Master of Science Thesis

wxHaskell for the web

Substituting C++ with Haskell and JavaScript

by

Ruben Alexander de Gooijer

October 29, 2012

Center for Software Technology
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

Daily Supervisor:
dr. A. Dijkstra
Second Supervisor:
prof. dr. S.D. Swierstra

Abstract

Traditionally applications were built to run on top of a desktop platform, but this is changing rapidly and the web is now becoming the default deployment platform. Especially, with the new HTML5 standard the web becomes an even more attractive platform for many hybrid client/server applications. In Haskell one of the goto libraries for building graphical user interfaces is wxHaskell. We are motivated by the idea of using the high level abstractions of wxHaskell to develop type-safe client-side web applications in Haskell. With the recent advent of a JavaScript back-end for UHC this has become an attainable goal. As a proof of concept we have ported a feature-light version of the wxAsteroids game from the original wxHaskell paper to the browser leaving the source code almost untouched. We have developed several tools that have helped us realizing the implementation. First, we improved the existing JavaScript FFI, its surrounding infrastructure, and created interfaces to the necessary platform interfaces. Second, we have developed a library for Object-Oriented programming in Haskell, inspired by OOHaskell, that contrary to OOHaskell does not dependent on functional dependencies. This library has enabled us to maintain the wxHaskell interfaces while substituting the wxWidgets C++ implementation for one written in Haskell implemented in terms of HTML5.

Contents

Chapter 1

Introduction

1.1 On a historical note

The internet has become a low cost model for delivering content and applications to end-users. Although historically based around a page-centric model there recently has been a shift taking place diverging from latter model to one wherein the page becomes a more active participant in the interaction of the user with the application [29]. Essentially taking the page-centric model and transforming it into a more desktop like experience where updates to the graphical user interface appear more or less instantaneous. To take advantage of the high reachability, centralized maintenance, and low distribution costs of the web platform many applications nowadays start out their life as web applications. Furthermore, many traditional applications that could benefit from the web platform are being rewritten such that they can be deployed on the web. A new class of Internet applications has come to light, often coined as the next generation of Internet applications or *Rich Internet Applications* [6], which try to offer the high level of interactivity that users are accustomed to from desktop applications.

Because the Internet standards were not initially built with this particular usage in mind there are many issues that needed to be resolved to develop RIAs on top of these standards. Several companies have tried to workaround the limitations by providing the extra functionality through a browser plugin [31, 64, 63]. However, with the recent resurgence in the amount of browsing platforms and the availability of more adequate standards have rendered these approaches partially redundant and or inadequate. The same capabilities are now either built in natively or can be simulated on top of the new standards [61].

Although the capabilities of web browsers have significantly improved they still lack many of the functionalities that desktop programmers have grown accustomed to. This gap in functionality is bridged by the development of graphical user interface (GUI) toolkits that offer things like layout and window management, more comprehensive widget sets, and mechanisms to integrate application data. Accessory to the development of RIAs is the increasing complexity of client-side applications. The client-side can no longer be regarded as merely a view on the application's data. It has gained a multitude of responsibilities such as the management of complex application state, synchronization of state with the server, validation of user input, etc. Consequently the client becomes tightly coupled with the server and needs to have in-depth knowledge about its data types and calling conventions. This can very quickly become a maintenance problem and a source for bugs when the different tiers (client, server, database) use different formalisms without providing some automated mapping between them [15]. The dynamically typed language JavaScript which is at the foundation of every interactive web application worsens the problem [47, 46]. Many people attempted to mitigate the problems by creating new languages [33, 10, 4], appropriate tooling [9], libraries [65, 54, 45], and any combination of these.

1.2 Motivation

The recent addition of a JavaScript back-end to the *Utrecht Haskell Compiler* (UHC) [19] opens up the possibility to create client-side web applications using Haskell. The compiler translates the UHC Core language (a minimal functional language) to a JavaScript program which may run directly in the browser (supported by a small runtime system).

The use of Haskell will likely induce a performance penalty, but in return offer many advantages over JavaScript: type safety, laziness, compiler optimizations, etc. Furthermore, thinking ahead, when augmented with a Haskell server-side component it is possible to automatically get data type consistency between client and server. This tier-less approach to programming web applications resembles that of the Google Web Toolkit (GWT) [34] and the proprietary WebSharper [8].

To move Haskell forward into the space of web programming, and in particular that of RIAs the presence of tooling for creating Graphical User Interfaces (GUI) is key. Although Haskell has already quite some existing GUI toolkits [1], none of them run in the browser. There has been a previous attempt to alleviate this problem, but unfortunately it depends on a proprietary (albeit widespread) browser plugin [56]. We have similar goals, but do not want to depend on a proprietary plugin that bypasses the web standards.

Because there are already many Haskell libraries for constructing desktop GUIs in Haskell it makes sense to retrofit an existing one such that it may run in the web browser. This allows us to benefit from years of experience constructing programming interfaces for GUI development and expedites porting existing desktop applications to the web (typically considered a large undertaking).

There are two disparate lines of programming GUIs in Haskell: using Functional Reactive Programming (FRP), or the traditional imperative event handler based approach. With FRP widgets (Window Gadgets) are typically viewed as stream processors - taking an input stream and producing an output stream. GUIs are formed by composing widgets using combinators that take two or more widgets and compose them into a single larger widget. The combinators designate how

[1] For a full list of the available GUI libraries in Haskell see `http://www.haskell.org/haskellwiki/Applications_and_libraries/GUI_libraries`

the constituent inputs are routed such that they may become outputs of the larger whole. A GUI application, from the FRP perspective, is therefore often viewed as a network of communicating widgets in which data flow is made explicit. The imperative approach is less explicit about its data flow. A typical imperative-style GUI is constructed by creating new instances of widgets, composing them in a tree-like structure, connecting callbacks to widgets allowing an application to react to event occurrences, and at the end initialize the GUI application by entering an *event loop* that detects events and dispatches them to the appropriate event handlers. This is the more traditional approach to program GUI applications and its lack of explicit data flow makes is much more flexible compared to the FRP approach at the expense of purity and ease of reasoning.

FRP is still active research and is slowly moving out of academia, but has thus far not yet really caught on as a popular way to construct GUIs. There also seems to be no general consensus on which FRP approach is best. On the contrary, the imperative approach has seen wide adoption with many GUIs constructed using it. Furthermore, existing FRP seem to often use advanced language extensions not supported by UHC which would make porting the interface arduous or maybe even impossible. Because of this practical issue and the fact that FRP is not well established as a GUI programming technique we lean towards the safe side and opt for a more imperative approach.

The most prominent imperative-style GUI toolkits for Haskell are Gtk2HS [23] and wxHaskell [42]. Both wrap existing C/C++ GUI toolkits (GTK+ [26], wxWidgets [57]) in Haskell and expose a more abstract interface. Comparatively wxHaskell offers nicer abstractions than Gtk2Hs does. The difficulty with both libraries is that they provide a mere interface to the functionality implemented in a foreign language. Porting any of them to the browser would require the reimplementation of this functionality. Because wxHaskell already works across desktop platforms and offers more evolved abstractions we choose to base our work on wxHaskell.

1.3 Research problem

The absence of an approach for programming client-side web GUIs in Haskell has led us to formulate our problem through the following research question:

- *How can wxHaskell be made to run in the web browser?*

In order to provide a suitable answer to this insidiously simple question we will investigate the different options for making wxHaskell run in the web browser, pick one, and demonstrate its viability by porting the implementation of wxAsteroids, a clone of the classic Asteroids game also used by wxHaskell to demonstrate its design and capabilities.

1.4 Outline

The outline of this thesis is as follows. In chapter 2 we provide some background information on GUI toolkits, the architecture of wxHaskell, the web browser platform, and the UHC compiler. Chapter 3 explores the different options for porting wxHaskell to the web. After picking a particular implementation path chapter 4 continues with developing the necessary tooling for interfacing with JavaScript from within Haskell. Subsequently, in chapter 5 we develop an OO programming library which we use in chapter 6 to implement a subset of wxHaskell necessary for the wxAsteroids game to work. Finally, we wrap up with a conclusion in chapter 7.

Chapter 2

Background

2.1 On the role of GUI toolkits

Before GUI toolkits existed every programmer constructed its own interface elements. Obviously the result was a lack of consistency between different user interfaces and abysmal reuse. This state of affairs led to the invention of the GUI toolkit to aid consistency and rapid development through reuse. Toolkits typically provide well integrated library of standardized widgets and a framework that deals with the low-level intricacies of graphic manipulation and event handling. The application programmer no longer needs to worry about consistency and may reuse existing interface elements. The GUI toolkit is a generic piece of software and as such does not deal with application specific logic. However, through its framework it provides the programmer with the possibility of integrating application specific functions that react to user input.

GUI toolkits have strong roots inside the object-oriented programming language (OOP) community [40], one of the many reasons why todays GUIs are more often created in object-oriented languages. Although GUI programming is not particularly tied to OOP, there is a clear correspondence between OOP concepts and those necessary to effectively model user interfaces. Object identity, state encapsulation, and inheritance all play an important role in many current GUI toolkit implementations.

Figure 2.1 shows where a GUI toolkit is typically situated on a desktop stack. Although the figure represents Unix-like environments it is quite similar to that of other operating systems. Libraries like GTK+ [26] and Qt [50] are situated at the foundational level and communicate through XLib with the X Window System [55].

The following sections describe the constituents of a GUI toolkit from a high level perspective.

2.1.1 Graphical representation

From an abstract interface description to pixels on the screen.

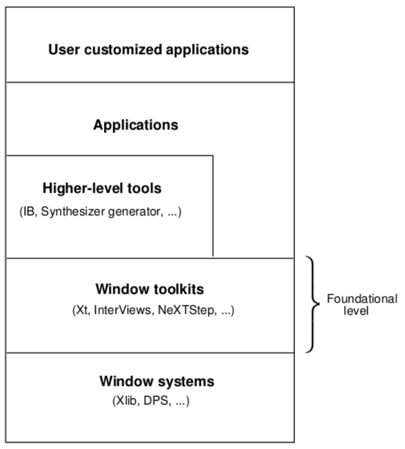


Figure 2.1: Unix desktop GUI stack [28].

An important part of what GUI libraries facilitate is the composition of an abstract interface description out of widgets. It is the responsibility of the GUI library to effectively communicate this abstract description to the underlying layer instructing a display device to display the correct image. Providing a balance between flexibility and standardization of graphical elements is key to every GUI library. Consequently widgets are typically parameterized over a large class of attributes (border style, background color, ...) allowing the programmer to influence the style and eventual location of their widgets. Often, however, the exact placement of widgets is not desirable *per se* and so called layout managers may be used that take in many widgets and provide for the automatic alignment of widgets.

By virtue of being a communication mechanism GUIs never exist in isolation but take the role of intermediary between a human and application. The host operating system may run multiple applications simultaneously of which an arbitrary number may use the screen to display their GUIs. The screen thus receives a stream of possibly interleaved drawing instructions and it suddenly becomes unclear what the end result will be. For instance, with access to the whole screen applications may erase (partially) each others GUIs by drawing at overlapping coordinates. To allow fair use of the display hardware and resolve the ambiguity of what will be displayed on screen a central coordination mechanism must be put in place.

The *X Window System* [55] is such a central coordinating system. It defines the concept of a *window*, a looking glass through which the user can interact with a computer program. In practical terms this means that it is a rectangular area on the screen capable of displaying graphics and receiving input. Every window is contained within another window. The *root window* is at the top of the window hierarchy and spans the whole screen. The direct children of the root window are called top-level windows and are treated as special by what is called a *Window Manager*. They are typically decorated with a title bar, a set of buttons (minimize, maximize, close), and may be moved around and resized. The window concept helps X with distinguishing applications allowing it to handle clipping [1] problems

[1] Clipping is a procedure that identifies if a picture is outside or inside a particular "clipping" area

effectively and implement flexible and efficient event handling mechanisms (see 2.1.2).

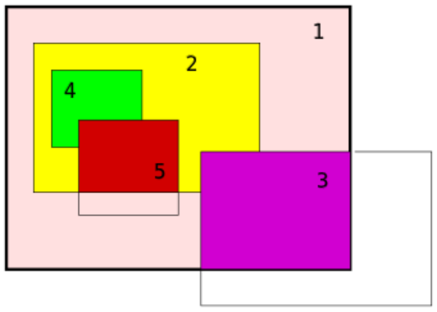
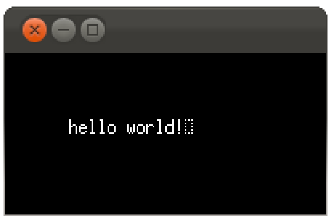


Figure 2.2: A possible placement of windows.
`http://en.wikipedia.org/wiki/File:Some_X_windows.svg`

On Unix-line operating systems the *X Window System* [**?**] is the most commonly used windowing system and many modern GUI toolkits are built on top of it [26, 50]. It provides a network-transparent window system through a client-server architecture where the server distributes user input to and accepts display instructions from its clients. The client communicates with the server and *vice versa* through the *X Window Protocol*. It is however uncommon that this happens directly and most GUI toolkits built on top of the *XLib library* [30]. Worthy of mentioning is that the client and server need not be on different machines and the standard situation on a desktop environment is that both reside on the same machine.



```
hello world!
```

Figure 2.3: A GUI running on Ubuntu 11.10 using XLib see appendix A.1.

