note here is that C++ templates, while offering meta-programming, are focused on providing parameterised types to the language. Template Haskell, on the other hand, is focused purely on meta-programming. Parameterised types and meta-programming are two distinctly different features in Haskell, whereas they are tightly coupled in C++.

Chapter 2

Haskell vs. Component Systems

2.1 Integrating Haskell and Component Systems

The challenge is to allow Haskell to communicate with component systems: Haskell programs must be able to locate and identify components, and send and receive messages to and from those components, just as any other component would be able to do. This must be done while still maintaining an *elegant* interface on the Haskell side, so that its important features such as strong type checking and its purely functional nature can still be used while interacting with components. The interface must also be *convenient* to use, so that using it in Haskell is at least as simple as using one of the component system's primary supported languages.

Ideally, Haskell should act as a first-class citizen in the component system, so that it is possible to write components or objects in Haskell and interface those Haskell-written components with the rest of the system. Again, this must be done by using a convenient, elegant Haskell interface to write the component.

Given these defined goals of elegance and convenience, it is possible to divide the challenge of integrating Haskell with component systems into two parts:

- allowing Haskell to communicate with other components or objects, and
- presenting an elegant, convenient interface on the Haskell side, so that the component system's concepts such as class inheritance can be understood and modelled on the Haskell side—with full support from the type system where appropriate.

At first glance, it may seem that cleanly separating the problem into these two parts is a wise idea: it should be possible to tackle each sub-problem individually. However, it may be difficult to layer a higher-level, more convenient API on top of a more primitive communications API if the right foundations are not present in the primitive API.

For example, it is tempting to use a single type to in Haskell to represent any object or component. This may enable a simpler design for the communications library. However, this design decision may cause problems when attempting to

model the object-oriented class hierarchy in Haskell. In particular, it will not be possible for the type system to distinguish between different types of objects if a single type is used to represent all objects.

2.2 Modelling an Object-Oriented Class Hierarchy

Many component systems borrow concepts from object-oriented systems, such as using object-oriented classes to represent components of different types. Some component systems even support the use of class inheritance, so that a component's interface or implementation can be based upon another component's. If a component system's class hierarchy can be modelled via Haskell types in an isomorphic manner, this would allow Haskell to perform static, strong type checking on code which involves components.

Unfortunately, numerous problems arise from the mismatch between Haskell's type system and the type systems typically found in object-oriented languages:

- Haskell lacks a defining feature of object-oriented systems: it does not have subtyping (inheritance).
- A subclass may inherit from multiple superclasses. It may only be allowed to inherit the interface of those superclasses (e.g. Java, C#, Objective-C), or it may inherit both the interface and implementations from its superclasses (e.g. C++).
- In some object-oriented systems (e.g. Java, Smalltalk, Objective-C), defining a class has a dual purpose: it creates a new type with the same name of that class, and it also creates an instantiated object known as a *class object*. This instantiated object is typically used as a *factory object*: to create an instance object which belongs to a particular class, you invoke a method of that class's factory object (e.g. often named alloc or new) which creates the instance for you.

The aim is to model these features of an object-oriented class hierarchy in Haskell, and do so in a manner which requires minimum extension to the language. The type checker must still be able to infer the types of objects and accurately type check code which uses objects—including any functions responsible for communicating with the component system.

There is also the non-trivial problem of performing the actual modelling. It is unacceptable for a programmer to tediously write the declarations required to encode a class hierarchy: the Java API's prototype declarations span hundreds of pages. Automating the transformation of the class hierarchy into Haskell must be possible.

2.3 Communication with Components

For Haskell to *communicate* with components, it must be able to send messages to and receive *messages* from a component. There are several problems involved with communication:

- How is it possible to build a message in Haskell which can then be sent to other components?
- How do we ensure that the information in each message follows the component's messaging protocol; e.g. that each item in the message has the proper type and alignment required by the receiving component?
- How can Haskell perform the actual operation of sending and receiving messages to and from a component?
- If sending a message to a component results in the component *replying* to us with a return value, how can the replied data, and its type, be obtained?
- Is it possible for the communications interface to *marshal* values between the Haskell environment and the component system, so that native Haskell data types can be understood by the component system and vice versa?
- How can Haskell *receive* messages from the component system, i.e. how can a component be written in Haskell?

2.4 Exception Marshalling

Exceptions are pre-defined events in program which indicate exceptional conditions, such as a division by zero error. They can potentially change the flow of control in a program. When Haskell is interacting with a component system, an exception may be *thrown* (or *raised*) in the component system, which should then be *caught* and *marshalled* to the Haskell environment, so that Haskell code can take an appropriate action to handle the exceptional condition. How can exceptions be marshalled to the Haskell environment, and how can a communications framework be designed to correctly catch and marshal exceptions?

2.5 Memory Management

Haskell features automatic memory management by using a *garbage collector*, so that a Haskell programmer never has to worry about allocating and deallocating memory for expressions. If Haskell is to interact with a component system, the automatic memory management that a programmer is used to in the Haskell environment should also be able to manage any foreign components. How can Haskell's memory management system be extended to support foreign components? It is always possible or necessary to perform such memory management when interfacing with component systems?

2.6 Summary of problems

There are a number of challenges involved in interfacing Haskell to a component system. It may be simple to enable Haskell to communicate with a component system, but presenting a convenient, elegant Haskell API for such communication is far harder. The main problem areas include:

- *Modelling an object-oriented class hierarchy* in Haskell while retaining strong, static typing where appropriate. This may include modelling subtyping, multiple inheritance and class objects, and also involves generating the a Haskell API to use any desired components.
- *Communicating with components* by building and sending messages and retrieving message replies—with full type checking and type inference. Writing components in Haskell and enabling those components to receive messages from the component system should also be possible.
- *Exception marshalling*: if sending a message to the component system results in an exception being thrown, it should be possible for the Haskell environment to catch those exceptions and take appropriate action.
- Automatic *memory management* of any components which the Haskell environment interacts with.

Chapter 3

Modelling a Class Hierarchy in Haskell

For Haskell to communicate with component systems, it would be ideal if there was a way to model the component system's class hierarchy and types in Haskell. Without Haskell knowing about the component system's class hierarchy, two of Haskell's most defining features—strong typing and type inference—would not be used to their full potential on Haskell code which uses with components.

Encoding a class hierarchy in Haskell is a non-trivial problem due to the mismatch between its type system and the type systems typically used by component systems and object-oriented languages. Nevertheless, there are several techniques that can be used which enable Haskell's type system to accurately model the typing used by object-oriented languages while still allowing strong typing and type inference.

3.0.1 Phantom types

One method of modelling an object-oriented class hierarchy with inheritance is using *phantom types*: a phantom type is a parameterised type where the parameter does not occur in the type constructor definition; in **data** Foo a = Foo, for example, a is a phantom type. The phantom type can be used by the type system to guarantee that any instances of that type are well-formed. In this case, the type's parameter can be used to model subtyping, to a limited extent, in the type system. For example, Listing 3.1 shows a Java class declaration for two classes Super and Sub; a Haskell encoding of the class hierarchy using phantom types is shown in Listing 3.2.

Listing 3.1: A superclass and subclass in Java

```
class Super { ... }
class Sub extends Super { ... }
```

The type Super () can then be used whenever an instance of the Super object-oriented class is required, but none of its subclasses would be accepted if this type is used. To accept any instance of the Super class, including any of its

Listing 3.2: A superclass and subclass in Haskell, modelled with phantom types

```
data SuperT a = SuperT
type Super a = SuperT a

data SubT a = SubT
type Sub a = Super (SubT a)
```

subclasses, the type Super a is used. This technique has been used in a number of previous approaches to bind Haskell to component systems [1–3]. Its main drawcard is that it was one of the first ways to model class inheritance in Haskell, while giving the type checker enough information about the types so that it could correctly reject cases where the type of an expression is not a subclass of the superclass which was required. This use of phantom types therefore enabled the type system to correctly, statically and strongly type check class inheritance trees.

However, using phantom types has two major disadvantages:

- *Long type names.* For APIs which use inheritance extensively, type names can become very long, which can result in the type checker outputting error messages which are hard to decipher.
- *Single-inheritance only.* Phantom types cannot model an object-oriented class hierarchy which uses multiple inheritance, regardless of whether the multiple inheritance only allows interface inheritance (e.g. Java, C#) or both interface and implementation inheritance (e.g. C++). Neither of these scenarios can be modelled using phantom types, since a subclass can only inherit from one superclass.

Since many component systems use multiple interface inheritance, phantom types cannot be the only method of modelling inheritance in Haskell.

3.0.2 Representing multiple inheritance via type classes

In Lambada [2], a Haskell to Java language binding, Finne and Meijer show that type classes can be used to augment phantom types, so that multiple inheritance in object-oriented systems can be properly modelled in Haskell. Thus, if the class Sub inherits from two Java interfaces AnInterface and AnotherInterface (as shown in Listing 3.3 and visualised in Figure 3.1), one would simply augment the Haskell code in Listing 3.2 with the code shown in Listing 3.4 to allow the type system to represent this inheritance.

A method which is part of the interface declaration simply uses the type variable constraint as part of its type signature. The function will then accept objects only if they implement (inherit from) that interface. For example, the type signature for a function foo which requires a parameter that implements AnInterface would have the type signature foo :: AnInterface a ⇒ a → .... Thus, both phantom types and type classes are used here to model a class hierarchy. While this technique still suffers from the problems of hard-to-read type errors and long type names, at least it is possible to model multiple inheritance, which is a big step forward.

Listing 3.3: Java class hierarchy with multiple inheritance

```
class Super { ... }

interface AnInterface { ... }

interface AnotherInterface { ... }

class Sub extends Super
    implements AnInterface, AnotherInterface { ... }
```

Figure 3.1: Visualisation of a Java class hierarchy with multiple inheritance. Ellipses with solid outlines represent classes, and ellipses with dashed outlines represent interfaces.

Listing 3.4: Phantom types augmented by type classes can model multiple inheritance

```
class AnInterface i
class AnotherInterface i

instance AnInterface (Sub a)
instance AnotherInterface (Sub a)
```

3.0.3 Moving on from phantom types

Shields and Peyton Jones [4] go one step further with type classes: they propose a way to use only type classes and types to model an object-oriented class hierarchy, thus not requiring phantom types at all. To model the Java class hierarchy presented in Listing 3.3 (page 12) (visualised in Figure 3.1 on page 12), the Haskell code in Listing 3.5 (visualised in Figure 3.2) can be used.

Listing 3.5: Using only type classes to model multiple inheritance

```
class Super c
data SuperInstance −− some opaque type
instance SuperInstance Super

class AnInterface c
data AnInterfaceInstance
instance AnInterface AnInterfaceInstance

class AnotherInterface c
data AnotherInterfaceInstance
instance AnotherInterface AnotherInterface

class Sub
data SubInstance
instance Super SubInstance
instance AnInterface SubInstance
instance AnotherInterface SubInstance
```

The type signature for a function foo which requires an argument that is an exact instance of the class Super would look like foo :: SuperInstance → .... This would not allow any subclasses of Super to be used as an argument to the function. To allow subclasses of Super to be passed to the function, the type class Super is used as a type variable constraint. This is the same technique used by Finne and Meijer [2] to implement Java interface inheritance in Lambada: the function's type signature would be foo :: Super a ⇒ a → ....

This use of only type classes to encode inheritance has several advantages over using phantom types:

- The type checker can give more readable error messages if it detects a type error.
- It is more elegant than using phantom types, and is more easily understood. Haskell programmers already know that type classes can be subclassed: it is a small knowledge leap for them to understand that the same system can also be used to model object-oriented subclassing, compared to explaining how phantom types are used to model subclassing. (Why learn two methodologies when one will suffice?)

While this scheme still has limitations (e.g. it cannot model *covariant result types*), it is sufficient to model the typical class inheritance hierarchies used by the vast majority of component systems.

Figure 3.2: Visualisation of a class hierarchy with multiple inheritance using Haskell type classes. Ellipses with solid outlines represent data types, and ellipses with dashed outlines represent type classes.

For all its advantages, using type classes to model inheritance has one drawback compared with phantom types: the type signature of a function that uses subclasses isn't quite as elegant as its equivalent using phantom types. If phantom types are used, one could write `foo :: Super a → ...` for a type signature that accepts an argument which belongs to a class or subclass of `Super`; instead of that, the type signature is now `foo :: Super a ⇒ a → ...`, which is more cumbersome.

For binding Haskell to component systems, the approach advocated by Shields and Peyton Jones [4]—using a type class & data type per object-oriented class—is sufficient to model an object-oriented class hierarchy, and works well. It is capable of modelling multiple inheritance without difficulty, and is also reasonably elegant.