Figure 2.3 shows the output of a traditional "Hello World" program written using XLib. What happens is that the application (client) opens a connection to the X server and requests the creation of a new top-level window. The request is intercepted by the *Window Manager* which *reparents* the window such that it becomes a descendant of a decorated window with a title bar and control buttons. The program proceeds by sending a command instructing the X server to draw the string *"hello world"* to the screen. The server delegates the request to the device driver responsible for managing the graphics hardware.

Now that we almost hit rock bottom we move back up the layers of abstraction and just above the GUI toolkit we find what are called *User Interface Markup Languages*. These languages, often dialects of *Extensible Markup Language* (XML), provide a declarative way of specifying a GUI making it easier to construct GUIs by not bothering the programmer with the distractions of the implementation language. Furthermore, the implementation of visual construction and manipulation tools becomes arguably easier. The presence of these tools ushered in a new era in which the programmer is no longer the predominant factor in the process of constructing GUIs (at least for the graphical part). There is a *scala* of different UI markup languages but some of the common ones are: XUL, XAML, and the arguably incomplete XHTML.

2.1.2 User input

From a mouse click to a button clicked event

Typically a user interacts with a computer program through input devices such as a mouse, keyboard, or touch pad. Upon interaction the device drivers should be aware of changes in device state. How this happens actually depends on the input device. It may happen through a hardware interrupt or by continuously polling for changes. A GUI registers interest in device state changes. Once a state change occurs the GUI is notified and it invokes the appropriate application logic which may perform arbitrary computations and provide the user with visual feedback. The essence of this process is clearly captured by Fruit [16], expressing the top level GUI as a Yampa [17] signal function (SF):

$$\textit{type}\ \mathit{SimpleGUI} = \mathit{SF}\ \mathit{GUIInput}\ \mathit{Picture}$$

The *GUIInput* type is a snapshot of the input device state. The *Picture* type contains the new visual representation in response to the *GUIInput*. Although accurate as a top level description it is not representative for internal widgets, whom are likely to observe additional arbitrary values:

$$\textit{type}\ \mathit{GUI}\ a\ b = \mathit{SF}\ (\mathit{GUIInput}, a)\ (\mathit{Picture}, b)$$
$$\mathit{SimpleGUI} \cong \mathit{GUI}\ ()\ ()$$

In between observing a state change at the device driver level and transforming it into a high level event a lot may happen. Take for example any GUI toolkit based on top of the X Windowing System. When X receives input it designates the *owner* window. The window abstraction makes this process efficient and straightforward. Each window is assigned a particular part of the screen, stacking order, and focus. Based on these properties X can distill which windows are visible and thus eligible for receiving events. Dependent on whether the window owner (an application) has registered interest in the particular event the event will be discarded or dispatched to the application. The application (GUI toolkit) takes notice of these events through what is called an *event loop* which dispatches events to the appropriate event handlers. Event handlers have intimate knowledge of the GUI and may interpret a low level *mouse clicked* event as a high level *button clicked* event. Although obviously a simplification, leaving out many details, it roughly corresponds to what happens.

2.1.3 Application integration

From a static user interface to an interactive application.

Widgets know how to build visual representations out of their abstract descriptions and interpret low level user input as high level semantic actions. To be of any use they allow the integration of application specific logic invoked upon the occurrence of an internal widget event (e.g. button clicked, item selected). These fragments of application specific logic are often supplied through *callback* functions, similar to how the lower level events are captured by the GUI toolkit. These callback functions, and all other code with access to the application state, may modify the state effectively rendering the widget's visual representation out of sync. The most basic GUI toolkits leave it up to the programmer to manually keep the GUI in sync with the application state.

State synchronization is not only a problem at the level of application integration, but also internally at the level of widget design. To remedy this problem typical implementations of the Model-View-Controller (MVC) pattern use the *Observer Pattern* to keep the view(s) synchronized with the model.

The MVC pattern, first described by Smalltalk [40], follows the design principle *Separation of Concerns* to separate out the distinct aspects of widgets - application state (model), visual representation (view), input events (controller) - which makes for a modular GUI library design. The MVC pattern has seen wide spread adoption (albeit in deviating forms) as a technique for developing GUI toolkits [27] [2].

The *Observer Pattern* has a strong resemblance with reactive programming techniques wherein a data flow graph is constructed such that changes can be automatically propagated to the data dependencies. Reactive programming techniques vary in explicitness about the data flow graph constructed. The Functional Reactive Programming (FRP) approach is typically very explicit about the data flow graph by using arrows to express data dependencies. Explicitness requires more thought, but allows for much better reasoning about what happens as a reaction to a certain event. Further advantages of FRP are that it is much easier to construct composite events (e.g. drap and drop) and because data flow is usually much more granular state changes may cause more efficient repainting.

An alternative approach to state synchronization is *data binding* which under the surface also uses the *Observer Pattern* but typically offers a more end-to-end approach to synchronization. It allows the programmer to create for example a data binding for a text field to a particular record in a database. The data binding will ensure that modifications to the text field's contents will automatically be propagated to the database.

2.2 wxHaskell: a quick overview

wxHaskell [42] is a GUI library for Haskell that wraps around the wxWidgets C++ library [57] offering a more declarative interface for programming GUIs. It aims to provide a library that is efficient, portable across platforms, retains a native look-and-feel, provides a lot of standard functionality, good abstractions, and is type safe where possible. The author notes that there are no intrinsic difficulties with achieving these desired properties, but that it takes a large initial effort followed by an

[2]For a list of MVC frameworks see `http://en.wikipedia.org/wiki/Model-view-controller`

enduring maintenance effort. Many previous attempts by the Haskell community to construct GUI toolkits have underestimated the amount of work that goes into maintenance and eventually turned out to fail. To avoid this pitfall wxHaskell builds on top of wxWidgets, a widely supported industrial-strength widget toolkit that eases the development of cross platform GUI applications.

The design of wxHaskell divides into four distinct increasingly abstract layers with at the bottom layer the low-level details of interfacing with C and at the top layer a declarative interface for programming GUIs:

1. **wxDirect**: responsible for generating the Haskell wrappers and foreign import declarations from C signatures.
2. **wxC**: provides the coupling of Haskell with wxWidgets. The wxWidgets FFI declarations are wrapped in Haskell functions that perform conversions between C and Haskell types and the converse.
3. **wxCore**: uses wxC to expose the core wxWidgets Haskell interface.
4. **wx**: uses wxCore to provide the user with a more declarative interface for programming wxWidgets. It also contains a combinator library to specify layouts.

In wxHaskell the development of GUI applications is centered around the imperative IO monad, something which in general Haskell programmers would like to avoid. Though because of Haskell's treatment of IO computations as first-class values the library can reach a much higher level of abstraction than can typically be attained in any other language lacking such treatment. Still, wxHaskell employs implicit data flow across event handlers through the use of mutable state.

Inside the bottom layer wxHaskell communicates with wxWidgets by using Haskell's C FFI. In order to retain type safety for widget operations wxHaskell wraps pointers to widgets inside a subtyping hierarchy by using *phantom types*:

```
type Object a = Ptr a

data CWindow a
data CFrame  a

type Window a = Object (CWindow a)
type Frame  a = Window (CFrame a)
```

Both *CWindow* and *CFrame* are considered phantom types, they lack a corresponding data declaration and thus only exist at compile time. Using type synonyms the subtyping relationship is encoded. Note that the type variable is left polymorphic, however often operations may want to specify that they expect and or produce an exact type:

```
frame :: [Prop (Frame ())] → IO (Frame ())
```

This is accomplished by applying the type to () (unit) making it monomorph. For example in the function *frame*, it takes a list of properties definable on a *Frame* or any supertype and produces an instance of a *Frame*.

Objects encapsulate state and provide methods for state manipulation. A typical pattern is to provide so called *getters* and *setters* for state manipulation. wxHaskell captures association of a particular attribute with a widget using the *Attr* type:

```
title :: Attr (Window a) String
```

A value of type *Attr* bundle both *getter* and *setter*:

```
data Attr w a = Attr (w → IO a) (w → a → IO ())
```

It does not contain any state but simply provides access to an object's accessor functions. Two helper functions are defined that allow both the retrieval of a value and the assignment of a list of values to an object:

```
get :: w → Attr w a → IO a
set :: w → [Prop w] → IO ()
```

When a value is combined with an attribute it is called a *property*. Properties are represented by the *Prop* type.

```
data Prop w =
      ∃a.(Attr w a) := a
    | ∃a.(Attr a w) :~ (a → a)
    | ...
```

Note the use of existential quantification which allows multiple properties of different value types to be stored in a homogeneous list. The following example combines all the features described thus far to capitalize the *title* of a *Window*:

```
capitalizeTitle :: Window a → IO ()
capitalizeTitle w = do
  t ← get w title
  set w [title := map toUpper t]
```

The definition of *title* is fine because its a fairly unique attribute, however an attribute like *text* is very common and shared by many widgets. Because attributes often overlap and user defined widgets might also want to reuse the same attribute wxHaskell uses type classes to model shared attributes.

```
class Textual w where
  text :: Attr w String
instance Textual (Window a) where
  text = ...
```

Where phantom types provide vertical reuse, *ad hoc* overloading provided by type classes allows for horizontal reuse. Though wxHaskell has many other features such as layout combinators, event handling, further discussion of features is postponed to the appropriate places in the upcoming chapters.

2.3 The target platform

The web browser has become a complex beast serving as a deployment platform for an ever increasing amount of web pages and applications. In this section we will provide a very brief overview of the core technologies used to built web pages and applications.

2.3.1 DOM

The *Document Object Model* is a cross-platform and language-independent standard for representing and interacting with objects defined in XHTML documents. It is the interface through which web browser expose their internal state of a web page for JavaScript to interact with.

2.3.2 Graphical representation

Web browsers offer several technologies for rendering graphics, standardized by the World Wide Web Consortium (W3C) [3]. This section provides a brief overview of the three major standards and describes their individual merits.

Canvas

The Canvas is an element in the DOM that can be used by JavaScript to perform basic drawing operations. The drawing operations are performed upon a bitmap surface that has no recollection about what is actually drawn. This makes the canvas applicable for things like animations, image manipulation, games, or anything that draws a large number of objects that are not necessarily interactive. Consequently event handlers maybe attached to the element itself but not to its contents. If such behavior is required it either has to be written from scratch or be simulated by overlaying XHTML elements. Noteworthy is that the Canvas API very much resembles the level of abstraction offered by many well known graphics APIs such as Java2D [4].

SVG

Scalable Vector Graphics (SVG) [62] is a XML markup language for creating vector graphics. SVG is a strictly higher level drawing facility than the Canvas. It remembers everything it drew inside a scene graph allowing it to be more intelligent in repainting its composites and support interactivity on every object it has drawn, contrary to Canvas. Furthermore, it tightly integrates with the DOM which makes cooperation with other web technologies such as XHTML and CSS easier.

[3]http://www.w3.org/

[4]http://java.sun.com/products/java-media/2D/index.jsp

XHTML - CSS

Cascading StyleSheets (CSS) apply visual styling to the structure of an XML HyperText Markup Language (XHTML) or to SVG. XHTML roughly defines a set of text elements, elements for attaching semantic meaning to other elements, media elements (video, audio), and a collection of form elements for user input. Each element may be have event handlers attached. The combination XHTML - CSS offers more in terms of standard facilities compared to Canvas and SVG but lacks the flexibility of both in terms of flexible drawing primitives. This deficiency is usually compensated by either using browser plugins, or more recently by embedding Canvas and SVG elements in XHTML documents augmenting the XHTML experience [5].

2.3.3 Interactive web pages with JavaScript

JavaScript/ECMAScript is a dynamically typed prototype-based language. It is the primary means for turning static web pages into highly interactive web applications. The web browser (the host environment) allows access to the elements of a web page by exposing its functionality to JavaScript through the Document Object Model (DOM). JavaScript features objects, first class - higher order - and variadic functions amongst other things [1].

2.4 UHC

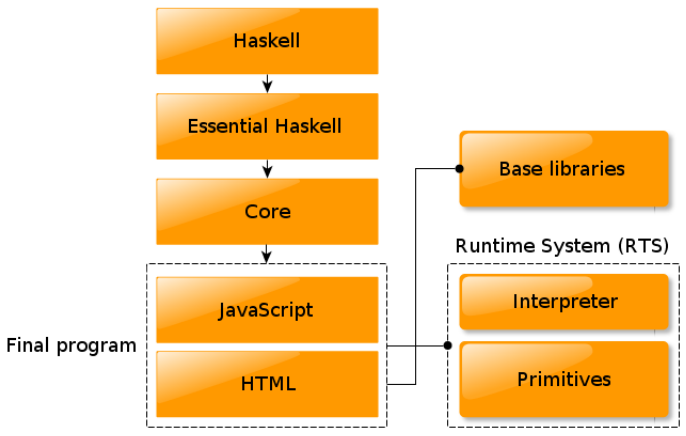


Figure 2.4: The UHC pipeline.

The Utrecht Haskell Compiler is an experimental compiler for Haskell 98 plus some additional language extensions, developed at the University of Utrecht [19]. Its purpose is more geared towards being a language experimentation platform as opposed to being a industrial-strength compiler like the Glasgow Haskell Compiler

[5]A hybrid HTML5 game, Canvas, XHTML, and CSS `http://www.cuttherope.ie/`

(GHC) [2]. Internally the compiler is organized into different language variants. Each variant may be compiled separately making language experimentation significantly easier. The compilation proceeds by pushing a Haskell through a series of transformations, expressed as algebras using the Utrecht University Attribute Grammar system (UUAG) [58], which result in intermediate languages that get progressively closer to the target platform.

Figure 2.4 displays the compilation pipeline targeting JavaScript. In the first phase the input Haskell program is desugared into Essential Haskell (EH) a desugared variant of Haskell. EH is transformed into the Core language which constitutes a very minimal functional language resembling the lambda calculus. Subsequently the JavaScript backend hooks into the compilation process and translates Core to JavaScript. It links the compiled source program together with its base library dependencies and the *Runtime System* (RTS) inside a single XHTML document.

Chapter 3

wxHaskell for the web: exploring the design space

Bridging the wxWidgets gap

The goal of this thesis is to show how a subset of wxHaskell can be ported to the web browser. However, as always, there are many routes that lead to Rome. In section 2.2 we briefly discussed the layered architecture of wxHaskell. Before we pick a particular implementation path we first consider the different ways we can cut the cake. Because wxWidgets is written in C++ and all its implementations are in terms of desktop technology there is no chance of reusing any code. Without wxWidgets, wxHaskell is just a small layer of abstractions. Somehow the gap left by wxWidgets needs to be filled with an implementation in terms the browser technology whilst trying to maintain the old programming interface. In figure 3.1 the different options are aligned next to the original wxHaskell set-up. They only differ in where the line is drawn separating Haskell from JavaScript. In this chapter we will discuss each option individually, weigh their pros and cons, and pick one approach that will dictate the further developments in this thesis.

3.1 Port wxC in Haskell (A)

The intention of this approach is to leave the original *wxC* interface intact, swap its implementation with equivalent functionality provided by some JavaScript GUI toolkit, augmented with some additional JavaScript wrapper code to overcome mismatches in functionality. With the JavaScript GUI toolkits maturing there are quite some options that provide approximately the same functionality as wxWidgets.

In order to access all the features of the underlying JavaScript GUI toolkit there is a significant amount of boilerplate code required. Assuming that we may generate this boilerplate automatically from an API description there still remains lots of subtle porting work in order to nicely fit the functionality with the *wxC* interface. From a Haskell perspective we end up in a rather strange situation wherein we are

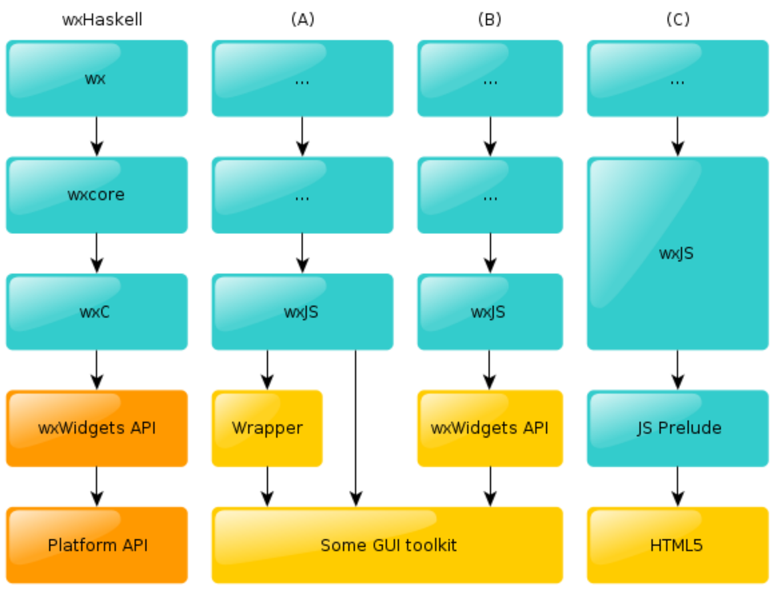


Figure 3.1: Design options: orange: C++, yellow: JavaScript, blue: Haskell

trying to sustain an imperative programming interface for an C++ API by porting its calls to a JavaScript API inside Haskell which does not natively support OO programming. Furthermore, the obtained solution will from a Haskell point of view not be very portable as the majority of the functionality is still implemented in a foreign language. Also, changing to another GUI toolkit requires a total rewrite.

Advantages:

- We get a large tested and maintained code base almost for free;
- Ideally, it is a drop in replacement for *wxC* requiring no change in the above layers.

Disadvantages:

- Connecting two OO interfaces inside Haskell will not result in idiomatic Haskell;
- The majority of the functionality is still written in JavaScript and thus cannot benefit from compiler optimizations;
- Tight coupling to JavaScript GUI toolkit results in poor portability and a high degradation risk;
- Poor extensibility from within Haskell.

3.2 Port wxC in JavaScript (B)

Option B draws the line between Haskell and JavaScript a little bit further down. The key idea is to port the wxWidgets API to JavaScript and perform the actual implementation in JavaScript. This has as advantage over *A* that it leads to a much more natural implementation. It does leave open the question whether there exists a reasonable semantics preserving mapping from C++ to JavaScript. The fact that most web browsers implement the DOM in C++ and provide a JavaScript API to access its functionality would suggest that there is. Also, the similarity between the wxWidgets JavaScript API and its C++ version will likely ease the implementation of the Haskell interface code. This can be explained by the nature of the mapping which will necessarily depend on the set of language features formed by lowest common denominator of C++ and JavaScript, ruling out the use of idiosyncratic features of JavaScript which make it particularly hard to bolt a type safe Haskell interface on top.

With a set-up much like wxHaskell it inherits many of its architectural properties. For instance, the JavaScript code base does not take part in the compilation process which complicates linking and optimization. This might lead to suboptimal code and larger binaries. Furthermore, every interaction with the wxWidgets implementation induces cross language communication inflicting a performance penalty. In wxPython (a Python wrapper for wxWidgets) widgets are extensible through inheritance just like in C++. However, in Haskell there is no direct equivalent to inheritance. This will eventually limit the flexibility of the end-user gets when using the library.

Advantages:

- Resembles the wxHaskell approach;
- Avoids many difficulties of interfacing with JavaScript from Haskell by providing a C++ like interface in JavaScript;
- See advantages of option A.

Disadvantages:

- The majority of the library is implemented in JavaScript;
- Not portable from a Haskell perspective;
- Not easily extensible from within Haskell.

3.3 Replace wxCore with a Haskell implementation (C)

We can also try to move the wxWidgets implementation as much as possible into Haskell thereby reducing the platform dependent code to a mininum greatly improving portability. With a major part of the code base in Haskell we may reap all of its benefits such as compiler optimizations, type safety, etc. Though implementing wxWidgets in Haskell is easier said than done. To stay true to wxWidgets we should port is OO design to Haskell, because Haskell was not envisioned as a OO language with typical features like: subtyping, inheritance, encapsulation; it is not clear at all to us whether this is even possible without extending the language. Fortunately, the authors of OOHaskell have shown that Haskell can indeed be used to model the typical, and some of the more advanced features of OO languages by using some common language extensions. We could use OOHaskell to transport the OO design to Haskell were it not that we are bound by UHC's features that does not yet include functional dependencies. The feasibility of the approach therefore hinges on the question whether there exists some other solution which works using UHC and is as powerful as OOHaskell or less powerful but still powerful enough to model the standard features of any OO language.

Advantages:

- Will result in a Haskell GUI library that is easier to port to other platforms besides the web platform;
- Increased extensibility with the core design writtein in Haskell;
- A larger Haskell code base can benefit from compiler optimizations, type safety, etc.

Disadvantages:

- The non-idiomatic use of Haskell will likely result in a less efficient implementation.

3.4 Conclusion

All three options are potentially viable solutions. Option *A* seems to be the least promising approach, because of its many disadvantages compared to the other two options. Option *B* is the most practical option and will most likely be directly useful. However, from a long term perspective it impairs portability of the whole code base across different target platforms/languages and does not benefit much from Haskell as language. Admittedly, this argument is weakened by the fact that the web platform is one of the most widely used standardized platforms available and will without doubt continue to be so in the foreseeable future. However, with the rate at which new GUI toolkits crop up it is hard to say which one will survive, and with a community of Haskell programmers not particulaly fond of JavaScript as a language it is hard to imagine that they would want to maintain a large JavaScript code base. Option C is also from an academic perspective a more interesting path to take, because to the best of our knowledge porting a real-world OO design to Haskell has not been done before. For the above reasons we choose to continue with option C. Next, we will look into interfacing with JavaScript of which the outcomes will also be useful for options A and B.

Chapter 4

Interfacing with JavaScript

The Utrecht Haskell Compiler can compile Haskell down to JavaScript, however for a JavaScript program to be of any real use it must be able to interface with the target platform. The Haskell Foreign Function Interface (FFI) addendum describes a framework for interfacing to foreign languages from within Haskell [12]. It instantiates the framework for the C calling convention (which should be supported by all Haskell implementations), and leaves open the possibility of extending it to other calling conventions. The C calling convention is significantly different from JavaScript's, therefore UHC has gained a new one specifically tailored for JavaScript [20].

In many aspects Haskell and JavaScript are each others opposites making it sometimes non-obvious how JavaScript functionality should be mapped onto Haskell and *vice versa*. This quickly becomes apparent when one tries to come up with meaningful type signatures for imported JavaScript functionality. Further, Haskell data types differ quite a bit from the ones in JavaScript leaving open the question on how to deal with conversions between the two different representations. When we started our research the JavaScript FFI was work in progress, and at times our efforts mingled with that of the authors of [20]. Our work can hence be viewed as a natural continuation of theirs with as goal making JavaScript programming more bearable. We make the following contributions:

- extend the existing infrastructure for programming with the JavaScript FFI;
- augment the FFI with a new keyword for creating JavaScript objects, and a simple way to incorporate external JavaScript dependencies;
- provide a model for primitive JavaScript types, together with type checking and marshalling functions.

The outline of the chapter is as follows: section 4.1 introduces the JavaScript FFI, section 4.2 discusses the possibilities of maintaining type-safety, section 4.3 describes a simple approach for converting Haskell values from and to JavaScript, section 4.4 shows how the JavaScript FFI can be used to model common JavaScript idioms.

4.1 Introduction

The UHC JavaScript FFI extends the *callconv* production of the FFI grammar, found in the Haskell FFI addendum [12], with a new keyword *js*.

```
 decl      :=  'import' callconv [safety]  impent var :: ftype
            |  'export' callconv           expent var :: ftype
 callconv  :=  'ccall' | .. | 'js'
```

Which calling convention is used determines how the compiler interprets the *impent* and *expent* strings. A formal grammar for the *impent* section is given in [20] describing a small subset of JavaScript with only few non-JavaScript parts used for expressing the connection between the formal arguments of a Haskell function and their position in the JavaScript expression. Here we present a revised version of the grammar:

```
impent   ::= "wrapper" | "dynamic" | jscc
expent   ::= "any string"

jscc     ::= ident '.js ' jsexpr     -- JS expression with external dependency
           | jsexpr
jsexpr   ::= '{}'                    -- Haskell constructor to JS Object
           | 'new'? ptrnOrId post*   -- JS expression
post     ::= '.' ptrnOrId            -- object field
           | '[' jsexpr ']'          -- array indexing
           | '(' args ')'            -- function call
args     ::= ε
           | '%*'                    -- match all arguments
           | ptrnOrId (,ptrnOrId)*
arg      ::= '%' int                 -- index a specific argument
           | literal
ptrnOrId ::= arg
           | ident

literal  ::= 'any character' | "any character"
ident    ::= letter (letter | integer)*
```

The grammar is extended with the possibility to specify external JavaScript dependencies, the *new* keyword for instantiating objects, and some ambiguity issues are resolved that arose from using specific combinations of *match all arguments* and *index a specific argument* in a single expression.