3.1 Automatic class hierarchy generation

The majority of language binding tools come with a program to automate the process of making the desired component system's API functions available to the language we wish to bind to. This program is called an *interface generator*.

Current Haskell language binding tools[1] typically use these to model the target API in Haskell. For example, the Gtk+HS [7] project, which binds Haskell to the open-source GUI toolkit GTK+, relies on the C→Haskell [5] interface generator to model GTK+'s object-oriented class hierarchy in Haskell. C→Haskell acts a as a pre-processor: it reads in C header files along with specially marked up Haskell modules, and writes out normal Haskell modules which then contain the Haskell mapping of the C API. While this is a perfectly fine way of producing an API binding to a foreign language, it would be more convenient this pre-processing stage can be eliminated.

[1] http://www.haskell.org/libraries/#interfacing

3.1.1 Meta-programming the interface generator

The introduction of version 6.0 of the Glorious Glasgow Haskell Compiler presents a powerful new idea which we can use to eliminate this pre-processing stage: *Template Haskell*. Template Haskell is a form of *compile-time meta-programming*. Meta-programming is writing a program which generates programs; compile-time meta-programming allows this code generation to happen at compile-time. As compilation happens, the compiler can *execute* user-written code, to produce the final code that is then placed into the resulting program. Template Haskell can be thought of as a powerful macro system, or, to quote from the original Template Haskell paper by Sheard and Peyton Jones [8], Template Haskell "allow[s] programmers to *compute* part of their program rather than *write* them, and to do so seamlessly and conveniently." (Emphasis in original.)

Template Haskell gives the programmer data types which represent Haskell syntax; for example, a function may output a value of a type Exp, which represents a Haskell expression, or it may output a value of type Dec, which represents a Haskell declaration. Values of these syntax data types are called *templates*, and can be executed at compile-time at various *splice points* in the program, so that template expressions and template declarations are evaluated at these splice points, just as if the spliced code were written there by the programmer instead of being generated by the templates. Since functions which generate templates are treated as normal functions, they can be called with parameters in splice points and be given customised data structures just as any normal function would. This enables generation of different templates, depending on what parameters were given to the template-generating function.

Listing 3.6 shows how to write the classic C printf function in Template Haskell, and how to use the printf template.

While Template Haskell has opened up many possibilities to the Haskell world, let us examine just one application of it: its role as a possible automated interface generator, thus fulfilling the same role as tools such as C→Haskell, Green Card[2], or HaskellDirect[3]. One can use Template Haskell as a sophisticated pre-processing system, but instead of generating text files that contain Haskell code—as most interface generators do—the Haskell code is generated directly, in the form of templates. By calling template functions with the appropriate parameters, the resulting code produced by the templates can model the target API. Since the compiler executes the template code at compile-time, not only can templtes perform the interface generation, but the interface generation is far more convenient since it is integrated into the compilation process. There is no separate pre-processor step required.

Of course, this simple means of using Template Haskell as an interface generator still requires the programmer to tell the template functions which interface APIs to generate; e.g. a Template Haskell-based IDL compiler would still require an IDL specification in some form. So far, the only difference that Template Haskell has made is saving a step in the build process of an application, by eliminating the need to execute a pre-processor before compilation.

[2] http://www.haskell.org/greencard/
[3] http://www.haskell.org/hdirect/

Listing 3.6: Implementing and using printf in Template Haskell

```
-- works with %d and %s
data Format = D | S | Literal String
    deriving (Show)

parse :: String → [Format]
parse "" = []
parse ('%':'d':xs) = D:(parse xs)
parse ('%':'s':xs) = S:(parse xs)
parse s = Literal upToAPercent:(parse afterPercent)
    where
        upToAPercent = takeWhile (≠ '%') s
        afterPercent = dropWhile (≠ '%') s

gen :: [Format] → Expr → Expr
gen [] x = x
gen (D:xs) x = [| \n → $(gen xs [| $x ++ show n |]) |]
gen (S:xs) x = [| \s → $(gen xs [| $x ++ s |]) |]
gen (Literal s:xs) x = gen xs [| $x ++ $(string s) |]

printf :: String → Expr
printf s = gen (parse s) [| "" |]

-- using printf
printfTest = do
    $(printf "A number %d and a string %s.\n") 69 "foo"
    $(printf "Just a number %d") 7
```

3.1.2 Autogeneration of Interface Definitions

> The three principle virtues of a programmer are Laziness, Impatience, and Hubris. — *Larry Wall, "Programming Perl"*

It would be very convenient if a template-based interface generator can automatically find the interfaces presented by the target API. If this is possible, then the task of mapping a component system's API in Haskell can be completely automated.

In fact, automating the search for interface definitions in Template Haskell would be exactly the same as automating it if a standalone tool such as C→Haskellwas being used, because arbitrary IO can be performed in Template Haskell via the *quotation IO monad*. While templates are being spliced in by the compiler, files or network sockets can be opened and even external programs may be executed—all during compilation!

Ian Lynagh uses this technique to great effect when creating a Haskell interface to the UNIX `curses` library [9]. He first uses Template Haskell to splice in declarations of FFI foreign import statements automatically into a module, which eliminates the need for the programmer to manually write those declarations. However, Lynagh then proceeds to build the C wrapper functions required by the just-declared FFI imports as Strings, writes out those C wrapper functions to C source and header files using the quotation IO monad, and then calls `gcc` *inside the splice* to compile the C wrapper functions which were just written!

So, by using Template Haskell and the quotation IO monad, Lynagh manages to completely automate building the `curses` binding. With no preparation beforehand, a programmer who wishes to access the `curses` API simply has to put in one splice in his program, and the templates will generate the code to build the binding, and expose the `curses` API in Haskell to the programmer—in one line of code. It would be easy to modify Lynagh's `curses` example to suit various other APIs: interface definitions could be generated from header files, XML definitions of the API or *JavaDoc* documentation. This shows that while Template Haskell is certainly useful as a simple replacement for an interface generator, it wields far more power when it can be used to automatically find the interface information as well.

The convenience that this brings to the programmer should not be underestimated. Arguably, a large reason why Haskell bindings to foreign APIs are not used more often is that programmers simply don't want to go to the trouble of building an API binding. At the very minimum, the interface generator must be built. Often, the programmer then has to find the interface definitions, and give the interface generator those definitions using a special syntax, so that the final API binding can be produced. Template Haskell eliminates those steps: instead, the programmer simply has to write a few extra lines of code to splice in the templates, which do all the magic work behind-the-scenes during compilation. Hopefully, programmers will take advantage of more API bindings if they are simpler to use.

3.1.3 Reflecting on Interface Definitions

Besides using the quotation IO monad to automatically search for and generate interface definitions, Template Haskell can also execute FFI functions in a splice, and let the FFI functions return the information necessary for the generation of

interface definitions. While this is not as flexible as performing arbitrary IO to derive interfaces, it is also sufficient (and preferred) when interfacing to component systems or object-oriented systems with reflection capabilities. Template Haskell simply FFI-calls the component system's reflective functions to locate components in the system and derive their interfaces. This has a number of advantages over using the quotation IO monad to generate the interfaces:

- Since the component system's reflective functions are invoked directly, it is guaranteed that the interface definitions given to the generator are *accurate*.
- It is *safer* than performing IO: the programmer is assured that the Haskell template code cannot execute arbitrary programs, unlink files, or write politically incorrect messages all over the user's terminal.

Thus, a target API's reflective capabilities can be used to automatically derive the information necessary to generate the interfaces. Again, all that may be necessary on the user's part is writing a few splices in their code, to direct the interface generator will do its work at compilation time using Template Haskell.

3.2 Upcasting and Downcasting Objects in the Class Hierarchy

> No-one likes to be typecast, anyway. — *Larry Wall*

All object-oriented systems have *container objects*: these are objects which, as the name implies, can contain other objects. Canonical examples of container objects include lists, dictionaries and finite maps; other not so immediately obvious examples include a GUI object-oriented toolkit, using container classes such as a GUI window which can contain many other types of GUI objects.

When the contained object is *retrieved* from the container object, there are two possibilities: either the retrieved object will be of the exact same type that it was before it was inserted into the container, or it may be a *supertype* of the original object[4]. To obtain the original type of the object, one must *cast* (or *coerce*) the retrieved object to its original type. Casting from a supertype to one of its subtypes is more specifically known as *downcasting*; casting from a subtype to a supertype is called *upcasting*.

Since upcasting and downcasting is a necessary operation in an object-oriented class hierarchy, it is essential to implement those operations to properly model an object-oriented class hierarchy in Haskell.

To provide an example class hierarchy, recall the Java code in Listing 3.3 (page 12), visualised in Figure 3.1 (page 12). Contrast the code with the Haskell equivalent in Listing 3.5 (page 13), visualised in Figure 3.2 (page 14). In this small class hierarchy, there are six possible casting operations which can be performed:

1. an upcast from Sub to Super,

[4] If a type α is a subtype of another type (i.e. it inherits from) β, then β is a supertype of α.

2. a downcast from Super to Sub[5],
3. an upcast from Sub to AnInterface,
4. a downcast from AnInterface to Sub[5],
5. an upcast from Sub to AnotherInterface,
6. a downcast from AnotherInterface to Sub[5],

Note that the names of the classes used here (such as Sub and AnInterface) correspond to the names of the classes given in the object-oriented Java class hierarchy. The corresponding casts in the Haskell class hierarchy would be upcasting and downcasting between SubInstance, SuperInstance, AnInterfaceInstance, and AnotherInterfaceInstance.

The problem is how to provide such upcasting and downcasting in Haskell. Haskell does not innately provide any such casting operations in the base language: there is no equivalent to C's (**int** $*$) operator in Haskell to, e.g. cast the result of a malloc expression from a **void** $*$ to an **int** $*$. How, then, can upcasting and downcasting be performed in Haskell?

3.2.1 Upcasting and downcasting with fromx and tox

One solution is to require any instances of the type class Super to implement the two functions shown in Listing 3.7: toSuperInstance and fromSuperInstance.

The function toSuperInstance will upcast an expression of any type which belongs to the type class Super to the SuperInstance data type: Listing 3.8 demonstrates how it can be used to perform upcasting.

The function fromSuperInstance is slightly more complex; it will cast an expression of the SuperInstance type to an *ambiguous type variable* s. Listing 3.7 shows that the type signature of fromSuperInstance is :: SuperInstance $\rightarrow$ s, which means that the final type of fromSuperInstance must be explicitly specified, or *fixed*, for the type system to determine exactly which implementation of fromSuperInstance to call. Listing 3.8 shows how to use fromSuperInstance to perform downcasting, and Figure 3.3 visualises how the downcast is performed.

This use of from*x* and to*x* functions to perform type conversion is not a novel idea: the Haskell 98 standard uses this exact technique by defining the functions fromInteger and fromInt in the Num type class.

The interface generator C$\rightarrow$Haskell [5] also uses this technique. It provides a feature named *class hooks*, which generates two functions per object-oriented class. These two functions are isomorphic to the from*x* and to*x* functions described in this section.

3.2.2 Convenient casting: upcast and downcast

While the respective from*x* and to*x* functions can perform the casting, Listing 3.8 (page 20) shows that they can be confusing to use because of the syntax difference between upcasting and downcasting. While upcasting uses the reasoanbly straightforward syntax toSuperInstance sub, downcasting is more cumbersome: fromSuperInstance super :: SubInstance. Moreover, the inconsistent

[5]The downcast assumes that the object being casted is actually of the type Sub.

Listing 3.7: Up/downcasting functionality for the Super object-oriented class

```
class Super s where
    -- give the type system an assertion that a expression foo of type
    -- 's' can be 'cast' to the SuperInstance type, by _fixing_ the
    -- type variable 's' with "foo :: SuperInstance"
    toSuperInstance :: s → SuperInstance
    -- 'cast' any instances of Super, from the instance type to an
    -- ambiguous type variable which belongs to the type class
    fromSuperInstance :: SuperInstance → s

data SuperInstance = SuperInstance -- some opaque data type
data SubInstance = SubInstance

instance Super SuperInstance where
    -- toSuperInstance :: SuperInstance → SuperInstance
    toSuperInstance x = x -- or also "toSuperInstance = id"
    -- fromSuperInstance :: SuperInstance → SuperInstance
    fromSuperInstance x = x -- or also "fromSuperInstance = id"

instance Super SubInstance where
    -- toSuperInstance :: SubInstance → SuperInstance
    toSuperInstance (SubInstance) = SuperInstance
    -- fromSuperInstance :: SuperInstance → SubInstance
    fromSuperInstance (SuperInstance) = SubInstance
```

Listing 3.8: Using fromSuperInstance and toSuperInstance to perform casting

```
-- an instance of super; we only care about the type for this example,
-- the value doesn't matter
super :: SuperInstance

-- downcasting from the 'super' type to the 'sub' type
sub = fromSuperInstance super :: SubInstance

-- upcasting from the 'sub' type to the 'super' type
super' = toSuperInstance sub
```

Figure 3.3: Visualisation of downcasting from SuperInstance to SubInstance, in a Haskell class hierarchy

syntax between upcasting and downcasting is rather unfriendly to a programmer's memory.