Every *jsexpr* is transformed into a valid JavaScript expression iff it is provided with a correct argument mapping (there are currently no checks in place ensuring this is the case). Furthermore, the FFI does not check whether the imported functionality is actually present at runtime. We imagine that this could be implemented in the future by inserting runtime checks or parsing external sources statically verifying the presence of the imported functionality (not a trivial problem).

With the JavaScript FFI in place we illustrate its use by a typical use case: displaying an alert message.

```
foreign import js "window.alert(%1)"
  alert :: a → IO ()
foreign import js "'Hello World!'"
  helloWorldStr :: a
main = alert helloWorldStr
```

Running *main* will result in the alert shown on the left in figure 4.1. We have imported *alert* such that it can only run inside the IO monad, because we know that it has the side-effect of displaying a message box. There is, however, nothing preventing us from importing *alert* as a pure function. It is the programmer's responsibility judge whether the imported functionality is pure or not. Also, note that the first argument of *alert* can be any value. We could just as well apply it to a Haskell string:

```
main′ = return "Hello World!" ≫= alert
```

The result of running *main′* is shown in the right in figure 4.1. Because *alert* accepts any type as argument and the representation of strings in Haskell differs from JavaScript we get garbage as output.

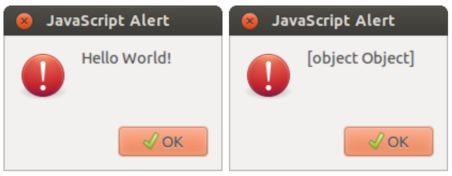


Figure 4.1: The result of evaluating *main* on the left, and *main′* on the right.

The problem with *alert* is fairly innocent, however it gets worse with a function like *plus2*:

```
function plus2(x) {
  return x + 2;
}
```

In general JavaScript functions can take any number of arguments with any number of types and in all but few cases result in a *TypeError* [60]. Conform JavaScript we leave the first argument of *plus2* polymorph:

```
foreign import js "plus2(%1)"
  plus2 :: a → IO Int
```

However, again *plus2* does not prevent us from passing in e.g. a boolean instead of a number. When we do JavaScript will coerce the boolean to a number, perform the addition, and return the result with the coercion going unnoticed. While in some cases this is intended behavior, there are many more cases where these coercions are plain programming errors. The silent coercions make it difficult to localize bugs and it gets worse when the size of the code base increases. The fact that tools such as Google Closure [9] are created, that help with type checking, proves that this is in fact a real problem. In the next section we will see how we can give functions like *alert* and *plus2* a better type signature.

4.2 Typing the Untyped

In terms of type systems Haskell and JavaScript are each others opposites. Haskell has a strong static type system, i.e. the type rules are checked at compile-time and type inconsistencies are reported to the user. A program that passes the type checker is sound with respect to the type-rules (i.e. if a program is well-typed it cannot cause type errors), hence type consistency checks can be omitted which leads to faster object code. On the other hand in JavaScript all type checking happens at runtime, types are never specified, variables derive their types from the value they point to at runtime, and type inconsistencies are resolved by implicit type conversions.

The JavaScript FFI opens up the beautifully consistent Haskell world to the unsafe JavaScript world. We would like to move away from importing JavaScript functions with all arguments left polymorph to a situation where we can be more precise about a function's type. This should lead to more idiomatic Haskell and thus allow us to benefit from static guarantees made by the Haskell type system. Before we can annotate functions with more precise types we first need a Haskell model of the JavaScript types. We imagine that these annotations may in the future be used to automatically insert runtime type checks, but for now we will resort to manual type checking. We will discuss the proposed JavaScript type model, type checking, and how to deal with union types.

4.2.1 A model for JavaScript types

The JavaScript language definition [1] describes several primitive types: undefined, null, boolean, number, and string; as well as several kinds of objects (plain objects, wrapper objects, function objects, array objects, regex objects). The primitive types bool, number, and string each have a corresponding wrapper object with an bi-directional conversion between each pair. Some operations (like +) only work for particular primitive types. When these functions receive a value of an other type than the expected type they automatically coerce the value to the expected type.

Of the primitive types we only model *undefined* and *null*. For the others we only model their wrapper object letting it range over values of the primitive type as well as their wrapper object. We piggyback on the coercion semantics of JavaScript, which ensures that we can always use a primitive type as if it were a wrapper object. Furthermore, we have $JSAny$ range over all JavaScript types. Figure 4.2 shows how the types are related to each other.

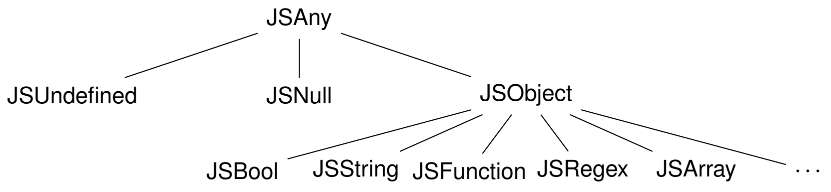


Figure 4.2: A model of the JavaScript types.

To translate the model to Haskell we make fruitful use of opaque types for *undefined* and *null*, and of phantom types to model a hierarchy of types [42, 25].

```
data JSAny a
data CJSUndefined
type JSUndefined = JSAny CJSUndefined
data CJSNull
type JSNull = JSAny CJSNull
data CJSObject a
type JSObject_ a = JSAny (CJSObject a)
type JSObject    = JSObject_ ()
data CJSBool
type JSBool = JSObject CJSBool
type JSString = JSObject PackedString
data CJSFunction a
type JSFunction_ a = JSObject (CJSFunction a)
data CJSRegex
type JSRegex = JSObject CJSRegex
type JSArray v = JSObject (BoxArray v)
```

Both *PackedString* and *BoxArray* are UHC specific types used internally for representing respectively plain strings and arrays. *JSObject_* and *JSFunction_* take an additional type parameter which can later be refined to either extend the hierarchy, or make a function type explicit. Note that we do not have a type for JavaScript numbers because all non-aggregate types like *Int*, *Float* and *Double* are shared with the JavaScript. *Integer* is the only exception, because JavaScript has no native support for arbitrary-precision integers an *Integer* value is wrapped by the RTS inside a BigInt object.

With the model for JavaScript types we can now give both *alert* and *helloWorldStr* a more precise type:

```
foreign import js "'Hello World!'"
  helloWorldStr :: JSString
foreign import js "window.alert(%1)"
  alert :: JSString → IO ()
```

The DOM defines many interfaces for communicating with the web browser. Each interface corresponds to an object in JavaScript, and we can now easily model these interface types by extending *JSObject_*. As example we use the *Node* interface:

```
data CNode a
type Node_ a = JSObject (CNode a)
type Node    = Node_ ()
```

The *nodeType* function is defined for any *Node*. Using the new type we just introduced we can easily express this by leaving the extension of the node polymorph:

```
foreign import js "%1.nodeType"
  nodeType :: Node_ a → IO JSString
```

Now *nodeType* can be applied to all objects that are *at least* of type *Node*. This is works using ordinary type unification and enables a form of subtype polymorphism.

Despite all the effort we may spend on typing JavaScript functionality there are plenty opportunities to undo any assumptions made about the types at runtime, e.g. nothing prevents us from changing the definition of *window.alert* to:

```
window.alert = undefined;
```

There is not much we can do about this and similar to the Closure compiler we are forced to make some assumptions about the runtime behavior to ensure consistency:

- all imported JavaScript functions and object properties do not change types at runtime;
- prototype chains do not change at runtime.

4.2.2 Type checking

As soon as data crosses the language border performing runtime type checks to preserve type safety becomes inevitable [5]. Take for example the *createElement* function:

```
foreign import js "document.createElement(%*)"
  createElement :: JSString → IO (Element ())
```

Dependent on which string we pass to *createElement* we get back a different subtype of *Element*. For this function to be useful we must be able to determine the actual return type by discriminating on the value's type at runtime. Once we have verified the value to be of a particular, more specific, type we may interpret it as such using a cast.

One approach to implement runtime type checking of JavaScript values would be to reuse the *Data.Dynamic* machinary which works by packing together a value with its type representation *TypeRep* in a data type called *Dynamic*. A *Dynamic* supports type safe projection of its contained value by comparing its *TypeRep* against an expected *TypeRep*. The difficulty lies in constructing a (*TypeRep*) for a given JavaScript value. There are two design alternatives: delegate the task to the runtime system or perform the mapping inside Haskell. The first approach leaks knowledge of data type compilation into the RTS thereby complicating it. The second, does not, but requires a larger family of type checking functions from the RTS to build up the *TypeRep* in Haskell.

Both design alternatives deserve further exploration, but we leave this as future work and for now resort to a much simpler approach. We simply extend the RTS with a family of type checking functions that given a value return either true or false dependent on whether the value's type matches the expected type.

$isNull, isUndefined, isBool, isString,$
$isChar, isInt, isDouble, isFloat, isNumber, isObject, isFunction :: a \rightarrow Bool$

The type checking functions are implemented in terms of *typeof*, which given a value returns a string describing its type. Here are the implementations of *isFunction* and *isBool*:

```
primIsFunction = function(a) {
  return PrimMkBool(
    typeof a === "function"
  );
}

primIsBool = function(a) {
  return PrimMkBool(
       typeof a === "boolean"
    || _primIsA(a, Boolean)
  );
}
```

```
foreign import js "primIsBool(%*)"
  isBool :: a → Bool
foreign import js "primIsFunction(%*)"
  isFunction :: a → Bool
```

The implementation reflects our decision to treat primitive and wrapper types as one and the same. We use *PrimMkBool* to directly create values of the Haskell type *Bool*. It embodies knowledge about how the compiler represents data types; like many other functions defined in the RTS. The *primIsA* function checks whether an object is exactly of some type by inspecting its constructor value.

```
_primIsA = function(a, b) {
  if(typeof a === "object" && a !== null && typeof b === "function") {
    return a.constructor == b;
  }
  return false;
}
```

To transitively test whether an object is of particular type we implement another function in terms of *instanceof*:

```
primIsInstanceOf = function(a, b) {
  if(typeof a === "object" && typeof b === "function") {
    return PrimMkBool(a instanceof b);
  }
  return PrimMkBool(false);
}
```

Using *primIsInstanceOf* we can implement the *cast* function we mentioned earlier on which guards the type conversion with a type check.

```
class JSCtor a where
  jsCtor :: a → b
cast :: JSCtor b ⇒ a → Maybe b
cast a :: Maybe b =
  if instanceOf a (jsCtor (⊥ :: b))
    then Just (unsafeCoerce a)
    else Nothing
```

```
foreign import js "primInstanceOf(%*)"
  instanceOf :: a → b → Bool
instance JSCtor (HTMLDivElement ()) where
  jsObjectConstructor _ = htmlDivelementType
foreign import js "HTMLDivElement"
  htmlDivelementType :: HTMLDivElement ()
```

With *cast* we can define a function *createDivElement* for creating *HTMLDivElement*s in terms of *createElement*.

```
createDivElement = do
  e ← createElement divString
  case cast e :: Maybe (HTMLDivElement ()) of
    Just x    → x
    Nothing → error "Something went wrong"
```

4.2.3 Representing union types

Many JavaScript functions take/return an union of types, best illustrated by an example:

```
function foo(b) {
  if(b) {
    return "an";    (1)
  } else {
    return false;   (2)
  }
}
```

The if statement works for any type by coercing its argument to a value of type *boolean*. Dependent on which branch is taken the result of *foo* will either be of type *string* (1) or *boolean* (2). Keeping up the spirit of providing type annotation *foo*'s type can be best described by:

```
foo :: a -> JSBool + JSString
```

where it takes any type to an union containing either a *JSBool* or *JSString*. The question is: how do we effectively represent a union of types in Haskell? In the following section we will discuss three different approaches: hand-made universes, dynamics, and extensible sums.

```
foreign import js "foo"
  _foo :: a → ?
```

Hand-made universes

The standard Haskell library comes with the *Either a b* data type with which we can represent a binary union.

```
data Either a b = L a | R b
```

We can use *Either* to wrap *_foo*, scrutinize the return value using a type check, and associate it with either a *L* or *R* tag. The use of *unsafeCoerce* is unavoidable because we gain knowledge about the types *at runtime* that cannot be legitimized at compile-time.

```
foo :: a → Either JSString JSBool
foo a =
  let ret r | isString r = L (unsafeCoerce r)
            | isBool   r = R (unsafeCoerce r)
  in ret (_foo a)
foreign import js "foo(%1)"
  _foo :: a → b
```

Even though the above encoding works fine it does not generalize nicely to n-ary union types. Especially the manual injection and projection of nested *Either* data types quickly becomes cumbersome.