Casting should use a convenient, consistent syntax: `downcast super :: SubInstance` for downcasting, and `upcast sub :: SuperInstance` for upcasting. It is possible to do this by using *multi-parameter type classes*: a Haskell 98 extension which, as the name indicates, permits more than one parameter for a type class. Listing 3.9 shows an implementation of upcast and downcast using multi-parameter type classes.

Listing 3.9: The downcast and upcast functions, implemented using multi-parameter type classes

```
class Cast sub super where
    upcast :: sub → super
    downcast :: super → sub

data SuperInstance = SuperInstance -- some opaque data type
data SubInstance = SubInstance

instance Cast SubInstance SuperInstance where
    upcast SubInstance = SuperInstance
    downcast SuperInstance = SubInstance
```

One interesting point about using a dedicated Cast type class to implement the upcast and downcast functions is that the casting function is completely independent of objects. An object-oriented class hierarchy is not required to use these casting functions.

There is no reason why a more general cast function, which performs either an upcast or downcast, cannot be implemented. Indeed, this more closely mirrors the usage of the cast operation in C, C++ and Java. However, the definition

of a Cast type class which requires both upcast and downcast functions ensures that it is impossible to implement an upcast function without implementing its corresponding downcast function.

3.2.3 Statically type-checking casting

By listing valid casts as instances of the Cast type class, the type checker is made aware of all possible casting operations, so all casts are statically type-checked to see if they are valid. One cannot upcast from Foo to Bar if there has been no declaration that Foo Bar is an instance of the Cast.

3.2.4 Dynamically type-checking casting

Many component systems allow for dynamically typed objects, which implies that there may be objects used whose types are not known at compile-time. It is even possible for the component system to demand-load components at runtime, send messages to those components, and receive a message reply which is a ambiguous object whose true type is unknown. However, according to knowledge gained at compile time about the system, this ambiguous object's type *should* be a subtype of AnInterface. Casting these objects to an interface[6] subtype is common practice in many component systems.

Component systems and object-oriented programming languages therefore support and use *dynamic casts* extensively. As such, convenient support for dynamic casts must be made available in Haskell. While implementing dynamic casts varies from one component system to another, it is possible to provide a universal approach for their implementation.

A dynamic cast is simply a type check performed at runtime by the component system. Therefore, to implement dynamic casts, the upcast and downcast functions call a function provided by the component system (via the FFI), which verifies that the cast can be performed. An appropriate action—such as throwing a Haskell exception—can be taken if the cast fails; otherwise, the dynamic cast proceeds normally.

3.3 Class Objects

In some object-oriented languages, defining a new class results in not only a new type, but also a new *class object*. Class objects are concrete representations of that class, and are typically used as *factory objects* which produce new instances of the class on request. Many object-oriented languages which use reflection have class objects, e.g. Smalltalk, Objective-C and Java[7]. To properly model these object-oriented systems' class hierarchies, Haskell must support the use of class objects.

Representing a class object in Haskell can be performed via a function which returns the class object, by calling the appropriate methods in the target object-oriented language. In addition, the Haskell class hierarchy model should be

[6] 'Interface', as used here, is equivalent to the Java definition of the term: i.e. a class which only specifies type signatures for its methods, and possesses no function implementations.

[7] Java hides the use of the class object as a factory, by having an explicit new operator. In contrast, Objective-C requires that the programmer directly invoke a method on the class object to produce an instance object

Figure 3.4: Visualisation of a Haskell class hierarchy which includes class objects. Ellipses with solid outlines represent data types, and ellipses with dashed outlines represent type classes.

extended to support two different inheritance trees: one inheritance tree for class instances, and another inheritance tree for class objects. Since there are now two trees, it is also advisable to provide two new data types which act as the *root* of the instance and class object trees. Listing 3.10 shows how such a mapping can be achieved; for a visual perspective on the class hierarchy, see Figure 3.4.

A more elegant way to retrieve class objects

Even though this class hierarchy model enables retrieving class objects, it is still slightly cumbersome to use getClassObjectFromName for this task; writing getClassObjectFromName "Sub" will not work since the final type of that expression is still a type variable which may be any class object.

It would be preferable to have a dedicated getClassObject function which only needs to be given a type signature to determine which class object to retrieve, e.g. getClassObject :: SubClassObject. Listing 3.11 shows how using a recursive **let** definition can be used to write such a getClassObject function. Since the definition is of getClassObject is not straightforward to understand, Appendix A (page 63) provides an explanation of how the function works.

Listing 3.11 also provides two functions which represent class objects: _Sub_ and _Super_. The rest of this thesis will assume that a function with a *_classname_* naming scheme represents a class object.

Metaclass objects

Some object-oriented languages also have *meta-class objects* (and possibly even *meta-meta-class objects*), which serve as the class objects of the class objects. It is straightforward (if not a little bit tedious) to provide three or more inheritance trees to represent such (meta-)meta-class objects, and to implement a similar

Listing 3.10: Modelling a class hierarchy which includes class objects

```
class Object o
data UniversalObject -- can represent any object
instance Object UniversalObject

-- the inheritance tree for instances
class Object i => Instance i
data InstanceObject -- can represent any instance object
instance Object InstanceObject
instance Instance InstanceObject

class Instance s => Super s
data SuperInstance -- some opaque type
instance Object Super
instance Instance SuperInstance
instance Super SuperInstance

class Instance s => Sub s
data SubInstance
instance Object SubInstance
instance Instance SubInstance
instance Super SubInstance
instance Sub SubInstance

-- the inheritance tree for class objects
class Object c => Class c where
    -- Given the name of a class, returns its class object
    getClassObjectFromName :: String -> c
    -- Given a class object, returns its name
    classObjectName :: c -> String

data ClassObject -- can represent any class object
instance Object ClassObject
instance Class ClassObject

class Class s => SuperClass s
data SuperClassObject
instance Object SuperClassObject
instance Class SuperClassObject
instance SuperClass SuperClassObject

class Class s => SubClass s
data SubClassObject
instance Object SubClassObject
instance Class SubClassObject
instance SuperClass SubClassObject
instance SubClass SubClassObject
```

Listing 3.11: The getClassObject function

```
getClassObject :: Class c ⇒ c
getClassObject =
    let
        x = upcast (getClassObjectFromName (classObjectName x))
    in
        x

-- usage:
_Sub_ = getClassObject :: SubClassObject

_Super_ = getClassObject :: SuperClassObject
```

getMetaclassObject function if one is needed. The *_classname_* syntax used to retrieve a class object can be logically extended to *__classname__* for metaclass objects.

3.4 Representing Components and Objects in Haskell

Now that an approach to model and automatically generate an object-oriented class hierarchy in Haskell is available, an obvious problem is how to *represent* an object or component in Haskell. What Haskell data type should be given to foreign components?

The question of how to represent objects in Haskell can be made more specific: what is the most *convenient* way to represent the object in Haskell? One can have two non-exclusive points of view about the convenience of an object representation: is it convenient for an *author* who is writing a binding from Haskell to the component system to manipulate objects, and is it convenient for a *user* of such a binding to manipulate objects? These two goals should be kept in mind, so that both writing a binding and using the binding is as straightforward as possible.

One obvious solution to this problem is simply duplicating each of the desired component's properties in a Haskell data type. Every object has methods and perhaps some public instance variables. If the Haskell environment can directly access these properties, perhaps by writing to the object's address space (for example, by making the object an instance of the FFI Storable type class), then component communication simply consists of direct function calls to invoke methods, reading and writing an object's instance variables can be performed with peek and poke.

This approach of duplicating as much of the target component as possible inside the Haskell environment has two drawbacks:

1. The target component system is not likely to expose the object's internals so easily, since encapsulation is an extremely important goal of component systems. A CORBA or DCOM component may reside on an Internet e-

Business Enterprise Application Server running on a different operating system or architecture, which makes it impossible to directly replicate the component's address space layout in the Haskell environment.

2. Representing as much of a component as possible increases—not decreases—the cost of *marshalling* the component to and from the component system. Any direct representation of an object's data in Haskell requires code in the Haskell environment to perform operations on the data.

Given these drawbacks, a logical alternative solution is to use the most *abstract* representation of an object or component, and this solution will indeed suffice for the majority of component systems. For instance, the most abstract representation of an object in Objective-C is simply a C pointer, which is easy to represent and marshal in Haskell. Similarly, Java uses an *object reference* to represent an object, which is the Java Virtual Machine's equivalent to a C pointer.

By using a highly abstract representation of a foreign component, a Haskell binding can use of all the component system's native capabilities to operate on the component, instead of re-writing all those operations in the component binding. After all, the component system must know how to access its own components, no matter how abstract its representation is! This approach is also less dependent on the component's representation, which allows the target component system to change and evolve without affecting bindings to the system.

Pointers to components

Since many object-oriented languages use a pointer as their most abstract representation of an object, how can one represent a pointer in the Haskell environment? The FFI provides a Ptr type for exactly this purpose, so it seems that this problem is easily solved. While it is easy to use type synonyms such **type** Super = Ptr () and **type** Sub = Ptr () to use void pointers to represent objects, there are two significant problems with this approach:

1. Pointers are typed. Both C and Haskell's type systems distinguish between an **int** $*$ (Ptr Int) and a **void** $*$ (Ptr ()), for example. Using type synonyms does not enable the type checker to distinguish between different types of objects. The Super and Sub types given in above example would be treated as exactly the same type.

2. Since type synonyms are nothing more than aliases, each type synonym cannot be made instances of different type classes. This renders it impossible to use the type class & data type scheme described by Shields and Peyton Jones [4] to model the object-oriented class hierarchy.

These two problems indicate that either the Haskell data or newtype declarations must be used to declare Ptrs. Here, the less popular newtype declaration is the better option. To see why, the Haskell 98 report explains what newtype is intended for:

> A declaration of the form newtype introduces a new type whose representation is the same as an existing type. The [new] type renames the datatype ... It differs from a type synonym in that it

> creates a distinct type that must be explicitly coerced to or from the original type. Also, unlike type synonyms, newtype may be used to define recursive types ... These coercions may be implemented without execution time overhead; newtype does not change the underlying representation of an object.
>
> New instances ... can be defined for a type defined by newtype but may not be defined for a type synonym. — *Haskell 98 report [10], Section 4.2.3.*

An important property of newtypes not quoted from the report is that any newtypes created are automatically marshallable by the FFI, if the type that they rename is also marshallable by the FFI.

Another useful technique is to make the newtype declaration recursive, e.g. newtype Object = Object (Ptr Object). This allows any methods which *unpack* the data type to still observe that the pointer in the data type does is indeed typed, and points to an Object. Lazy evaluation ensures that the parameterised type argument in Ptr will never be evaluated. C→Haskell uses this technique when automatically generating pointer declarations with its *pointer hooks*.

Chapter 4

Component Communication

Given a design for modelling an object-oriented class hierarchy in Haskell, we now address the question of how to communicate with other components. While the way to model an object-oriented class hierarchy is powerful, is it possible to communicate with other objects/components without sacrificing any of the power of this model? It is possible to build well-typed messages which contain these objects, send those messages to and from other objects, and receive object types as a message reply? Is it possible to do this while still retaining features such as Haskell's type inference?

4.1 Sending Messages: the Low-Level Interface

The first problem to overcome is how to actually send messages to and receive messages from other components. This problem is tackled first, because it may have a significant impact in how the rest of the communications library is designed. For example, if the low-level messaging functions are allowed to send and receive messages which contain values of any type, then this may ease the design of the rest of the communications library, compared to the situation where the types are restricted.

4.1.1 The Foreign Function Interface

Other Haskell language bindings have all used the standardised Foreign Function Interface (FFI) to call functions written in other languages, and for good reason. The FFI is the *dé facto* way for Haskell environments to interact with external environments such as the operating system or other programming languages. The FFI is most often used to call C functions, since the majority of operating system APIs are written in C. However, it is not limited to C: the FFI is designed in such a way to perform function calls for any language or platform, and in its present form already incorporates support for calling functions written for the C, C++, .NET, Java Virtual Machine, and Win32 platforms.

4.1.2 The FFI: Extend or Embrace?

Since the FFI is extensible, there are two possibilities for how to use it to communicate with component systems:

1. extend the FFI to natively support the desired component system, enabling direct method invocation on components, or

2. implement *proxy* or *stub* functions in a language which the FFI already supports, and rely on the proxy functions to establish communication channels with the component system.

It is tempting to extend the FFI to natively support communicating with the desired component system, but there one major practical drawback to this approach: it's a lot of work. An FFI extension would have to be implemented for each different Haskell compiler or interpreter. To emphasise this point, many Haskell environments only support the C and Win32 calling conventions. Some component systems, such as CORBA, do not even define a standard calling convention, so extending the FFI to natively support the communications protocols of those component systems doesn't make any sense—there is no standard communications protocol to support! To extend the FFI to communicate with these sorts of systems, we would have to instead target one or more particular implementations of the component system's reflection service. For example, instead of bridging to CORBA, the FFI could be extended to enable communication via CORBA's IIOP messaging standard, or perhaps via a concrete Object Request Broker implementation such as ORBit.

So, while having a native method of communication with the component system would be quaint, we will instead choose to take advantage of the popular support for the C and Win32 FFI bindings. As long as the goal of communicating with the component system is achieved, this design decision will enable a component interface to work across multiple Haskell implementations, and is arguably a less complicated and less error-prone path to walk.

It can be said that this choice simply shifts the problem of marshalling messages from Haskell to whatever proxy language is chosen in for the component binding (such as C). However, the point of this problem-shift is that it is far more likely that bindings already exist for talking to the component system in the external language. There are numerous implementations of the IIOP messaging protocol in C, for example.

Note that by using the FFI to communicate with external components, the implementation of the communications library will be bound by the FFI's capabilities. For example, the FFI restricts the type definitions of the foreign functions: it allows only a small set of *basic foreign types* to be passed between external functions and Haskell. Moreover, if a specific FFI language binding such as C is chosen, the types which can be used will be further restricted by the C-specific part of the FFI.

In reality, this restriction is not too worrisome since it is possible to interchange arbitrary data types back and forth from Haskell, as long as those data types are instances of the Storable type class. (Keep in mind that the FFI permits the construction of Storable instances for any data type, and this is exactly what interface generation tools such as C→Haskell—or Template Haskell—can be used for.) Still, this means that we must be conscious about the FFI's restrictions during the design of the communications interface.

4.1.3 C as the Proxy Language

To keep examples simple, we will assume that C is being used as a proxy language for the rest of this thesis. The Haskell environment will call C functions to send messages, and it is the responsibility of the C functions to communicate with the component systems. How the C implementation works is irrelevant: it is sufficient as long as it can *forward* the messages from Haskell to the component system, and return any message replies from the component system back to Haskell.

C has been chosen not only because it is the most common FFI language binding available, but also because it is quite restrictive. Other proxy languages or platforms such as .NET or the Java Virtual Machine can of course be used, but they may have extra features (such as innate ad-hoc polymorphism capabilities) which C does not possess. It is ideal to design a communications interface which will be usable on as wide a range of FFI language bindings as possible.

4.1.4 Messages, Receivers & Message Expressions

In Java, a method invocation on a component is performed by writing aComponent.aMethod(argument1, argument2, ...). Using this example, let us define a few terms more specifically:

Message The name of the method to invoke, along with any of its arguments; e.g. aMethod(argument1, argument2, ...).

Receiver The object/component which receives the message, e.g. aComponent.

Message Expression A particular instance of a message together with a corresponding receiver, i.e. a concrete representation of a method invocation.

Thus, to send a *message*, one needs to pair it with a *receiver*, which produces a resulting *message expression*.

It is possible to represent these elements of a method invocation in Haskell by defining them as data types. For example, one possible way to define these types is shown in Listing 4.1.

Listing 4.1: Definitions of the Message, Receiver, and MessageExpression data types

```
data Message a = Message a
-- a is the message's arguments, given as tuples such as (foo, bar).
-- Note that we can't use a list to represent the arguments, since
-- each argument may be a different type.

type Receiver = UniversalObject

data MessageExpression a = MessageExpression (Message a) Receiver
```

However, there is a problem if method invocations are defined using such data types: eventually, we must call the C proxy message functions to send

messages, which means that the Message type must be a type which can be marshalled by the FFI. So, the Message type must either be basic foreign type or an instance of the Storable FFI type class, which implies that implementations of the functions required by Storable (such as peek and poke) must be written.

In order to make marshalling easier, a Haskell data type which is already an instance of Storable can be used: the humble pointer (Ptr) type. Instead of defining the data types for Message, Receiver and MessageExpression similarly to Listing 4.1 (page 30), it is sufficient to define them as Ptrs and simply use extra C functions to set the primitive information about a message expression's contents, as shown by Listing 4.2.

Listing 4.2: Primitive MessageExpression functions

```
newtype MessageExpression =
    MessageExpression (Ptr MessageExpression)

type Receiver = UniversalObject

foreign import ccall
    makeMessageExpression :: IO MessageExpression

foreign import ccall
    setReceiver :: MessageExpression → Receiver → IO ()

foreign import ccall
    setMethodName :: MessageExpression → CString → IO ()
    -- assuming that the method name to invoke is a C string
foreign import ccall
    setCIntArgument :: MessageExpression → Int → CInt → IO ()
    -- the 'Int' argument specifies which index in the argument list
    -- to set
foreign import ccall
    setCStringArgument :: MessageExpression → Int → CString → IO ()
    -- (repeat for the rest of the C basic foreign types) ...
```

There are a few noteworthy points about representing message expressions in this manner:

- Defining a data type using the newtype keyword instead of data will automatically make the newly defined type marshallable, as long as the type referred to in the newtype declaration (in this case, a Ptr) is also marshallable.
- Only a MessageExpression data type needs to be defined.
- There is no need to marshal an argument list (at least, not in the low-level communications layer), which may be of varying length and can contain elements of different types. Instead, extra C functions are available to individually set each argument in a message expression.

A disadvantage of this design is that there must be one function for every possible type of an argument, but this not a problem since there are only a limited number of basic foreign C types. This design is also completely type-safe, in both the Haskell and C environments.

- The representation of the message expression has been moved from the Haskell environment to the C environment. Exactly how the message expression is represented in the C environment is completely inconsequential to the Haskell/C interface.

  This has another extra benefit: if the component system has already defined a data type for a method invocation or message expression, the C proxy functions can simply use that data type to represent the message expression, instead of having to define one. This may make communication with the component system a bit more speedy, since the C functions (and thus the Haskell code) may be using the component system's native representation of a method invocation. Of course, one could argue that this speed benefit would also apply if the native data type was represented in Haskell, but again, the component system is likely to have C bindings already available, which saves writing more code.

After defining the C proxy functions, it is now possible to build message expressions in Haskell. Still, message expressions are relatively useless if they are never used, so let us now address how to send the message expressions to the component system.

4.1.5 The mailman arrives

The one thing which is required to send a message expression is simply a definition of the C function to send it, as shown in Listing 4.3.

Listing 4.3: sendMessageExpressionWithNoReply interface

```
foreign import ccall
    sendMessageExpressionWithNoReply :: MessageExpression → IO ()
```

This leaves the responsibility of the actual message sending completely in the hands of the C function. How the message sending is performed varies between component systems, and is thus not a concern for the Haskell/C interface.

Of course, this assumes that the message expression does not receive a reply of some sort. Since it is rather desirable to receive replies from the component system, Listing 4.4 defines a few more C functions which can extract the reply from a sent message expression, and pass it back to Haskell.

Again, the C functions are responsible for extracting the reply and passing them back to Haskell, and how this is performed depends on the component system.

Note that using these low-level functions to receive replies from messages *may break type safety*: if sendMessageExpressionWithIntReply is called and the C function passes back a CString as a result, then behaviour from that point is

Listing 4.4: Other sendMessageExpressionWith*x*reply functions

```
foreign import ccall
    sendMessageExpressionWithIntReply :: MessageExpression → IO Int
foreign import ccall
    sendMessageExpressionWithIntReplyP :: MessageExpression → Int
    -- 'Pure' version of the above (has no side effects); useful for,
    -- e.g. using a math library component. Define a pure version
    -- for each type of reply if appropriate.

foreign import ccall
    sendMessageExpressionWithCStringReply ::
        MessageExpression → IO CString

foreign import ccall
    sendMessageExpressionWithUniversalObjectReply ::
        MessageExpression → IO UniversalObject

-- etc ...
```

undefined. If the component system's message passing protocol has the functionality to obtain the type of the message reply, it is certainly possible to add in type checks which will be performed at run-time and return a exception if such a type mismatch occurs. This is a prudent idea, and it is up to the C proxy functions to perform these dynamic type checks and propagate any type errors back to the Haskell environment. Type checking can also be enforced if it is known at compile-time that sending a particular message will always evoke a reply of a certain type. However, it is not the responsibility of the low-level communications interface to perform this static type checking: since a component system may be inherently dynamically typed, the message sending functions must also be dynamically typed.

Now that it is possible to build message expressions and send them to a receiving component, we have a complete low-level interface in Haskell to communicate with a component system. It is important to note that the concept of a message expression and the corresponding C functions to act on it are generalised enough to work with *any* component or object-oriented system, and that an existing C binding for the component system can be used, if one is available, to implement most, if not all, of these functions—which saves much time and effort.

4.2 Creating a friendlier mailperson

While the low-level communication interface is necessary, it is evident that using these functions is not particularly convenient. Not only is it necessary to set the arguments of a message expression individually, but different functions must be invoked depending on what types the arguments are. Thus, the next priority is to *wrap* the low-level functions with a higher-level interface, so a programmer

can write code that builds & sends message expressions in one line, instead of forty-two.

A convenient interface would be to have a generic sendMessage function, which can be used to send a message that contains an arbitrary number of arguments to any receiver, as shown in Listing 4.5.

Listing 4.5: An ideal interface for sendMessage

```
sendMessage aReceiver "aMethodName" arg1 arg2 arg3 -- :: IO ()
sendMessage anotherReceiver "anotherMethod" arg1-- IO ()
-- sendMessage may also return a value:
aNumber ← sendMessage yetAnotherReceiver
    "yetAnotherMethod" arg1 arg2 -- IO Int
```

4.2.1 Setting the Message Arguments

Perhaps the most cumbersome aspect of the low-level interface are the functions which set the arguments of a message: setIntArgument, setCStringArgument, etc. These functions are used in the same manner, except that the programmer must call a different version of the function depending on the type of the argument that they wish to set. This is a perfect example of ad-hoc polymorphism, and Haskell offers type classes to solve this exact problem, as shown in Listing 4.6.

Listing 4.6: Using ad-hoc polymorphism to write setArgument

```
class Storable a ⇒ Argument a where
    setArgument :: MessageExpression → Int → a → IO ()

instance Argument CInt where setArgument = setCIntArgument
instance Argument CString where setArgument = setCStringArgument
instance Argument UniversalObject where
    setArgument = setUniversalObjectArgument
-- etc ...
```

Now, to send a message with the arguments (6, 'n'), instead of writing setCIntArgument 1 6; setCharArgument 2 'n', we instead write setArgument 1 6; setArgument 2 'n'. The burden of selecting the appropriate function is shifted from the programmer to the type system.

This approach of overloading the functions which marshals individual arguments is very similar in concept to the approach taken by Lambada [2], although the equivalent type class used in Lambada (named GoesToJava) can marshal more than one argument. Our Argument class works with just one argument.

4.2.2 Sending Variable Arguments in a Message

In the ideal version of sendMessage in Listing 4.5, it is possible to set the message arguments by simply passing them to sendMessage as function parameters.

Unfortunately, it is impossible to define such a version of sendMessage, since the number of message arguments can vary from message to message, and Haskell 98 does not permit definitions of functions with a variable number of function parameters. This is for good reason: if a variable number of function parameters were permitted, two powerful Haskell features—partial application and currying—would be far harder (if not impossible) to implement. How, therefore, do we implement a generic sendMessage function which has a *fixed* number of parameters, yet still allows for sendMessage to work with variable numbers of arguments per message?

One solution is to allow for one of the parameters in sendMessage to contain an arbitrary number of expressions. We require this *message arguments parameter* to have two properties:

1. It must be able to store an arbitrary (but finite) number of message arguments.

2. Each message argument may be of a different type, so the parameter which stores the message arguments must be able to accommodate different types of elements.

Lists with existential types

It is possible to use a normal Haskell list to represent the message arguments: lists are designed to contain an arbitrary number of elements, after all! The problem is that lists can only hold elements of the same type: e.g. the list [6, 'n'] is ill-typed.