Dynamics

Using the *Data.Dynamic* library we can hide values of different types in a single value of type *Dynamic*. To make the running example a bit more interesting we cook up a new function with a slightly more complicated type (implementation is not important):

```
bar :: JSNull + Int → JSBool + JSString
```

The argument and result type of *bar* collapse into a *Dynamic* type.

```
bar :: Dynamic → Dynamic
bar d =
  let jsVal =
    case fromDynamic d :: Maybe JSNull of
      Just v    → unsafeCoerce v
      Nothing → case fromDynamic d :: Maybe Int of
                      Just v    → unsafeCoerce v
                      Nothing → error "impossible"
    ret r | isString r = toDyn (unsafeCoerce r :: JSString)
          | isBool   r = toDyn (unsafeCoerce r :: JSBool)
  in ret (_bar jsVal)
foreign import js "bar(%1)"
  _bar :: a → b
```

To invoke *bar* we pre- and suffix it with dynamic unwrapping and wrapping calls.

$$(\mathit{fromDynamic} \circ \mathit{bar} \circ \mathit{toDyn})\ \_\mathit{null} :: \mathit{Maybe}\ \mathit{JSString}$$

Unfortunately, the projection (*fromDynamic*) and injection (*toDyn*) functions only work on monomorphic types that are instances of Typeable. Because *_bar* does not care about the type of its argument we can freely use *unsafeCoerce* as an escape hatch making the type system forget that it actually knows the type of v. In the result position we can use the same trick to let the type system learn about the new types.

Using dynamics is significantly simpler when dealing with n-ary union types compared to hand-made universes. However, there are several shortcomings:

- limited to injecting and projecting monomorphic types;
- the *Dynamic* type has no descriptive value.

Unfortunately, the fact that dynamics are limited to monomorphic types does not align well with our encoding of objects. Suppose we have a function with type:

$$\mathit{getNodeType} :: \mathit{Node}\ a + \mathit{JSNull} \to \mathit{IO}\ \mathit{Int}$$

A value of type *Node a* cannot be injected into a *Dynamic*. What we can do is temporarily make the type monomorph by flagging the type parameter position with a special data type.

```
data TyVar
  deriving Typeable
```

Using a pair of injection and projection functions we can turn a polymorphic into a monomorpic type and the other way around.

```
prjNode :: Dynamic → ∃a.Maybe (Node a)
prjNode = unsafeCoerce ∘ (fromDynamic :: Dynamic → Maybe (Node TyVar))
injNode :: Node a → Dynamic
injNode = toDyn ∘ (unsafeCoerce :: Node a → Node TyVar)
```

The *prjNode* function returns a value of type *Maybe* (*Node a*) where the type variable a is existentially quantified over preventing the caller from instantiating it to anything other than a polymorphic type variable. Unfortunately, the function pair is type specific, and we would like to abstract from the specifics using a type class:

```
class Iso f where
  inj :: f a → Dynamic
  prj :: Dynamic → ∃a.Maybe (f a)
```

However, it turns out that this does not help much as the nested types cannot be directly used to create instances of *Iso* for. First, they need to be wrapped inside a newtype such that they can be partially applied.

```
newtype One   a x     = One   {unOne   :: a x        }
newtype Two   a b x   = Two   {unTwo   :: a (b x)    }
newtype Three a b c x = Three {unThree :: a (b (c x))}
 ...
```

The programmer still needs to manually inject and project its type in and out of a member of the newtype family, nothing is gained in terms of usability.

```
instance (Typeable1 a, Typeable1 b) ⇒ Iso (Two a b) where
  inj   = toDyn ∘ (unsafeCoerce :: a (b x) → a (b TyVar)) ∘ unTwo
  prj d =
    case fromDynamic d :: Maybe (a (b TyVar)) of
      Nothing → Nothing
      Just x   → Just (Two $ unsafeCoerce x)
```

Extensible unions

Using type classes binary unions can be generalized to n-ary union types with automatic injection and projection functions [44, 59]. The trick is to rely on a right-associative nesting of types, and let type class instances generically traverse the type structure to either inject or project a type. First, we define a binary union:

```
data a + b = L a | R b
infixr 5 +
```

Using nested constructor applications of this data type we can write, e.g., a ternary union:

```
R (R True) :: Int + String + Bool
```

The injection and projection functions of the *SubType* type class automatically find value level injections and projections for values of type +.

```
class SubType sub sup where
  inj :: sub → sup          -- injection
  prj :: sup → Maybe sub -- projection
```

The implementation of *inj* and *prj* is covered by the following instances:

```
instance SubType a a where
  inj = id
  prj = Just
instance SubType a (a + b) where
  inj       = L
  prj (L x) = Just x
  prj _     = Nothing
```

```
instance (SubType a c) ⇒ SubType a (b + c) where
  inj         = R ∘ inj
  prj (R x) = prj x
  prj _       = Nothing
```

The first instance states that *SubType* is reflexive. The second instance states for injection that if we have a value of type a we can inject it into $a+b$, and for projection that provided with a value of type $a + b$ we can project out its value if it matches L. The third instance asserts for injection that provided we can inject a value of type a into c we can also inject a into a larger type $b + c$ by composing the first injection with an additional R, and for projection that provided we can project a out of c we can also project out a from a larger type $b + c$ if its value matches R.

Using extensible unions we may rewrite *bar* such that its type becomes much more informative compared to the dynamics approach.

```
bar :: JSNull + Int → JSString + JSBool
bar a =
  let jsVal =
    case prj a :: Maybe JSNull of
      Just v    → unsafeCoerce v
      Nothing →
        case prj a :: Maybe Int of
          Just v    → unsafeCoerce v
          Nothing → error "impossible"
    ret r    | isString r = inj (unsafeCoerce r :: JSString)
             | isBool   r = inj (unsafeCoerce r :: JSBool)
  in ret (_bar jsVal)
```

Similar to the dynamics approach a call to *bar* should be wrapped with projection and injection functions:

```
(prj ∘ bar ∘ inj) _null :: Maybe JSString
```

It gets interesting if union types also contain polymorphic types:

```
getNodeType :: Node a + JSNull → IO Int
```

Suppose we want to call *getNodeType* with a value of a type that unifies with *Node a*. Injecting this value into the argument type is going to fail, unless we provide a type annotation which instantiates the type variable to a concrete type matching the given type. For example, we could apply *getNodeType* to a HTMLElement as follows:

```
getNodeType (inj (⊥ :: HTMLElement ()) :: HTMLElement () + JSNull)
```

We can even inject a polymorphic value of type *Node a* given that we provide type annotations, and use scoped type variables to ensure that identically named type variables are considered the same. The same rules hold for projection.

```
getNodeType (inj (⊥ :: Node a) :: Node a + JSNull)
```

Of the three alternatives this one is the clear winner. It requires less boilerplate compared to the first approach, and provides more informative types than the second. Also, it is more flexible than dynamics as it naturally allows polymorphic types inside a union type as long as they do not overlap.

4.3 Marshalling

Aggregate Haskell types do not map directly onto JavaScript types. The difference in data representation calls for conversions functions between Haskell data, and their JavaScript equivalent. There are cases where such a mapping is obvious, e.g. a *Bool* simply maps to the JavaScript boolean type. However, there are also cases where such a mapping is less obvious, e.g. in the case of *Maybe a*. To express the mapping between Haskell values, JavaScript values, and *vice versa* we imagine two mapping functions *haskToJS* and *jsToHask* for some of the standard Haskell types:

```
haskToJS :: Haskell → JS
haskToJS ()      = ⊥
haskToJS True    = true
haskToJS False   = false
haskToJS ""      = ""
haskToJS Nothing = null
 ...
```

```
jsToHask :: JS → Haskell
jsToHask ⊥     = ()
jsToHask true  = True
jsToHask false = False
jsToHask ""    = ""
jsToHask null  = Nothing
 ...
```

Besides the obvious mappings there are also a few interesting ones. In particular, unit (()) which is most often used as a dummy value for expressing that a function returns no meaningful result, hence it maps naturally to the JavaScript *undefined* value returned when a function has no result. Also, *Nothing* which we could have mapped to undefined, but did not because there is a subtle difference between undefined and null. An undefined value is most often used in cases where something really is undefined, e.g. when accessing a non-existent object property, whereas null can be used by the programmer to explicitly state that something is not defined very much like *Nothing*.

Because both mappings are only defined for of few standard Haskell types, and they both range over a set of types in their argument as well as in their result type, we model them using a type class:

```
class ToJS a b where
  toJS :: a → b
```

```
class FromJS a b where
  fromJS :: a → Maybe b
```

Although for the standard types there exist a bidirectional mapping this may not always be to case for very instance of either class, which is why we use separate type classes. Also, because converting from a JavaScript value to a Haskell value may go wrong the *fromJS* function is partial.

For booleans the implementation is straight-forward.

```
instance ToJS Bool JSBool where
  toJS True  = jsTrue
  toJS False = jsFalse
foreign import js "true"
  jsTrue :: JSBool
foreign import js "false"
  jsFalse :: JSBool
```

```
instance FromJS JSBool Bool where
  fromJS v =
    if isBool v
      then if jsEq jsTrue v
        then Just True
        else Just False
      else Nothing
```

The *jsEq* function is a wrapper around the JavaScript strict equality operator. Every time a *JSBool* is converted to a *Bool* some type checking is performed. For booleans this conversion is relatively inexpensive, but for list-like structures this can become much more expensive. Hence, in the future compiler support for other strings representations (using overloaded strings) is unavoidable in order to gain performance. For now the conversion of Haskell to JavaScript strings takes linear time in the length of the string.

Until now we have only considered simple Haskell types, but when interfacing to JavaScript libraries it is very useful to be able to convert a Haskell record to plain JavaScript object. The authors of [20] posited some design alternatives, and decided on a solution where with the help of the RTS a record can be converted to a JavaScript object by forcing its components WHNF. The functionality is exposed through the FFI using the "{}" notation.

```
data JSBook_
type JSBook = JSObject_ JSBook_
foreign import js "{}"
  mkJSBook :: Book → IO JSBook
data Book = Book {author :: JSString, title :: JSString, pages :: Int}
book = mkJSBook (Book {author = toJS "me", title = toJS "story", pages = 123})
```

The result of applying *mkJSBook* to a *Book* value is a simple JavaScript object:

```
{author : "me", title : "story", pages : 123}
```

The opposite conversion is left for the programmer to implement.

4.4 JavaScript Idioms

In this section we will explore how the JavaScript FFI can be used to deal with common JavaScript idioms.

4.4.1 Instantiating objects

In JavaScript objects are instantiated using the *new* keyword. Its absence in the former incarnation of the JavaScript FFI has led us to consider lifting it into a primitive function.

foreign import js "primNew('B', %1, %2)"
  $newB :: a \to b \to IO\ c$

```
function primNew(obj) {
  var args = Array.prototype.slice.call(arguments);
  args.shift();
  var l = arg.length;
  switch(l) {
    case 0: return new obj;
    case 1: return new obj(args[0]);
    case 2: return new obj(args[0], args[1]);
    case 3: return new obj(args[0], args[1], args[2]);
  }
  throw new Error('Too many arguments, not supported.');
}
```

Unfortunately, the lifted version does not scale to an arbitrary number of constructor arguments. There are other solutions such as implementing it as a method on the Function object, suggested in [18]. However, since its part of the language we simply choose to expand the JavaScript FFI. Its usage is no different from that in JavaScript:

foreign import js "new String(%*)"
  $newString :: JSString \to IO\ JSString$

4.4.2 Higher-order call

JavaScript functions can be passed as arguments to other functions. If we want to pass a Haskell function to a higher-order JavaScript function we must first wrap it as a JavaScript function.

foreign import js "twice(%*)"
  $\_twice :: JSFunction\_\ (IO\ ()) \to IO\ ()$
foreign import js "wrapper"
  $\_twice\_hof :: IO\ () \to JSFunction\_\ (IO\ ())$
$twice :: IO\ () \to IO\ ()$
$twice = \_twice \circ \_twice\_hof$

```
function twice(f) {
  f();
  f();
}
```

The *JSFunction_* type can be seen as a small box wrapped around the original Haskell function, which can be used by regular JavaScript code as if it were a normal function. However, internally the Haskell calling convention is maintained, and when it returns back into the JavaScript world its return value is evaluated to WHNF.

It is not uncommon for JavaScript functions to return a function, the *createCounter* is an example of such a function. When invoked it returns a new function that at every call increments a variable and returns it.

```
function createCounter() {
  var i = 0;
  return function() {
    return i++;
  }
}
```

We can import *createCounter* just like any other function, except that when we invoke it we use the dual of "wrapper", called "dynamic", which takes a JavaScript function and returns a Haskell function wrapping the JavaScript function.

```
foreign import js "createCounter()"
  createCounter :: IO (JSFunction (IO Int))
foreign import js "dynamic"
  mkCountFunc :: JSFunction (IO Int) → IO Int
```

We can now use *createCounter* to create a counter function, which we use to print incrementing numbers to the screen.

```
main = do
  counter ← createCounter
  let count = mkCountFunc counter
  mapM_ (λm → m >>= print) [count, count, count]
```

4.4.3 Exporting Haskell functions

When integrating with existing code it is useful to be able to call Haskell functionality from JavaScript. The FFI *export* directive provides such functionality by exporting a Haskell function under a stable name.

```
minus :: Int → Int → Int
minus x y = x − y
foreign export js "minus"
  minus :: Int → Int → Int
```

The compiler generates a wrapper around the *minus* function, and it can be called using the name given to it in the export declaration prefixed with the module name.