However, a Haskell 98 type extension known as *existentially quantified data constructors with type classes* enables the construction of an *existential type*. An existential type can hold any arbitrary data type, which may optionally be contained by a type class. Listing 4.7 shows how they can be used to implement a list which stores different types of message arguments.

Listing 4.7: Using existential types for an argument list

```
data OneArgument = forall a. Argument a ⇒ AnArgument a

-- how to use the data type as an argument list
arguments :: [OneArgument]
arguments = [AnArgument 6, AnArgument 'n']
```

While this technique enables the use of lists to encapsulate message arguments, it has two disadvantages:

1. It is not standard Haskell 98. While many Haskell environments support existential data types (including the Glasgow Haskell Compiler, HUGS, and the Chalmers Haskell Compiler/hbc), it is preferable to use standard Haskell 98 constructs where possible.

2. It requires prefixing the data type constructor to every argument in the list, which is hardly convenient. As demonstrated by Listing 4.8, even

if the data constructor in Listing 4.7 was shortened from AnArgument to simply A, using the sendMessage function would continue to be neither convenient nor elegant.

Listing 4.8: Using sendMessage with a list of existential types

```
sendMessage aReceiver "aMethod" [A 6, A 'n']
```

The inconvenience of the need to tag each argument with a data constructor is indeed a shame; otherwise, existential types would be a perfect way of marshalling arguments.

A sendMessage template

Another solution to enable calling sendMessage with multiple messages arguments simply bypasses the need to have a single parameter which contains the message arguments. With Template Haskell, one can create a template for the sendMessage function call, which, at compile-time, is evaluated to produce functions that have the correct number of message arguments for each particular message. As an example of how this works, the original Template Haskell paper [8] implements the C printf function—which uses a variable number of arguments—in Haskell; the usage of printf and an analogous sendMessage function is shown in Listing 4.9.

Listing 4.9: Using Template Haskell versions of printf and sendMessage

```
-- the $(...) syntax 'splices' in a definition of a template,
-- which produces a specific instance of the function as appropriate

main = do
    $( printf "A string %s, a character %c and a number %d\n")
        "Foo" 'c' 69
    $( printf "Only the number %d\n") 69
    -- a possible analogous version of sendMessage
    $(sendMessage aReceiver aMethod) "Foo" 'c' 69 -- 3 arguments
    $(sendMessage anotherReceiver anotherMethod) 69 -- 1 argument
```

The trick to implementing printf is that the template function for printf is called with a format string, which explicitly tells printf how many arguments follow the splice: in the first use of printf in Listing 4.9, the format string contains three *conversion specifiers*: %s, %c and %d, so printf knows that there are three arguments following the splice.

This has implications for a how to implement a template version of sendMessage: the sendMessage template must know exactly how many arguments the message requires. There are two approaches that can be taken to give sendMessage this knowledge:

1. sendMessage determines the number of messages directly from the method name or the combination of the receiving object and method name: this

is possible in some component systems. Objective-C, for example, has method names such as initWithScheme:host:path:; the number of arguments that each method requires is determined exactly by the number of colons (:) in the method name. Still, this may not be true of every component system, so this approach is not universal.

Note that building a lookup table which yields the number of arguments required, given a method name, does not solve the problem, since it would not be possible to send message expressions for method names which are unknown at compile-time; this is a legitimate and often mandatory requirement if the component system is dynamic.

2. Each sendMessage function call would have to be explicitly annotated with the number of arguments that the method requires; e.g. $(sendMessage aReceiver aMethod 3)"Foo" 'c' 69. While this solution works, it is clearly an inconvenience for the Haskell programmer, and is also inelegant—it is preferable for sendMessage to automatically infer the number of arguments needed for the message.

So, while it is possible to use Template Haskell to enabling building a sendMessage command that accepts a truly variable numbers of arguments, it will either not work for all component systems, or the programmer must explicitly specify how many arguments are used. Another solution will have to be found.

Using tuples for message arguments

The humble tuple is normally used to group together related items, such as the common (Key, Value) tuple used for lookup tables. Recall that the 'message arguments' parameter passed to sendMessage must have two properties: it must support encapsulating an arbitrary number of elements, and it must support message arguments of different types. Tuples have both these properties: they can be of any length (although usually the Haskell compiler imposes a limit on the maximum number of elements, such as 64), and they may contain elements of different types.

The problem is writing a *function* which allows an arbitrary-length tuple: what would be its type signature? A type signature of :: (a, b, c) is perfectly acceptable; this allows a three-element tuple where each element may be of a different type. However, it is impossible to give a type signature to a function which requires a tuple parameter of different lengths.

The trick is to not use tuples in the type signature at all. Tuples are still used to 'wrap' all the message arguments in a single parameter; they just don't appear in the type signature. Instead, a type variable is used to represent the parameter where the tuple would normally be; the type variable can then accept *any* type of parameter, including tuples of varying length. Listing 4.10 shows how such a type variable may be used.

Listing 4.10: sendMessage type signature with an arbitrary parameter

```
sendMessage :: Receiver → MethodName → a → -- etc ...
```

There is one show-stopper problem with this, though: since the 'message arguments' parameter given to sendMessage can be of any type, one cannot do anything useful with that type—other than pass the parameter on to another function—since it is impossible to know what operations can be performed on it. In particular, it is not possible to pattern match against the tuple to extract the elements from it, since it is not even known whether the argument passed to the function is, in fact, a tuple at all!

The goal, then, is to allow sendMessage to accept a parameter with any particular type, except that the parameter passed to it needs to have certain properties: in this case, it should have the property of being a tuple, so that its elements can be extracted, and setArgument can be called on each of them. This requirement of a type possessing certain properties is exactly what type variable constraints and type classes are designed for; we therefore define an Arguments type class[1] in Listing 4.11 which contains functions that allows for the extraction of the tuple's elements.

Listing 4.11: The Arguments type class

```
class Arguments args where
    arguments :: args → Int
    elem1 :: Argument a ⇒ args → a -- extract first element
    elem1 t = error "Attempted to extract 1st element from length "
        ++ show (arguments args) ++ " tuple" -- default method
    elem2 :: Argument a ⇒ args → a -- extract second element
    elem1 t = error "Attempted to extract 2nd element from length "
        ++ show (arguments args) ++ " tuple"
    ...
    elem63 :: Argument a ⇒ args → a -- as many as is needed

instance Arguments () where
    arguments _ = 0
instance Arguments (a) where
    arguments _ = 1
    elem1 a = a -- or 'elem1 = id'
instance Arguments (a, b) where
    arguments _ = 2
    elem1 (a, b) = a
    elem2 (a, b) = b
-- repeat for ≥ 2 arguments
```

This method of using a type class to represent and extract information from a tuple was inspired by, and is nearly the approach which Lambada [2] uses in its GoesToJava type class, to marshal multiple arguments for a Java method invocation. However, Lambada's GoesToJava type class abstracts away the need to extract each message argument from the tuple: instead, it defines a function marshal which extracts the elements from the tuple and calls the equivalent of setArgument itself.

[1] Note the s on the end of Arguments; this is a different type class to Argument which was defined in Listing 4.6 (page 34).

This approach used by Lambada is superior, not only because it allows the type class to be written with less code, but it also abstracts away the data type entirely: as long as the data type implements the marshal function, it does not matter what data type is used to store the message arguments. Thus, the Arguments type class can be re-implemented more succinctly as shown in Listing 4.12; our equivalent to Lambada's marshal function is named setArguments.

Listing 4.12: The Arguments type class, redux

```
class Arguments args where
    setArguments :: MessageExpression → args → IO ()
    setArguments = setArgument 1

-- 'tuples' with only one element (not really a tuple)
instance Arguments Char where setArguments = setArgument 1
instance Arguments Int where setArguments = setArgument 1
instance Arguments CString where setArguments = setArgument 1
-- etc ...

-- tuples ≥ 1 argument
instance (Argument a, Argument b) ⇒ Arguments (a, b)
    where
        setArguments expr (a, b) = do
            setArgument 1 expr a
            setArgument 2 expr b
instance (Argument a, Argument b, Argument c) ⇒ Arguments (a, b, c)
    where
        setArguments expr (a, b) = do
            setArgument 1 expr a
            setArgument 2 expr b
            setArgument 3 expr c
-- etc ...

-- we can even use existential types to store message arguments,
-- if we want to prove that the Arguments type class can abstract
-- over the data type:
data ExistentialArgument = forall a. Argument a ⇒ A a
instance Arguments [ExistentialArgument] where
    setArguments args =
        mapM_ (λ(i, a) → setArgument i expr a) argsWithIndices
            where
                argsWithIndices = zip [1..] args
```

Now that an Arguments type class has been defined, it is possible to pass arbitrary message arguments to the sendMessage function; after creating a new message expression using the C proxy function makeMessageExpression defined in Listing 4.2 (page 31), sendMessage merely has to call the setArguments function to set the arguments in the built message expression.

4.2.3 Overloading the message reply type

Now that sendMessage is capable of being called directly with a variable number of message arguments, the last step in implementing the ideal version of sendMessage is to make it automatically determine the type of the message reply, and invoke the appropriate sendMessageExpressionWith*x*Reply function from Listing 4.4 (page 33).

Once again, Lambada [2] has a solution to this ad-hoc polymorphism in the form of its ComesFromJava type class; our version is conceptually exactly the same. We declare a new type class named MessageReply, and require any instances of the type class to implement a type-specific version of a generalised sendMessageExpression function, as shown in Listing 4.13.

Listing 4.13: The sendMessageExpression implementation using a MessageReply type class

```
class MessageReply r where
    sendMessageExpression :: MessageExpression → IO r

instance MessageReply () where
    sendMessageExpression = sendMessageExpressionWithNoReply
instance MessageReply Int where
    sendMessageExpression = sendMessageExpressionWithIntReply
instance MessageReply CString where
    sendMessageExpression = sendMessageExpressionWithCStringReply
-- etc ...
```

4.2.4 Implementing and using sendMessage

Finally, it is possible to implement sendMessage by building on all the techniques introduced in this section: Listing 4.14 contains the full implementation of sendMessage. Thanks to liberal use of type classes to perform ad-hoc polymorphism, its implementation is (perhaps surprisingly) short and straightforward—especially considering how many lines of code that the function saves from writing, compared to using the low-level interface to send messages.

There are two caveats to the final implementation and usage of sendMessage as shown in Listings 4.14 and 4.15, compared to our 'ideal' version in Listing 4.5 (page 34):

1. The ideal version of sendMessage allows for a truly variable number of parameters: one function parameter per message argument. Our concrete implementation of sendMessage requires bundling up all the message arguments into one parameter.

2. Since sendMessages final type is a (constrained) type variable, we must *fix* the type variable by giving sendMessage an explicit type signature whenever it is used; see Listing 4.15 for an example.

Listing 4.14: The sendMessage implementation

```
sendMessage :: (Arguments args, MessageReply r) ⇒
    Receiver → MethodName → args → IO r
sendMessage receiver methodName arguments = do
    e ← makeMessageExpression
    setReceiver e receiver
    setMethodName e methodName
    setArguments e arguments
    r ← sendMessageExpression e
    return r

-- pure version of sendMessage (useful for e.g. numeric libraries
-- which are available as objects/components)
sendMessageP :: (Arguments args, MessageReply r) ⇒
    Receiver → MethodName → args → r
sendMessageP receiver methodName arguments =
    unsafePerformIO (sendMessage receiver methodName arguments)
```

Listing 4.15: Using sendMessage

```
-- assume that we have the following components declared
numericLibraryComponent :: UniversalObject
networkComponent :: UniversalObject

main = do
    let eighteen = sendMessageP numericLibraryComponent
        "multiply" (3, 6) :: Int
     fileDescriptor ← sendMessage networkComponent
        "openSocket" ("localhost", 6667) :: IO CInt
    sendMessage networkComponent
        "writeString" ( fileDescriptor ,
            "USER foo localhost localhost : Haskell") :: IO ()
```

Even with these caveats, sendMessage is reasonably convenient to use. It allows sending any arbitrary message to any component, and can receive a reply of any type: it is thus completely dynamic in nature.

4.3 Direct Messaging: Statically type-checked message sending

Recall the definition of sendMessage in Listing 4.14 (page 41). While this definition of sendMessage is dynamic enough to allow for any type of message to be sent to any receiving object and receive any type of message reply, this dynamism is also a weakness. In a sense, it is *too* dynamic for most purposes: the type system can only assume that the final type for the sendMessage function will be *some* type r, where r could be *any* type which belongs to the type class MessageReply. This has the unpleasant side-effect that whenever the sendMessage function is called, the type system must be told explicitly what the reply type will be (i.e. the resulting type variable must be *fixed*), otherwise the type of sendMessage will be indeterminate.

It is also necessary to explicitly upcast the receiving object to the UniversalObject type, so that the sendMessage expression will be well-typed. This inconvenience is due to a requirement for a messaging function which can communicate with any object: message sending can only work if the receiving object is the UniversalObject type, so an explicit upcast is mandatory *somewhere* before sendMessage calls the C proxy function to perform the message sending.

A similar problem is that *any objects in the argument list* must also be upcast to the UniversalObject type when using sendMessage. Moreover, if the message *reply* is then an object and its type is known at compile-time to be a subtype of UniversalObject, an explicit downcast must be performed on the message reply to its known subtype.

As a example, let us examine a typical hashtable container object, which stores an object (a *value*) of type SomeValue, which can be retrieved from the hashtable by using another object as a lookup parameter (the *key*). Compare and contrast the Java (Listing 4.16) and Haskell (Listing 4.17) versions of putting an object anto the hashtable and retrieving it.

Listing 4.16: Hashtable object manipulation in Java

```
aHashtable.put (aKey, aValue);
SomeValue theRetrievedValue = (SomeValue) aHashtable.get (aKey);
```

It is clear that the Haskell version does not meet the two goals of allowing elegant and convenient communication with components. Ideally, the Haskell code to perform message sending should be as succinct as possible, so that instead of the tedious usage of sendMessage demonstrated by Listing 4.17 (page 43), its usage would look more like the code in Listing 4.18, which is how one would expect to use those functions if there was a Haskell Hashtable module which made them available.

The goal is therefore to have functions which automatically perform any typecasting and type fixing necessary to send a message, so that the program-

Listing 4.17: Hashtable object manipulation in Haskell using sendMessage—the code looks far more verbose than Java!

```
sendMessage (upcast aHashtable :: UniversalObject) "put"
    (upcast aKey :: UniversalObject,
     upcast aValue :: UniversalObject) :: IO ()
valueUniversalObject ← sendMessage
    (upcast aHashtable :: UniversalObject)
    "get" (upcast aKey :: UniversalObject) :: IO UniversalObject
let theRetrievedValue = downcast valueUniversalObject :: IO
    SomeValue
```

Listing 4.18: Convenient hashtable component communication in Haskell

```
put aHashtable (aKey, aValue)
theRetrievedValue ← get aHashtable aKey
```

mer does not have to explicitly perform those tasks. To implement these functions, the dynamic sendMessage function can have its dynamism constrained by wrapper functions such as put and get, when there is knowledge of all the message expression's types at compile-time. Let us call these static wrapper functions *direct messaging functions*.

4.4 Implementing Direct Messaging

The interface and implementation of the direct messaging functions has four requirements:

1. Their usage must be simple and concise, so that they can be used in a manner similar to Listing 4.18.
2. Both the type checker and type inference engine must be able to act on all expressions involving direct messaging; i.e. allowing ill-typed expressions or introducing inconsistencies in the type system is not permissible.
3. Whatever scheme is used to implement the direct messaging functions must be reasonably lightweight, so that it is feasible to construct and use thousands of these functions without substantial overhead.
4. It must be possible to automatically construct every direct messaging function using an interface generator such as C→Haskell or Template Haskell; it would be senseless to automatically map an entire object-oriented class hierarchy onto Haskell, only to require manual intervention from the programmer when constructing direct messaging functions.

Listing 4.19 shows a first attempt at writing the put and get functions. There are a number of similarities which can be observed in the structure of the functions:

Listing 4.19: Implementing the put and get functions

```
put :: Instance object
    ⇒ HashtableObject -- hashtable
    → (object , object) -- key, value
    → IO ()
put hashtable (key , value) =
    sendMessage (upcast hashtable :: UniversalObject) "put"
        (upcast key :: UniversalObject ,
         upcast value :: UniversalObject)

get :: Instance object
    ⇒ HashtableObject -- hashtable
    → object -- key
    → IO UniversalObject -- returned value
get hashtable key =
    sendMessage (upcast hashtable :: UniversalObject) "get"
        (upcast key :: UniversalObject)
```

1. They are both simple wrapper functions around sendMessage; their only task is to call sendMessage.
2. Both functions upcast the receiving object and all objects in the argument list to the UniversalObject type.
3. Both functions, like sendMessage, accept three parameters.
4. Even though sendMessage has a type variable for its result type, the sendMessage expression here does not need to be given an explicit type signature when it is called. The type signatures given for the put and get functions are used by the type system to *fix* sendMessage's free type variable.

Automatically upcasting objects in the argument list

Enabling an object to be sent in the argument list—without explicit upcasting—can be performed by simply making the relevant object data type (HashtableObject, in this example) instances of the Argument type class described in Listing 4.6 (page 34).

A lightweight way of doing this for many objects is by declaring a new type class named ObjectArgument, which all objects in the class hierarchy are instances of. ObjectArgument implements a *default function* for the type class named setObjectArgument, which can then be called by the objects when they are required to implement the setArgument function. Listing 4.20 shows the relevant code to do this.

Generalising put and get

There is one major problem with put and get: they only work with a receiving object type of HashtableObject. What if another object-oriented class, e.g.

Listing 4.20: The ObjectArgument type class

```
class Object arg ⇒ ObjectArgument arg where
    setObjectArgument :: Int → MessageExpression → arg → IO ()
    setObjectArgument index expr arg = setUniversalObjectArgument
        index expr (upcast arg :: UniversalObject)

instance ObjectArgument HashtableObject
instance Argument HashtableObject where
    setArgument = setObjectArgument

instance ObjectArgument FiniteMapObject
instance Argument FiniteMapObject where
    setArgument = setObjectArgument
```

FiniteMap, also provided put and get functions? In an object-oriented language, each class has a separate name space for method names. By writing o.m();, the method name m is chosen from within o's method namespace. Haskell does not have namespace separation between objects, so such a method invocation would be written as m o instead.

Moreover, it is possible to have two classes Foo and Bar, which both contain a method named overloadedMethod—and not only can overloadedMethod be overloaded within the class so that it can be called with different types of arguments, but the two classes may contain completely different return types for overloadedMethod! Listing 4.21 shows an example of this. If overloadedMethod is made available in Haskell to act as a convenient wrapper for sendMessage, what would its type signature be?

Listing 4.21: overloadedMethod in Java

```
class Foo
{
    // overloaded methods in Java must still have the same
    // return type
    int overloadedMethod (Object o, char c) { ... }
    int overloadedMethod (float f) { ... }
}

class Bar
{
    void overloadedMethod (int i) { ... }
    void overloadedMethod () { ... }
}
```

Resolving overloading

In fact, there is already a solution to this problem. sendMessage's purpose is to send a message to any receiving object, with an arbitrary argument list, and receive with any message reply type. So, writing an overloadedMethod function is a simple exercise of wrapping sendMessage, as shown in Listing 4.22.

Listing 4.22: Implementing overloadedMethod in Haskell

```
overloadedMethod ::
    (Object receiver , Arguments args, MessageReply reply)
    ⇒ receiver → args → reply
overloadedMethod receiver args reply = sendMessage
     receiver "overloadedMethod" args

-- usage (assume Foo and Bar are instances of the
-- appropriate type classes)
data Foo
data Bar

usage = do
    -- note that we must fix the return type of overloadedMethod,
    -- since its result type is a type variable
    i ← overloadedMethod Foo (anObject, 'c') :: IO Int
    i' ← overloadedMethod Foo (2.69 :: Float) :: IO Int
    overloadedMethod Bar (4 :: Int) :: IO ()
    overloadedMethod Bar () :: IO ()
```

Now that implementing a function such as overloadedMethod is possible in Haskell, let us once again address the two functions put and get. It is possible to implement put and get similarly to overloadedMethod, but if that is done, put and get will have the same freedom as overloadedMethod. What is required is a way to *constrain* the type variables which put and get can operate with. The type checker should produce an error if put and get are used with an combination of types which have not been specifically permitted.

put and get's types can be constrained by using a *multi-parameter type class*, where one parameter is used for each type variable that needs to be constrained. put, get, and any other method invocation have three parameters which need to be constrained: the receiving object, the types of the argument list, and the reply type.

Additionally, the type system must be told that the reply type is uniquely determined from the receiving object, otherwise it will not be able to properly *fix* the result type of the put and get functions. This can be done by using a Haskell 98 type extension known as *functional dependencies*. Listing 4.23 shows an implementation of put and get, using multi-parameter type classes with functional dependencies to constrain the types that can be used with the two functions.

Listing 4.23: put and get, allowed only to operate on Hashtable and FiniteMap receiving objects

```
class (Object receiver , Arguments args, MessageReply reply) ⇒
    Put receiver args reply
        -- the reply type is uniquely determined by the receiving
        -- object
        | receiver → reply

put :: ( Put receiver args reply ) ⇒ receiver → args → reply
put = sendMessage (upcast receiver :: UniversalObject ) "put" args

class (Object receiver , Arguments args, MessageReply reply) ⇒
    Get receiver args reply | receiver → reply

get :: ( Get receiver args reply ) ⇒ receiver → args → reply
get = sendMessage (upcast receiver :: UniversalObject ) "get" args

-- specifying which types can be used with put and get
instance Put HashtableObject (UniversalObject , UniversalObject ) ()
instance Get HashtableObject (UniversalObject ) UniversalObject

instance Put FiniteMapObject (UniversalObject , UniversalObject ) ()
instance Get FiniteMapObject (UniversalObject ) UniversalObject

-- the type checker will disallow the following instance declaration ,
-- since otherwise the reply type cannot be uniquely determined
-- from the receiving object . this is similar to declaring two
-- methods with the signatures "void foo() ;" and "int foo() ;" in
-- the same class in Java.
--
-- instance Put HashtableObject (UniversalObject , UniversalObject ) Int
```

The DirectMessage type class

Listing 4.23 shows many similarities between the Put and Get type classes. The only difference between them is that the Put type class is used to constrain the type variables of the put function, and the Get type class constrains the type variables of the get function.

It is possible to eliminate this requirement of one type class per function by introducing a DirectMessage type class which also includes the *method name* as an additional *phantom parameter*, and declaring a unique *method data type* for each method name. Each instance of DirectMessage uses the method data type in its instance declaration, and each direct messaging function *fixes* the phantom parameter using the method data type. Listing 4.24 shows how put and get can be implemented using the DirectMessage type class.

Listing 4.24: put and get implemented with the DirectMessage type class

```
class (Object receiver , Arguments args, MessageReply reply) ⇒
    DirectMessage receiver methodName args reply
        -- the reply type is uniquely determined by the receiving
        -- object _and_ the method name.
        | receiver methodName → reply

-- any definition for the method data type is okay; it only
-- matters that the method data type is unique per method name
data MethodName_put
put :: ( DirectMessage receiver MethodName_put args reply) ⇒
    receiver → args → reply
put = sendMessage (upcast receiver :: UniversalObject ) "put" args

data MethodName_get
get :: ( DirectMessage receiver MethodName_get args reply) ⇒
    receiver → args → reply
get = sendMessage (upcast receiver :: UniversalObject ) "get" args

-- specifying which types can be used with put and get
instance DirectMessage HashtableObject MethodName_put
    (UniversalObject , UniversalObject ) ()
instance DirectMessage HashtableObject MethodName_get
    (UniversalObject ) UniversalObject

instance DirectMessage FiniteMapObject MethodName_put
    (UniversalObject , UniversalObject ) ()
instance DirectMessage FiniteMapObject MethodName_get
    (UniversalObject ) UniversalObject
```

This design of using a DirectMessage type class is lightweight enough to be used with thousands of direct messaging functions. Another desirable property of this design is that an automated interface generator will find it trivial to generate the permitted declarations.

One interesting point about the DirectMessage class is that is in fact allows for *greater* flexibility in overloading than what would be possible in most object-oriented languages such as Java and C++. If the functional dependency for DirectMessage is modified to receiver methodName args → reply, this enables *multiple method definitions in the same class which can have different result types depending on the arguments.* This is not possible in Java or C++; writing **interface** Foo { **void** aMethod(); **int** aMethod(**int** i); } in Java would result in a compile-time error.

4.5 Transparent marshalling

The Argument and MessageReply type classes enable more than just determining which data types can be sent and retrieved from the component system. They can perform *transparent marshalling.* For example, Listing 4.25 shows how the Haskell String type can be transparently marshalled to and from a StringObject before a message is sent or a message reply is retrieved from the component system.