4.4.4 Behavior of this

In JavaScript the *this* keyword has a rather peculiar semantics different from many other OO like languages. It is different in that it is dynamic, i.e. it adapts to whatever object it is called through. This, in combination with higher-order functions, can cause problems with what *this* is expected to refer to inside the body of a wrapped function.

For example, in jQuery[1], when an event handler is triggered jQuery executes the handler with *this* set to the event source. The problem is that we do not have access to *this*, and because our Haskell callback is wrapped by the compiler such that jQuery can execute it as a normal JavaScript function the event source is also lost. There seems to be no general solution to this problem. This solution proposed in [20] is to reify *this* as an additional parameter to the callback function using a helper function:

```
function wrappedThis(f) {
  return function() {
    var args = Array.prototype.slice.call(arguments);
    args.unshift(this);
    return f.apply(this, args);
  }
}
```

We illustrate how this is done for registering click events:

```
foreign import js "%1.click(%2)"
  _registerClick :: JQuery → JSFunction (EventSource → IO ()) → IO ()
foreign import js "wrapper"
  mkCb :: (EventSource → IO ()) → IO (JSFunction (EventSource → IO ()))
foreign import js "wrappedThis(%1)"
  mkWrappedThis :: JSFunction (a → IO ()) → IO (JSFunction (a → IO ()))
registerClick :: JQuery → (EventSource → IO ()) → IO ()
registerClick jq f = mkCb f ↣ mkWrappedThis ↣ _registerClick jq
```

4.4.5 Optional arguments

In JavaScript all function arguments are optional by default. When an argument is not provided it simply defaults to *undefined*.

```
function foo(x, y) {
  if(!y) {
    y = 0; // default value
  }
  return x + y;
}
foo(3);
```

The closest correspondence in Haskell to optional arguments is an argument of type *Maybe*, or when there are many options a record with defaults for every selector. The easiest way to deal with JavaScript functions with optional arguments is to import several versions of the same function:

[1]http://www.jquery.com

```
foreign import js "foo(%1)"
  foo1 :: Int → Int
foreign import js "foo(%1, %2)"
  foo2 :: Int → Int → Int
```

Although this is an easy solution it is far from pretty, and quickly explodes when the number of optional arguments increases. A better option would be to import the function with all of its arguments, and write a wrapper function that uses a *Maybe* for all optional arguments.

```
foreign import js "foo(%*)"
  _foo :: Int → a → Int
foo :: Int → Maybe Int → Int
foo a Nothing = _foo a jsUndefined
foo a (Just x) = _foo a x
```

4.4.6 Global state

JavaScript is at its core a language with mutable state, i.e. at each statement the value pointed to by a variable may change. The ability to import global state is a practical necessity. In Haskell to goto model for mutable state are IORefs. However, they are meant for modeling mutable references to immutable Haskell values, not references to mutable JavaScript values. We want changes to the global state to immediately reflect in our reads, i.e. in effect two consecutive reads of the same piece of global state may yield entirely different values.

We create an interface, very similar to that of IORef, for importing mutable JavaScript state. The differences lie in the creation of a mutable reference, and the ability to distinguish between read and read-write references.

```
-- Wraps a getter and setter
data Lens a = Lens (IO a) (a → IO ())
-- Use a phantom type as flag for read or read and write capabilities
newtype JSRef t a = JSRef (Lens a)
data Read
data ReadWrite
newJSRef         :: IO a → (a → IO ()) → JSRef ReadWrite a
newReadOnlyJSRef :: IO a → JSRef Read a
readJSRef        :: JSRef t a → IO a
writeJSRef       :: JSRef ReadWrite a → a → IO ()
```

As a simple example on how *JSRef*s can be used we import a piece of global JavaScript state (x), and an accompanying mutator (mutX).

```
x = 0;
function mutX() {
   x += 10;
}
```

```
foreign import js "x"
  readVarX :: IO Int
foreign import js "mutX()"
  mutX :: IO ()
```

We disallow writes to *x*, and hence create a read-only *JSRef*. The following fragment illustrates how changes made by the *mutX*, outside the grip of Haskell, are reflected in the value read through *readJSRef*:

```
globSt =
  refX ← newReadOnlyJSRef readVarX
  x ← readJSRef refX
  putStr (show x)
  mutX
  x ← readJSRef refX
  putStr (show x)
```

The *JSRef* interface is not only useful for importing global state, but also for modeling a more Haskell like interface to object properties. Furthermore, using *JSRef* instead of *IORef* for partially applying event handlers with global state solves the problem of having stale values, where the authors of [20] struggled with.

4.4.7 Variadic functions

JavaScript functions can take an arbitrary number of arguments. A typical example of such a function is the string concatenation function *concat* (a pure function). Similar to how we dealt with optional arguments we can import *concat* by importing different versions.

```
foreign import js "%1.concat(%*)"
  concat1 :: JSString → JSString → JSString
foreign import js "%1.concat(%*)"
  concat2 :: JSString → JSString → JSString → JSString
```

However, this is a poor choice as it does not truly uphold the semantics of *concat*. A better option would be to use the JavaScript *apply* function. Where *apply* is defined as:

```
fun.apply(thisArg[, argsArray])
```

Its first argument is where *this* is going to point to when *fun* is called, and the second argument is an array with function arguments. Using `apply` we can rewrite *concat* such that it works for an arbitrary number of arguments.

```
foreign import js "%1.concat.apply(%*)"
  _concat :: JSString → JSString → JSArray JSString → JSString
concat :: JSString → JSArray JSString → JSString
concat x xs = _concat x x xs
```

While we think this to be an acceptable encoding it still does not truly encode variadic functions. It has been shown that variadic functions can be simulated in Haskell using type classes[37, 7], but they are not commonly used and we instead stick with the more lightweight approach.

4.5 Linking JavaScript libraries

Web applications are constructed using a multitude of technologies. They use HTML in combination with CSS to convey rendering information to the browser, and use JavaScript for adding interactivity to an otherwise static rendering of the HTML tree. The technology triad constitutes the corner stone of every web application, which is served to the end-user by means of a HTML document that links all necessary JavaScript and CSS resources together.

In the current UHC pipeline, shown in figure 2.4, a Haskell program is compiled down to JavaScript, and linked into a single HTML file together with all its module dependencies, and the RTS. Without optimizations UHC uses a HTML script tag to link each dependency into the HTML file. With whole program linking turned on it links all dependencies into a single file.

The compilation pipeline delivers a very basic web application. There are, however, many possible configurations to packacke a web application. It need not even be a single binary, but may be spread over several independent units that may be loaded using variety of linking strategies. Furthermore, how a web application is assembled and deployed depends very much on the type of web application. Of all these aspects UHC currently does not address:

- external JavaScript dependencies;
- inclusion of CSS files;
- inclusion of HTML markup;
- post-processing.

Although it is possible to let UHC deal with all these issues we deem it not wise to do so. The purpose of UHC is to compile Haskell to JavaScript, and the different concerns of assembling, post-processing, and application distribution should be the task of some other software product. In the future UHC should no longer generate a HTML file itself, but produce a manifest containing a list of dependencies with which other tools can create a web application.

Contrary to the assemble process, the specification of external JavaScript dependencies is a something UHC should allow for. JavaScript FFI declarations import functionality that may depend on the presence of some external JavaScript library. To make the compiler aware of external dependencies there should some interface for conveying this information to the compiler. The dependencies could be specified in a special dependency file, but this requires a new file format, and adds to the semantic distance between the JavaScript FFI declarations and the supporting JavaScript code. A better approach would be to reuse the existing infrastructure and specify the dependencies at either module or function level.

```
-- Module level
{-# INCLUDE "jquery.js" #-}
module JQuery where
-- Function level
foreign import js "jquery.js %1.append(%*)"
  append :: a → b → IO ()
```

Specifying dependencies at the module level has as advantage that is it not necessary to repeat it for every function. However, this ease of use comes at the cost of loosing granularity in the linking process. Also, for flexibility reasons, the task of resolving the filename to an absolute location should be a responsibility of the compiler (search paths should be supplied as a compiler option).

We have implemented a proof of concept for the function level interface, because GHC has deprecated the language pragma, and the function level interface provides more granularity. We found that there was no infrastructure present for letting *Haskell Interface* (HI) files carry external dependencies of any sorts. Hence, in the current implementation only dependencies specified in the *Main* module will be considered during the linking process. In the future this should, evidently, be extended to all modules.

4.6 Related work

The omnipresence of JavaScript makes it an attractive target language. There have already been many attempts at compiling languages to JavaScript[2], of which the Google Web Toolkit (GWT) [34] (Java to JavaScript) has undoubtedly seen most traction among the commercial programming community. Unlike GWT the compilation of Haskell to JavaScript is still very much in its developing stages. Especially the FFI to JavaScript is still under developed. In the following sections we will discuss some of the more prolific attempts at compiling Haskell to JavaScript, and in particular how their FFI implementation compares to UHC's.

4.6.1 York Haskell Compiler

YHC was the first to compile Haskell to JavaScript [3]. It translated the intermediate Core language to JavaScript (similar to UHC), and had tool support for converting IDL definitions to Haskell, emulation of threading on top of *window.setInterval* and CPS, exception handling, an abstraction layer on top of DOM functionality, and a library for building widgets with inter-widget communication based on [49]. As a way to communicate with JavaScript it used a special function called *unsafeJS*, which for example could be used to convert some value to a string:

```
unsafeToString a = unsafeJS "return new String(exprEval(a));"
```

[2]A listing of languages that compile to JavaScript: `https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS`

The *unsafeJS* function is passed a string containing a JavaScript expression, where the function parameters are brought into scope in the JavaScript expression under an identical name. The same mechanism was used to implement primitive RTS operations:

```
global_YHC'_Primitive'_primIntegerAdd a b =
  unsafeJS "return exprEval(a) + exprEval(b);"
```

Where in YHC the runtime evaluation strategy leaks into the FFI, UHC hides it from the programmer through its FEL. Unfortunately, due to amount of work that comes with maintaining a compiler, and the recognition that GHC is the leading Haskell compiler, the authors have decided to discontinue support for the YHC project.

4.6.2 GHCJS

The GHCJS project generates JavaScript based on the STG output it gets by hooking into the GHC compilation pipeline. It appears as if the focus of the project thus far has been mainly on the compilation part, and not so much on the FFI. Its FFI is rather primitive, and piggybacks on the C calling convention:

```
foreign import ccall "logResult"
  logResult :: Ptr JSObject → IO ()
```

There, however, seems no support for anything else but function calls. Also, being no experts on the possibilities of GHC hooks, we imagine that the decision to overload the C calling convention is born out of necessity. Modifying the GHC front-end to add new syntax will probably require a fork, which is a severe price to pay. This is where the first-class compiler support for a JavaScript back-end, like with UHC, really shines as it provides for maximum flexibility.

4.6.3 Haste

Haste [21] was born out of dissatisfaction with the pre-existing Haskell to JavaScript compilers. Similar to GHCJS it hooks into the STG phase of GHC, however it does make quite a few different design decisions. For instance, it chooses to not support concurrency, leave out on-demand code loading, and use a symbolic intermediate between STG and JavaScript to make many simplifications and optimizations possible.

Haste also uses the C calling conventions for interfacing with JavaScript:

```
foreign import ccall foo :: Int → IO JSString
```

The author also shows that the C calling convention can be used to model callbacks.

```
foreign import ccall cb :: (JSString → IO ()) → IO ()
```

However, the programmer needs to be careful when invoking the callback. A lot of RTS details shine through at this point:

```
function cb(callback, _state_of_the_world) {
    A(callback, [[1,'Hello, world!'], 0]);
    return [1, "new state of the world"];
}
```

Although *dynamic* and *wrapper* can be both implemented using this functionality there is no syntax and automatic wrapping/unwrapping support. Finally, Haste allows external JavaScript dependencies to be included, not based on FFI imports, but simply by providing it as a compilation parameter.

4.7 Conclusion, Discussion & Future Work

In this chapter we have continued the work of [20] by extending the existing infrastructure for programming with the JavaScript FFI. We did this by providing a model for JavaScript types in Haskell, together with type checking and marshalling functions, further we augmented the FFI with a new keyword for creating JavaScript objects, and a simple way to incorporate external JavaScript dependencies.

There is, however, much work to be done before the JavaScript back-end is ready for prime time. The inefficiencies caused by mismatches in data representation should eventually be solved by the compiler, e.g. for strings this could be done by implementing overloaded strings. Also, the large number of thunks generated by Haskell programs are a major cause performance problems in the web browser. Support for strictness annotations and analyses in UHC would likely improve this situation. Also, more research is necessary into what the best intermediate representation is for generating JavaScript code. The decision to compile from Core to JavaScript has not been made because it is the best match, but for reasons of simplicity. Besides the performance issues it is definitely worth the effort to look into what it takes to support: automatically generated FFI definitions, concurrency, asynchronous server calls, *Data.Dynamic* as library for type checking JavaScript, automatic insertion of type checks based on FFI type annotations, exceptions, and better error reporting.

Chapter 5

A lightweight approach to Object-Oriented programming in Haskell

We are motived to explore the possibilities of Object-Oriented (OO) programming in Haskell by our desire to port wxWidget's design to Haskell. While Haskell was not originally envisioned as a language for OO programming there have been several attempts at forging it into a OO language either by: extending the language with subtype polymorphism [51], or embedding a DSL (OOHaskell) with the help of some common language extensions [38]. The latter approach has shown that Haskell's type-class-bounded and parametric polymorphism together with multi-parameter type classes and functional dependencies is expressive enough to model even the more advanced features of modern OO languages.

Because we do not intend to extend Haskell our interest goes out to OOHaskell. The model used by OOHaskell to encode objects and their types is based on polymorphic, extensible records of closures, and favors encapsulation by procedural abstraction over existential types [53]. Haskell records are not polymorphic and extensible, hence OOHaskell makes heavy use of the HList library [39] which models polymorphic extensible records through advanced type-level programming using functional dependencies. In their paper the authors also discuss several more primitive OO encodings. With one standing out in particular as bearing most resemblance with OOHaskell. It is described in section 3.4 *"Mutable objects, with tail polymorphism"* and attempts to reify extensible records on top of regular records by leaving the so called tail of the record polymorph. After discussing the alternative encodings the authors venture into the more involved aspects of OO programming using the superior HList encoding. Several subjects such as code reuse, casting, self-returning methods, and more advanced forms of subtyping are left undiscussed for the more primitive encodings discharging them as: involved, requiring lots of boilerplate, or even infeasible.

It would make sense to reuse OOHaskell were it not that UHC does not yet support functional dependencies. Driven by the practical necessity of a working OO approach not dependent on functional dependencies we attempt to stretch the possibilities of the *"Mutable objects, with tail polymorphism"* approach and try to make its limitations manifest by submitting it to the different scenarios OOHaskell is submitted to. The outline and examples in this chapter will therefore, to a large extent, be shamelessly based on OOHaskell.

Our contribution consists of an extensive exploration of the *"Mutable objects, with tail polymorphism"* approach where we augment the original approach with a generic up and downcast operation and a combinator for expressing inheritance. Furthermore, we generalize the approach to a restricted form of parameterized classes. All results are bundled inside a ready to use library which comes with useful combinators and macros for deriving some of the boilerplate.

5.1 Introduction

5.1.1 What is Object-Oriented programming?

The fundamental concepts of OO programming originated in Simula 67 [35]. Later Smalltalk [32] extended and refined the concepts of Simula 67 by treating everything as an object (even a class) and uniformly interpreted all operations as passing messages to objects. Since Simula 67 and Smalltalk, OO languages have evolved and many varieties exist today which would make it pretentious to suggest we can provide a fitting answer to the section's title. However, there exists a common conception among researchers about what features are typically found in OO languages. According to Benjamin C. Pierce [52] the fundamental feature set consists of:

1. **Multiple representations**. Objects with the same interface may use entirely different representations, i.e. an object interface is an abstract representation of the many possible instantiations. Method invocation works irrespective of the object representation.

2. **Encapsulation**. The internal representation of an object is hidden such that only its methods may access it.

3. **Subtyping**. The type of an object can be described by its interface, nominal name, or both. Often we want to write functionality which depends only on a part of an object's type. It would be too restrictive if we limit the functionality to work on *exactly* one object type. Subtyping loosens this restriction allowing functionality to work for many types as long as the expected type is related to the given type through the subtyping relation.

4. **Inheritance**. It is common for objects to share behavior with other objects. Inheritance is a mechanism which allows a particular form of behavior sharing; it accomplishes this by allowing the incremental extension of classes with the possibility to override pre-existing behavior. A class acts as a template for object instantiation. By extending a class we obtain a subclass which is just a regular class.

5. **Open recursion**. Typically, an OO language allows the body of an object method to refer to other methods of the same object using a special identifier usually called *this* or *self*. In combination with inheritance it is essential that *self* is *late-bound* allowing it to refer to methods defined at a later point.

5.1.2 Outline

In the section 5.2 we provide a high-level overview of the library. In section 5.3 we incrementally develop the type-independent part of the library. In section 5.4 we look at the library from a type perspective and develop generic casting operations, discuss self-returning methods, and generalize the approach to a restricted form of parameterized classes. In section 5.6 we conclude with a discussion about the usability and efficiency of the library and provide some directions for future work.

5.2 The ‘shapes‘ example

The ‘shapes‘ example [1] combines the typical aspects found in OO languages into a single crisp benchmark.

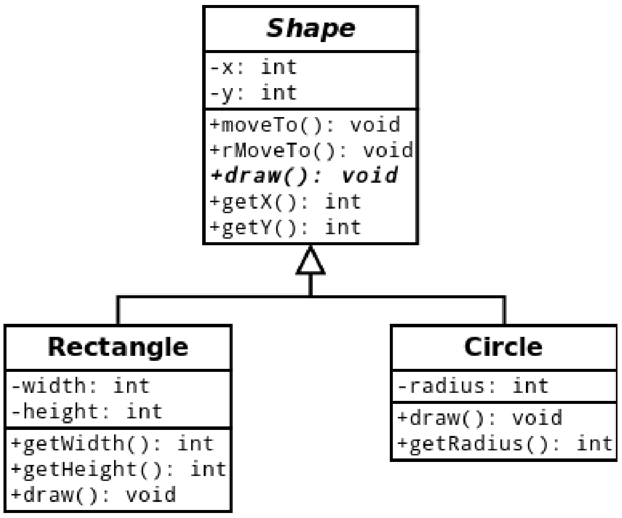


Figure 5.1: An UML diagram for the shapes example. The boxes are subdivided into three compartments. The top-level compartment contains the class name, beneath it is a list of member variables prefixed with +/- respectively public or private, and at the bottom a list of methods. The arrows indicate an inheritance relationship, and bold faced text denotes an abstract method.

Figure 5.1 shows the abstract class *Shape* with two concrete subclasses *Rectangle* and *Circle*. A *Shape* maintains a position and provides methods to directly *moveTo*

[1] See `http://onestepback.org/articles/poly/` for a multi-lingual collection of implementations in both OO as well as non-OO languages

a new position, move relative to current position *rMoveTo*, or *draw* the *Shape* in question. *Rectangle* and *Circle* augment their superclass with additional geometric data and implement the *abstract draw* method. To exercise subtype polymorphism different kinds of shapes are placed inside a collection containing shapes. The collection is then iterated over drawing each individual shape.

We first show the implementation of the ‘shapes‘ example in Java followed by an implementation in Haskell using our library.

5.2.1 Shapes in Java

The *Shape* class can trivially be translated to Java:

```
public abstract class Shape {
  private int x;
  private int y;

  public Shape(int newx, int newy) {
    x = newx;
    y = newy;
  }

  public int getX() { return x; }
  public int getY() { return y; }
  public void setX(int newx) { x = newx; }
  public void setY(int newy) { y = newy; }

  public void moveTo(int newx, int newy) {
    x = newx;
    y = newy;
  }

  public void rMoveTo(int deltax, int deltay) {
    moveTo(getX() + deltaX, getY() + deltay);
  }

  public abstract void draw();
}
```

The *Shape* constructor receives an x and y coordinate of type *int* and assigns them to its private member variables. The *draw* method is marked as abstract. Subclasses stay abstract if they do not implement *draw* or add new abstract methods.

Here follows the definition of *Rectangle*:

```
public class Rectangle extends Shape {

  // Private attributes
  private int width;
  private int height;
  // Constructor
  Rectangle(int newx, int newy, int newwidth, int newheight) {
    super(newx, newy);
    width  = newwidth;
    height = newheight;
  }

  // Accessors
  public int getWidth() { return width; }
  public int getHeight() { return height; }
  public void setWidth(int newwidth) { width = newwidth; }
  public void setHeight(int newheight) { height = newheight; }

    // Implementation of the abstract draw method
  public void draw() {
    System.out.println(
         "Drawing a Rectangle at:("
      ++ getX() ++ "," ++ getY()
      ++ "), width " ++ getWidth()
      ++ ", height " << getHeight()
    );
  }
}
```

Circle is defined similarly, we elide its full definition for the sake of brevity.

```
public class Circle extends Shape {
  Circle(int newx, int newy, int newradius) {
    super(newx, newy);
    ...
  }
}
```

Next, we put different kinds of shapes into a single collection of shapes. Inserting a *Rectangle* and *Circle* into a collection where shapes are expected exercises the language’s ability to perform subtype polymorphism.

```
Shape[] scribble = new Shape[2];
scribble[0] = new Rectangle(10, 20, 5, 6);
scribble[1] = new Circle(15, 25, 8);
for(int i = 0; i < 2; i++) {
  scribble[i].draw();
  scribble[i].rMoveTo(100, 100);
  scribble[i].draw();
}
```

We iterate over the list and draw the individual shapes to the screen.

5.2.2 Shapes in Haskell

We will now show how the shapes example is transcribed to Haskell using our library for OO programming. The library works with regular Haskell records, the *Data.Dynamic* library, and uses the C pre-processor (CPP) to derive some of the necessary boilerplate.

First, we transcribe the interface of the *Shape* class as a Haskell record. Analogous to a Java interface.

```
data IShape a = IShape {
   getX        :: IO Int
  , getY        :: IO Int
  , setX        :: Int → IO ()
  , setY        :: Int → IO ()
  , moveTo      :: Int → Int → IO ()
  , rMoveTo     :: Int → Int → IO ()
  , draw        :: IO ()
  , _shapeTail  :: Record a
}
 -- Derive boilerplate
DefineClass_macro (Shape, IShape, shapeTail, , 1)
```

Record selectors correspond to methods in the *Shape* class. There is a *single* special method *_shapeTail* for the extension of the record or as we shall call it *the tail of the record*. It is an artifact of our dependence on regular Haskell records and cannot be abstracted over. The technique of leaving the tail polymorph is known as type extension through polymorphism [11]. Finally, we use a CPP macro for deriving some of the boilerplate for manipulating records (see section 5.5).

The implementation of *Shape* is given by *shape* function:

```
 -- An implementation of the shapes interface
shape newx newy concreteDraw = clazz $ λtail self →
   -- Create references for private state
  x ← newIORef newx
  y ← newIORef newy
   -- Return a Shape
  return IShape {
     getX     = readIORef x
    , getY     = readIORef y
    , setX     = writeIORef x
    , setY     = writeIORef y
    , moveTo   = λnewx newy → do
       self # setX $ newx
       self # setY $ newy
    , rMoveTo = λdeltax deltay → do
       x ← self # getX
       y ← self # getY
       (self # moveTo) (x + deltax) (y + deltay)
    , draw     = concreteDraw self
```

```
        , _shapeTail = tail
      }
```

It takes the two initial values for x and y, an implementation for the *draw* method, an extension of the record, a self-reference, and returns an instance of *Shape* (i.e. a value of *IShape*). The *concreteDraw* parameter makes it explicit that we cannot obtain an instance of *Shape* unless we provide it with an implementation of *draw*. Consequently, we can easily create instances of *Shape* without creating a subclass – analogous to an anonymous inner class in Java. The *clazz* combinator, only used for classes with no parent class, brings two additional parameters into scope *tail* and *self*.

```
clazz cont tail self = tail ↣ λt → cont t self
```

The *tail* parameter represents the extension of the record and should only be used at the tail position, *_shapeTail* in this case. Interestingly, *self* is an explicit parameter of the function whereas in most OO languages it is implemented as an language primitive. For stylistic purposes we use some syntactic sugar to distinguish between regular functions and methods:

```
-- Reverse application
(#) :: a → (a → b) → b
o # f = f o
```

The *Rectangle* interface is transcribed similar to *Shape*’s the only difference is that we use a different macro for deriving the boilerplate.

```
data IRectangle a = IRectangle {
   _getWidth       :: IO Int
  , _getHeight     :: IO Int
  , _setWidth      :: Int → IO ()
  , _setHeight     :: Int → IO ()
  , _rectangleTail :: Record a
}
-- Boilerplate for record manipulation and subtype axioms
DefineSubClass_macro (Rectangle, Shape, IRectangle, rectangleTail, , , , 1, )
```

Here follows the implementation of the *Rectangle*:

```
rectangle x y width height =
   -- Create a new object generator by connecting the records of shape and rectangle
  (rectangle′ ‘extends‘ shape x y draw) noOverride set_Shape_Tail
  where
  rectangle′ tail super self = do
    w ← newIORef width
    h ← newIORef height
    return IRectangle {
        _getWidth      = readIORef  w
      , _getHeight     = readIORef  h
      , _setWidth      = writeIORef w
      , _setHeight     = writeIORef h
      , _rectangleTail = tail
    }
  -- The implementation of the abstract draw method
  draw self = printLn (
        "Drawing a Rectangle at:("
    << self # getX
    << ", "
    << self # getY
    << "), width "
    << self # getWidth
    << ", height " << self # getHeight
  )
```

We use the *extends* combinator in order to make *Rectangle* a subclass of *Shape*. It combines the implementation of *Rectangle* given by *rectangle′* with that of its superclass. The right-hand side of *extends* is analogous to the call to super in the Java example. In between the extension of the superclass with its subclass there is an opportunity to override functionality defined in the superclass. The function that allows for this to happen is also passed as a parameter to *extends*. Because *Rectangle* does not override any functionality we simply pass in *noOverride* which is essentially the identity function.