Listing 4.25: Transparent String ↔ StringObject marshalling

```
instance Argument String where
    setArgument expr index arg = do
        -- create the string object (by using the direct
        -- messaging function named "new")
        stringObject ← new _StringObject_ arg
        setArgument expr index stringObject
        where
            _StringObject_ = getClassObject
                :: StringObjectClassObject

instance MessageReply String where
    sendMessageExpression receiver methodName args = do
        o ← sendMessageExpressionWithUniversalObjectReply
            receiver methodName args
        let stringObject = downcast o :: StringObject
        -- get the string inside the string object as a C string
        cString ← getCString stringObject
        -- return the CString
        peekCString cString
```

4.6 Monadic binding and object-oriented syntax

For clarity, the sendMessage function has demonstrated so far with the receiving object as the first parameter in the argument list. However, it is more advantageous to placing the receiving object *last* in the argument list, because then it is possible to create a new # function which allows writing object # method rather than method object. This more closely resembles object-oriented notation.

It is also possible to use the monadic bind operator (>>=) to permit building *message chains*, where the result of one message expression is used as the *last parameter* to a following message expression. Listing 4.26 shows the implementation of the # function, and Listing 4.27 demonstrates how using # and >>= can lead to compact, object-oriented like code.

Listing 4.26: Implementation of the # function

```
(#) :: object → (object → reply) → reply
object # method = method object
```

4.7 Receiving messages

For Haskell to act as a first-class citizen in a component system, it must be able to function as a component, and receive and reply to messages as well as send them.

There are two approaches which enable Haskell expressions to be represented as a component. These parallel the two approaches described in Section 3.4 (page 25):

1. The component is represented in Haskell as a data type which is completely compatible with and indistinguishable from any other component: a Haskell expression *is* a component.

2. A *surrogate* component is written in the proxy language, which is capable of forwarding messages and queries received from component system to the Haskell environment.

It is recommended, wherever possible, to use a surrogate component instead of representing the component natively in Haskell, and to perform the actions required to create a component in the proxy language rather than Haskell. The reasons for this recommendation are the same as the reasons for representing foreign components in Haskell: greater encapsulation and decreased marshalling code.

The interface required of a component and the process of creating and registering a component with the component system are completely dependent on the component system being used. Even so, it is possible to propose some universal guidelines which are independent of the component system:

1. The code to handle incoming messages should be written in the surrogate component where possible. This decreases the amount of marshalling code which has to be written.

2. There should be a restricted number of entry points into the Haskell environment. Again, this decreases the amount of marshalling code which has to be written.

3. It may be possible for the surrogate component to marshal all incoming messages as objects. If this is done, the communications functions defined previously in this chapter can be used to retrieve information from

Listing 4.27: Using # and >>= for succinct code

```
-- exactly the same implementation of sendMessage shown
-- previously, except that the receiving object is placed
-- as the last parameter instead of the first parameter
sendMessage :: (Arguments args, MessageReply r) =>
    MethodName -> args -> Receiver -> IO r

-- assume that the following methods exist, implemented
-- via direct messaging functions which wrap around
-- sendMessage. like sendMessage, the receiver in all these
-- functions is also the last parameter.

-- creates a new instance object from a given class object;
-- may return a different type of instance object depending
-- on the type of the class object
new :: (Arguments a, ClassObject c, Instance i) => a -> c -> i
-- retrieves the data at a particular URL
fetch :: Arguments a => a -> URLObject -> DataObject
-- returns the information in the data object as a StringObject,
-- which is automatically marshalled to a String
getContents :: Arguments a => a -> DataObject -> String

-- implementation of a www client in Haskell, using
-- a component system's URL library
main :: IO ()
main = do
    args <- System.getArgs
    -- create a URL from argv[0]. transparent marshalling
    -- is performed, to marshal the given String to the
    -- component system's StringObject type
    url <- _URLObject_ # new (head args)
    -- fetch the URL and retrieve its contents as a String,
    -- again using transparent marshalling to convert the
    -- resulting
    webpageContents <- url # fetch () >>= getContents ()
    -- display webpage contents to stdout
    putrStrLn webpageContents
    where
        _URLObject_ = getClassObject :: URLClassObject
```

the message, and there only needs to be a single point of entry into the Haskell environment, which is passed the concretised message object as a parameter. One can define a new MessageExpressionObject specifically for this purpose.

4. Depending on the component system's capabilities, it may be possible to provide a single surrogate object which can masquerade as any object or class in the component system. The Haskell environment can call functions in the surrogate component to instruct it to act as a particular object in the component system's class hierarchy. Effectively, the surrogate functions as a *factory object*, and the Haskell environment can create specific instances of the surrogate which gives it the capabilities to act as a particular component.

   If it is possible to use the surrogate component as a factory object, this eliminates the need to write different code for each surrogate component. By using the FFI foreign import ccall "wrapper" declaration[2], it is possible to provide a pointer to a Haskell function, which each instantiation produced by the surrogate factory object can use to call the Haskell environment.

5. A component written in Haskell requires some way to maintain the state of its instance variables. This can be done in one of two ways:

   (a) An object specifically created for the purpose of storing instance variables can be created. This *instance variables object* can provide the traditional get and set methods, parameterised by a String to indicate which variable to operate on. Listing 4.28 shows how a Haskell component may be written using an instance variables object.

   This approach of storing instance variables is therefore capable of storing any Haskell data type which is marshallable to the component system via the Argument type class. While this makes accessing instance variables slightly inconvenient, helper functions can be implemented in a similar fashion to the Arguments type class, to perform setting and retrieving on multiple instance variables. Note that the surrogate component can act as a *factory object* for the instance variables object.

   (b) Use the fearsome unsafePerformIO (newIORef *x*) technique [11] to enable mutable, top-level (global) variables in Haskell. The use of this technique is neither advocated or disapproved of by this thesis, and there is much debate in the Haskell community [13] about properties of its usage.

[2] A deprecated form of this declaration is foreign export "dynamic".

Listing 4.28: A Haskell component which uses an *instance variables object*

```
-- convenience functions to access instance variables.
-- these declarations could be automatically generated via
-- Template Haskell.
getCounter = getInstanceVariable "counter"
setCounter i = setInstanceVariable ("counter", i)

incomingMessage ::
    MessageExpressionObject → InstanceVariablesObject → IO ()
incomingMessage messageExpression instanceVariables
-- initialisation function called by the surrogate component
| messageRequest == "initialise" = -- or "initialize" ...
    instanceVariables # setCounter 42
-- masquerade as a "Counter" object, which only inherits from
-- the root of the class hierarchy
| messageRequest == "class" =
    messageExpression # setReply "Counter"
| messageRequest == "superclass" =
    messageExpression # setReply "Object"
-- methods to increment and decrement the counter
| messageRequest == "increment" = do
    i ← instanceVariables # getCounter
    instanceVariables # setCounter (i + 1)
| messageRequest == "decrement" = do
    i ← instanceVariables # getCounter
    instanceVariables # setCounter (i - 1)
-- returning the current state of the counter
| messageRequest == "getCounter" = do
    i ← instanceVariables # getCounter
    -- the surrogate component can read the value
    -- set in the MessageExpressionObject by the Haskell
    -- environment, and return that value to the component
    -- system. note that the setReply function is overloaded.
    messageExpression # setReply i
where
    messageRequest = messageExpression # getMessageRequest
        -- :: String
```

Chapter 5

Exception Marshalling

Exceptions are special, pre-defined events which can change a program's execution path. A division by zero is an example of an exception. It is possible that sending a message to a component can result in an exception being *thrown* (or *raised*) by the component system. The thrown exception should then be marshalled to Haskell, so that the Haskell environment can *catch* the exception and take an appropriate action.

How an exception is thrown and caught and how it is represented varies greatly from one component system to the next. As such, it is the responsibility of the communications functions in the proxy language to correctly catch any exceptions, and marshal it in a form which Haskell can interpret.

To marshal the exception to the Haskell environment, the duty of invoking a message expression is split into two functions:

1. The sendMessageExpression function shown in Listing 4.13 (page 40) is modified so that it sends the message but does not return a reply value. It instead returns an exception object if an exception was thrown, or a nullPtr if no exception was thrown.

2. A new function named getMessageExpressionReply is defined, to retrieve the message reply from the result of the message expression.

Listing 5.1 demonstrates the changes which can be made to the appropriate communications functions and type classes. For comparison, the original implementation of the sendMessage function was introduced in Listing 4.14 (page 41).

Listing 5.1: An implementation of sendMessage which marshals exceptions

```
class MessageReply r where
    getMessageExpressionReply :: MessageExpression → IO r
instance MessageReply () where
    getMessageExpressionReply = return ()
instance MessageReply Int where
    getMessageExpressionReply = getMessageExpressionIntReply
-- etc ...

foreign import ccall
    getMessageExpressionIntReply :: MessageExpression → Int
-- etc ...

foreign import ccall
    sendMessageExpression :: MessageExpression → IO UniversalObject

sendMessage :: (Arguments args, MessageReply r) ⇒
    Receiver → MethodName → args → IO r
sendMessage receiver methodName arguments = do
    e ← makeMessageExpression
    setReceiver e receiver
    setMethodName e methodName
    setArguments e arguments
    -- see section 5.10 in the FFI addendum for details
    -- on throwIf
    throwIf
        (≠ nullPtr)
        (show) -- replace with appropriate marshalling code
        (sendMessageExpression e)
    r ← getMessageExpressionReply e
    return r
```

Chapter 6

Memory Management

Haskell features automatic memory management using a *garbage collector*, which is aware of whether any components or objects are no longer being used by the Haskell environment. The FFI provides a specialised data type to represent objects outside of the Haskell context called a ForeignPtr, to which *finalizers* may be attached. Finalizers are functions which are called when the garbage collector detects that an object is no longer being referenced by the Haskell system.

If a component system uses a *reference counting* scheme to perform memory management, ForeignPtrs are a simple way to enable the Haskell system to decrease the reference count of a component when its garbage collector detects it is not being used any more. In the event that a new object enters the Haskell environment, two steps should be taken. Firstly, its reference count should be increased if this has not already been done by the component system. Secondly, the FFI function newForeignPtr should be called to attach a finalizer to the pointer and create the foreign pointer. The finalizer simply decreases the reference count of the object. If ForeignPtrs are used, the sendMessage function must also be modified so that it can act on the ForeignPtr type; it can use an FFI function such as withForeignPtr to obtain the Ptr and pass it to the communications functions in the proxy language.

One problem with this scheme is that not every type of object may require manipulation of a reference count. In particular, class objects provided by the component system are typically present throughout the lifetime of the system, and do not have reference counters associated with them. This poses a problem for the class hierarchy tree. The UniversalObject type is required to be a ForeignPtr, so that any new, unknown objects which enter the Haskell environment can have their reference counts properly decreased when they no longer being used. However, if class objects are simply Ptr types instead of ForeignPtr types, then now there are two different types of pointers. In particular, if sendMessage now only works with ForeignPtrs, how is it possible to communicate with class objects? Here are three possible solutions to this problem:

1. Two types of sendMessage functions can be created: one to send messages to objects represented as ForeignPtrs, and one to send messages to Ptrs. The sendMessage function which operates on ForeignPtrs can easily be

Figure 6.1: A class hierarchy which supports both ForeignPtr and Ptr data types. Shapes with solid outlines represent data types, and shapes with dashed outlines represent type classes. Rectangles indicate data types which are wrappers around Ptrs, and ellipses represent data types which are wrappers around ForeignPtrs.

implemented using the sendMessage function which operates on Ptrs[1].

The class inheritance tree shown in Listing 3.4 (page 23) is modified to include *two* types which may represent any object, as shown in Figure 6.1: a UniversalObjectPtr type which is a Ptr, and a UniversalObject type which is a ForeignPtr.

All class objects are upcast to the UniversalObjectPtr type before being used with the version of sendMessage that operates on Ptrs. All other objects are upcast to the UniversalObject type, for use with the sendMessage function that operates on ForeignPtrs.

Note that using this design means that there are now *two* data types which can act as the root of the class hierarchy.

2. Any retrieved class objects are upcast to the UniversalObject type before they are sent with sendMessage, and the casting functions perform the task of converting them from a Ptr to a ForeignPtr in the process. The casting functions attaches a *dummy* function as a finalizer, which simply does nothing.

3. All retrieved class objects are treated as ForeignPtrs, with a *dummy* function attached to them as a finalizer.

Each choice has their own advantages and disadvantages, but the recommend approach is to retrieve all class objects as ForeignPtrs and attach *dummy* functions as their finalizers. This approach is preferred for two reasons:

[1] Note that when the *Ptr* is retrieved from the ForeignPtr to be sent, the garbage collector may sense that was the last use of the of the ForeignPtr, and execute the attached finalizer *before* the message is sent. To ensure that this never occurs, either the withForeignPtr or touchForeignPtr functions should be used appropriately.

1. Using two sendMessage functions to handle Ptrs and ForeignPtrs requires more than one data type to represent the root of the object-oriented class hierarchy. This is both confusing for programmers and inelegant, since an object-oriented class hierarchy should never have more than one object as the root of a single inheritance tree. Additionally, there will be different representations of objects in the Haskell environment, so not all objects may be treated similarly by functions which operate on them.

2. Representing retrieved class objects as Ptrs and relying on the cast functions to convert between the Ptr and a ForeignPtr requires the casting functions to be written differently, depending on whether a class object or an instance object is being typecast. Different representations of objects are also used.

Chapter 7

MochΛ: Haskell & Cocoa

7.1 MochΛ

The ideas presented in this thesis have been used to implement a language binding between Haskell and Objective-C, named MochΛ. Objective-C uses a small number of extensions to the C language to implement a highly dynamic and reflective object-oriented programming language. It is used extensively in the Mac OS X, NeXTStep and GNUstep environments, and features two *frameworks*—*Foundation* and *AppKit*—which together allow for very rapid software development. *Cocoa* is the name that Apple® has given to the combination of the *Foundation* and *AppKit* frameworks in association with several more Objective-C classes specific to the Mac OS X platform.

7.1.1 The Cocoa classes

To model the class hierarchy, MochΛ generates four class hierarchy trees for the Haskell programmer: one to represent instance objects, one for class objects, one for meta-class objects, and one for Objective-C's *formal protocol* objects[1]. Class objects can be retrieved by prefixing and postfixing the class name with an underscore, e.g. _NSObject_, and meta-class objects use two underscores, e.g. __NSObject__. Formal protocol names are appended with Protocol, e.g. the Cocoa NSCoding protocol is named NSCodingProtocol in MochΛ. Formal protocol objects, if they are ever required, can be retrieved using the same _*classname*_ syntax, e.g. _NSCodingProtocol_.

Interface definitions are provided for the entire Cocoa framework, and on-the-fly interface generation is provided via Template Haskell, so that user-written Objective-C frameworks can easily be used when writing a Haskell program. (The Template Haskell-based interface generator was also used to generate the interface definitions for Cocoa.) Instantiating new objects is performed by sending an alloc message to a class object, and sending an init message to the resulting alloc'ed instance object. Explicit upcasting and downcasting of objects is provided, so that objects retrieved from Cocoa container objects (such as NSArray or NSDictionary) can be cast appropriately.

[1] A *formal protocol* is similar to a Java *interface* or a C++ *pure virtual class*. Formal protocols enable objects to inherit multiple interface definitions, but not inherit multiple implementation definitions.

7.1.2 Communication with Objective-C

MochΛ enables Haskell to send messages to Objective-C objects, and also enables Objective-C objects to be written in Haskell. Sending messages from Haskell to Objective-C uses the same techniques mentioned in this thesis, with the exception that an additional type class has been used in MochΛ to integrate Objective-C's *type encodings* with the communications functions.

Objective-C method names are encoded in Haskell by lowercasing the first letter and replacing any colons (:) in the method name with underscores, omitting the last colon. For example, the NSURL class' initialisation method initWithScheme:host:path: is written as initWithScheme_host_path.

The abstract MessageExpression data type used in Chapter 4 is implemented in MochΛ using the NSInvocation class, which exactly fulfils the properties required of a MessageExpression. MochΛ implements transparent marshalling between Haskell Strings and the Foundation framework's NSString class, and provides *direct messaging functions* for the entire Cocoa framework.

MochΛ uses surrogate objects to forward messages from the runtime system to the Haskell environment, and these surrogate objects are capable of masquerading as any type of object in Objective-C. As a result, it is possible for the Haskell environment to respond to actions produced by a user in a GUI interface, or even for Haskell code to function as a fully-fledged NSDocument or NSWindow controller.

7.1.3 Building Cocoa applications

The ability of MochΛ to write Objective-C objects in Haskell enables building complete GUI applications on Mac OS X. Moreover, MochΛ provides integration with Apple's Project Builder and Interface Builder programs, which together comprise an *integrated development environment* (IDE) to build Cocoa applications. This integration enables a programmer to use Interface Builder to build a GUI application's interface, and use Project Builder to directly edit and compile Haskell source code into a resulting application—all within the IDE. MochΛ is currently only implemented on the Mac OS X platform, but it should be possible to port it to other Objective-C systems such as GNUstep without much difficulty.

The homepage for MochΛ is at http://www.algorithm.com.au/mocha/. In accordance with the Haskell and Apple open-source developer and research communities, it has been provided under the liberal BSD license.

Chapter 8

Discussion & Conclusion

8.1 Summary of ideas

The ideas used in this thesis can be used to write a binding from Haskell to a reflective object-oriented programming language or component system. A complete communications framework has been given which allows low-level messaging functions to be wrapped by elegant, more convenient higher-level *direct messaging* functions.

It is possible to model component system's object-oriented class hierarchy by using the techniques discussed in this thesis. Objects can be representing in an isomorphic Haskell class hierarchy, it is possible to upcast and downcast objects in the class hierarchy, and class objects may be represented and used. To access a particular component's API, a novel approach of using Template Haskell as an *automatic interface generator* has been proposed.

There were also approaches and solutions offered for other important problems, such as transparent type marshalling, enabling components to be written in Haskell, exception marshalling and memory management. A Haskell to Objective-C language binding, MochΛ, has been written, as a concretisation of the ideas in this thesis. MochΛ enables a Haskell environment to both communicate with and act as Objective-C objects, and can be used to build full GUI applications on Mac OS X.

8.2 Conclusion

MochΛ shows that Haskell has much to gain from interacting with object-oriented languages and component systems, by enabling access to rich frameworks and libraries which would otherwise not be available. As well as simply enabling Haskell to interact with component systems, binding Haskell to these systems also enables programmers that operate in these systems to now use Haskell as an alternative programming language to write components.

8.3 Future Work

Even though this thesis has proposed a practical, universal framework for Haskell to interoperate with component systems, there are still a number of areas where the interoperability may be improved.

- Some object-oriented programming languages use *named parameters* to perform function calls or method invocations. For example, one would write [url initWithScheme:"http" host:" localhost" path:"/"] in Objective-C to invoke the url object's initWithScheme method, with the two named parameters host and path[1]

  It is currently unknown how to provide a convenient, elegant way for Haskell to call functions which use named parameters, so this is an area for future research. One promising option is a proposal for records in Haskell [14], which would allow writing url # initWithScheme ("http", { host = " localhost" , path = "/" }) to mirror the Objective-C example given above.

- The syntax to declare a function which can operate on a particular class or type class is slightly inconvenient, since a type variable must specifically be mentioned. It would be preferable if one could write :: Object → ... instead of :: Object o ⇒ o → ... for a type signature, by using the type system to expand the first type signature to the second.

- It would be interesting to use a component system to enable Haskell components to communicate with each other. Since component systems and some object-oriented frameworks have features such as dynamic loading and network transparency, Haskell could simply use the component system's architecture to allow interaction between different Haskell modules. For example, using the component system's dynamic loading facilities would enable Haskell to load new components and data types at run-time, and perhaps enable dynamic typing and binding for greater extensibility of Haskell programs.

- When Haskell is used to write components, the two approaches proposed in Section 4.7 (page 50) for storing instance variables seem ad-hoc and inelegant. More research needs to be done on how Haskell components can retain state between invocations of the component.

[1] This is not quite true—Objective-C does not, strictly speaking, have named parameters. In this example, the method name is actually initWithScheme:host:path, and the parameters for host and path are passed as normal positional parameters in an argument list. However, it would still be useful if Haskell could emulate such syntactic sugar which appears much like a function with named parameters.

Appendix A

An explanation of getClassObject

The following is an explanation of how the getClassObject function, described in Listing 3.11 (page 25), works:

1. classObjectName is called to determine the name of the desired class object. classObjectName knows which name to return since the type of the entire getClassObject expression is *fixed* at the point of the function call, by writing getClassObject :: SubClassObject.

2. The name returned by classObjectName is passed to getClassObjectFromName, to obtain the class object for the given name. The x used as the result of the function again *fixes* the result of getClassObjectFromName, so it knows exactly the type of the class object to retrieve.

3. The upcast function is used to cast the result from a class object data type to the ambiguous type variable c, defined in getClassObjects type signature. The type system is able to disambiguate the type variable, since the type of the entire expression has been fixed at the point of the function call.

Listings

3.1 A superclass and subclass in Java 10
3.2 A superclass and subclass in Haskell, modelled with phantom types 11
3.3 Java class hierarchy with multiple inheritance 12
3.4 Phantom types augmented by type classes can model multiple inheritance 12
3.5 Using only type classes to model multiple inheritance 13
3.6 Implementing and using printf in Template Haskell 16
3.7 Up/downcasting functionality for the Super object-oriented class 20
3.8 Using fromSuperInstance and toSuperInstance to perform casting . 20
3.9 The downcast and upcast functions 21
3.10 Modelling a class hierarchy which includes class objects 24
3.11 The getClassObject function 25
4.1 Definitions of the Message, Receiver, and MessageExpression data types 30
4.2 Primitive MessageExpression functions 31
4.3 sendMessageExpressionWithNoReply interface 32
4.4 Other sendMessageExpressionWith*x*reply functions 33
4.5 An ideal interface for sendMessage 34
4.6 Using ad-hoc polymorphism to write setArgument 34
4.7 Using existential types for an argument list 35
4.8 Using sendMessage with a list of existential types 36
4.9 Using Template Haskell versions of printf and sendMessage 36
4.10 sendMessage type signature with an arbitrary parameter 37
4.11 The Arguments type class 38
4.12 The Arguments type class, redux 39
4.13 The sendMessageExpression implementation using a MessageReply type class 40
4.14 The sendMessage implementation 41
4.15 Using sendMessage 41
4.16 Hashtable object manipulation in Java 42
4.17 Hashtable object manipulation in Haskell using sendMessage . . . 43
4.18 Convenient hashtable component communication in Haskell . . . 43
4.19 Implementing the put and get functions 44
4.20 The ObjectArgument type class 45
4.21 overloadedMethod in Java 45
4.22 Implementing overloadedMethod in Haskell 46
4.23 put and get, allowed only to operate on Hashtable and FiniteMap receiving objects 47
4.24 put and get implemented with the DirectMessage type class . . . 48
4.25 Transparent String ↔ StringObject marshalling 49
4.26 Implementation of the # function 50
4.27 Using # and >>= for succinct code 51
4.28 A Haskell component which uses an *instance variables object* . . 53

5.1 An implementation of sendMessage which marshals exceptions . . 55

Bibliography

[1] Finne, S., Leijen, D., Meijer, E., and Peyton Jones, S., *Calling hell from heaven and heaven from hell* (1999), International Conference on Functional Programming, 1999

[2] Finne, S., and Meijer, E., *Lambada, Haskell as a Better Java* (2000), Haskell Workshop 2000

[3] Peyton Jones, S., Meijer, E., and Leijen, D., *Scripting COM components from Haskell* (1998), Proceedings of ICSR5

[4] Shields, M., and Peyton Jones, S., *Object-Oriented Style Overloading for Haskell* (2001), Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01)

[5] Chakravarty, M. M. T., *C→Haskell, or Yet Another Interfacing Tool* (1999), Implementation of Functional Languages, 11th. International Workshop (IFL'99)

[6] Finne, S., Leijen, D., Meijer, E., and Peyton Jones, S., *H/Direct: A Binary Foreign Language Interface for Haskell* (1998), International Conference on Functional Programming, 1998

[7] Chakravarty, M. M. T., *A GTK+ Binding for Haskell*, http://www.cse.unsw.edu.au/~chak/haskell/gtk/

[8] Sheard, T., and Peyton Jones, S., *Template metaprogramming for Haskell* (2002), Haskell Workshop 2002

[9] Lynagh, I., *Template Haskell: A Report From The Field* (2003), (unpublished; available at http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/#reportfromfield)

[10] Peyton Jones, S. (editor), et al., *Haskell 98 Language and Libraries: The Revised Report*, http://www.haskell.org/definition/

[11] Hughes, J., *Global Variables in Haskell*, (unpublished; available at http://www.math.chalmers.se/~rjmh/)

[12] Chakravarty, M. M. T. (editor), et al., *Haskell 98 Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report* http://www.haskell.org/definition/

[13] (members of the haskell-cafe@haskell.org mailing list) *Global variables?*, http://haskell.org/pipermail/haskell-cafe/2003-January/003884.html

[14] Peyton Jones, S., and Morrisett, G., *A proposal for records in Haskell* http://research.microsoft.com/~simonpj/Haskell/records.html