Notice how we left out the underscore prefix on the method invocations inside the *draw* implementation, because objects are represented as a nested records we need a helper method to invoke for instance *_getWidth*. The implementation of these helper functions corresponds to the top-down unwrapping of the record extensions until the target method is reached.

```
-- Boilerplate for explicit method lookup
getWidth  = _getWidth  ∘ unRecord ∘ _shapeTail
getHeight = _getHeight ∘ unRecord ∘ _shapeTail
-- etc.
```

In an OO language the method lookup algorithm takes care of these method lookups starting at the callee tracing the pointers until the relevant method is found. Our method lookup works the other way around by starting at the top.

We leave out the implementation of *Circle* which is conceptually no different from *Rectangle* and continue with the implementation of the scribble loop:

```
myOOP = do
  s1 ← new $ rectangle 10 20 5 6
  s2 ← new $ circle 15 25 8
  -- Create a single homogeneous list of shapes
  -- Shape is short for: IShape ()
  let scribble :: [IShape ()]
      scribble = consUb s1 (consUb s2 nilUb)
  -- Iterate over the homogeneous list with a monadic version of map discarding the result
```

```
sequence_ $ map (λshape → do
                        shape # draw
                        (shape # rMoveTo) 100 100
                        shape # draw)
  scribble
```

We use the *new* combinator to new create object instances. Unlike Java or any other OO language with subtype polymorphism we cannot simply place s_1::*Rectangle* and s_2 :: *Circle* inside a list of shapes, so we use a helper function *consUb* to automatically convert their types to *Shape* before we cons them onto the list.

```
ghci> myOOP
Drawing a Rectangle at:(10, 20), width 5, height 6
Drawing a Rectangle at:(110, 120), width 5, height 6
Drawing a Circle at:(15,25), radius 8
Drawing a Circle at:(115,125), radius 8
```

5.3 Objects in Haskell

In this section we will incrementally develop the object encoding used by our library. Similar to OOHaskell we follow the examples from the OCaml tutorial [43].

5.3.1 Objects as tail-polymorphic records

"The class point below defines one instance variable varX and two methods getX and moveX. The initial value of the instance variable is 0. The variable varX is declared mutable. Hence, the method moveX can change its value.", section 3.1

OCaml
```
class point =
  object
    val mutable varX = 0
    method getX = varX
    method moveX d = varX <- varX + d
end;;
```

In a first attempt at transcribing the one-dimensional *Point* class we map its interface to a record and let every method correspond to a selector.

```
data Point = Point {
  getX     :: IO Int
  , moveX :: Int → IO ()
}
```

The *point* function instantiates a *Point* by creating a new value of type *Point*. It models the mutable variable *varX* as an *IORef* lexically scoped over the record.

Here objects are closures of records[2].

```
point = do
  varX ← newIORef 0
  return Point {
    getX    = readIORef varX
    , moveX = λd → modifyIORef varX ((+) d)
  }
```

A method is a function which works on and belongs to an object, i.e. it has access to the state encapsulated by the object. Using *point* we can write some basic OO code:

```
myFirstOOP = do
  p ← point
  p # getX ↣ print
  p # moveX $ 3
  p # getX ↣ print
```

```
> myFirstOOP
0
3
```

There are a couple of problems with the encoding. In a typical OO language methods and data are for efficiency reasons modeled as separate entities such that methods can be shared among objects of the same type. Because efficiency is not our primary concern we stick with the conceptually simpler approach where data and methods are modeled as a single entity. Of a more pressing nature is the impossibility to extend objects with additional methods. For this reason, amongst other things, the authors of OOHaskell resort to using extensible records [39]. Instead of lifting the Haskell 98 restriction we stick with regular records and use a poor man's approach to extensible records called type extension through polymorphism [11], which can easily be modeled using a parameterized record type:

```
data Point α = Point {
  ...
  , _pointTail :: α
}
```

The *Point* type is modified to take a type parameter (α) which represents the extension/tail of the record together with a special method for manipulating it. To account for the extension of *Point* the *point* function is also modified to take an additional parameter representing a computation that will result in the tail:

```
point tail = do
  ...
  record ← tail
  return Point {
```

[2]Objects as closures in Scheme: `ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/pubs/swob.txt`

```
      ...
    , _pointTail = record
  }
```

Now that we can extend *Point* with another record we also want to have a way of closing the record extension.

```
emptyRecord :: IO ()
emptyRecord = return ()
```

The *emptyRecord* represents the end of a record extension. Like before we can construct a *point* object, but we now first apply it to the *emptyRecord*:

```
myFirstOOP = do
   p <- point emptyRecord
   ...
```

We create an extension of *Point* called *Point2D* for representing points in the 2-dimensional plane.

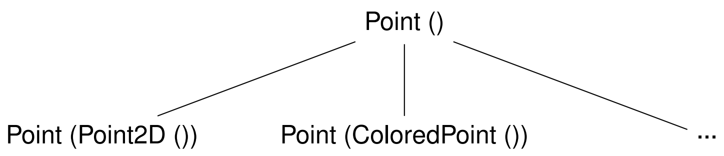
```
point2d tail =
  point point2d'
  where
  point2d' = do
    varY   ← newIORef 0
    record ← tail
    return Point2D {
      _getY          = readIORef varY
      , _moveY       = λd → modifyIORef varY ((+) d)
      , _point2DTail = record
    }
```

We omit the interface definition of *Point2D* as its signature can be easily inferred from the implementation. The fact that we pass *point2d′* to *point* clearly expresses that a 2d-point is constructed out of a *Point* linked to a *Point2D*. This becomes even clearer when we look at the type:

```
>:t point2d emptyRecord
Point (Point2D ())
```

The type structure revealed by our encoding is in fact equivalent to the phantom type structure used in [24] to model a type safe interface to external OO code, which was later formalized in[25]. However, contrary to the phantom types our encoding has meaningful Haskell inhabitants.

We have already encoded a small hierarchy of points. To accompany colored points we can simply extend the hierarchy:



Every node in the hierarchy represents an unique object type, each child node has exactly one more nested record than its parent, and there are a finite number of object types. Under these conditions the hierarchy forms a finite subtype hierarchy where each child is in a subtype relationship with its parent.

We can benefit from type unification to express the subtyping relationship to the type system:

$getX :: Point\ \alpha \rightarrow IO\ Int$

By leaving the tail of *Point* polymorph *getX* can be applied to any object type that is *at least* a *Point*. The $Point\ \alpha$ type is an abstract encoding matching a set of concrete encodings. The phantom type encoding allows abstract encodings to occur in the co-variant (producing) position [25], but because we always have values associated it is not safe to do so. Hence we limit abstract encodings to the contravariant (consuming) position.

In the previous section we showed how method lookups are performed, but left their type unspecified. In our encoding a method always expects an abstract encoding as its first argument enabling reuse across all subtypes.

```
-- Optional type
getY :: Point (Point2D a) → IO Int
getY = _getY ∘ _pointTail
```

To make our notion of interface, class, and object somewhat less vague we provide their definitions.

Definition 1. *An interface is a record C with the following shape:*

$data\ C\ t = C\ \{m, \_cTail :: t\}$

where C takes a type parameter t representing the tail, and may be instantiated to either () or a interface. In the body m expands to zero or more methods, and $\_cTail$ is a special method where c is the uncapitalized version of C.

Definition 2. *A class is a function that provides the implementation of an interface.*

Definition 3. *An object is an instantiation (value) of an interface obtained through a class.*

Thus far we have explained the basic object encoding underlying our library together with its rational. Although we will later discover that we need to slightly modify it in order to facilitate casting the basic idea will remain the same.

5.3.2 Constructor arguments

"The class point can also be abstracted over the initial value of $varX$. The parameter x_init is, of course, visible in the whole body of the definition, including methods. For instance, the method $getOffset$ in the class below returns the position of the object relative to its initial position.", section 3.1

OCaml
```
class para_point x_init =
  object
    val mutable varX = x_init
    method getX      = varX
    method getOffset = varX - x_init
    method moveX d   = varX <- varX + d
end;;
```

The previous incarnation of *point* allocated an initial value for *varX* inside its body. There is nothing restriction us from moving the initial value out of the body and turning it into an argument. Here follows a more general version of *point*:

```
para_point tail x_init = do
  record ← tail
  varX   ← newIORef x_init
  return ParaPoint {
    getX            = readIORef varX
    , moveX         = λd → modifyIORef varX ((+) d)
    , getOffset     = readIORef varX ≫= λx → return (x − x_init)
    , _paraPointTail = record
  }
```

5.3.3 Construction-time computations

"Expressions can be evaluated and bound before defining the object body of the class. This is useful to enforce invariants. For instance, points can be automatically adjusted to the nearest point on a grid, as follows:", section 3.1

OCaml
```
class adjusted_point x_init =
   let origin = (x_init / 10) * 10 in
   object
     val mutable varX   = origin
     method getX        = x
     method getOffset   = x - origin
     method moveX     d = x <- x + d
   end;;
```

Similar to OCaml we may perform computations prior (in the non-strict sense) to object construction by using a let binding.

```
adjusted_point tail x_init =
  let origin = (x_init / 10) ∗ 10
  in do ...
      varX ← newIORef x_init
      return ParaPoint {
        ...
        , getOffset = readIORef varX ≫= λx → return (x − origin)
        ...
    }
```

5.3.4 Semi-explicit parameterized classes

The *para_point* function has its argument type fixed to *Int*. This may proof unnecessarily restrictive. We can lift the restriction by introducing an additional type parameter to *ParaPoint*.

```
data ParaPoint a t = ParaPoint {
  getX            :: IO a
  , moveX         :: a → IO ()
  , getOffset     :: IO a
  , _paraPointTail :: t
}
```

The type inferencer will automatically infer a more general type for *para_point* without any modifications to *para_point*.

```
para_point :: Num a ⇒ a → IO (ParaPoint a t)
```

Parameterized points are now bounded polymorph and can be constructed using *any* type of number.

```
myPolyOOP = do
  p  ← para_point emptyRecord (1 :: Int)
  p′ ← para_point emptyRecord (1 :: Double)
  p # moveX $ 2
  p′ # moveX $ 2.5
  p # getX ≫= print
  p′ # getX ≫= print
```

If we were to apply methods of *p* with the wrong type

```
 -- Ill-typed
myPolyOOP = do
  ...
  p # moveX $ 2.5
  ...
```

we get a type error because the type checker expected the argument of *moveX* to have type *Int* but it got a *Double*. The generalization of classes to multiple type parameters is further explored in section 5.4.3.

5.3.5 Nested object generators

"The evaluation of the body of a class only takes place at object creation time. Therefore, in the following example, the instance variable varX is initialized to different values for two different objects.", section 3.1

```
OCaml
let x0 = ref 0;;
class incrementing_point :
  object
    val mutable varX = incr x0; !x0
    method getX      = varX
    method moveX d   = varX <- varX + d
end;;
```

The variable x_0 mimics what would be referred to in OO terminology as a *class variable*. The scope of a class variable is not limited to object instances, but as its name suggests ranges over all instances of a particular class. We could model x_0 as a global variable like in the fragment above. However, we can do much better by using what OOHaskell calls nested object generators.

```
makeIncrementingPointClass = do
  x0 ← newIORef 0
  return $ λtail → do
    record ← tail
    modifyIORef x0 (+1)
    varX ← readIORef x0 ↣ newIORef
    return Point {
      getX      = readIORef varX
      ,moveX = λd → modifyIORef varX (+d)
      ,_pointTail = record
    }
```

The *makeIncrementingPointClass* consists of two levels: the outer level describes the *class template*, the inner the point class. This is possible because there is nothing preventing us from returning classes instead of object instances. Classes are like objects just values unlike the case in many OO languages where they are special constructs.

```
myNestedOOP = do
  localClass ← makeIncrementingPointClass
  localClass emptyRecord ↣ (#getX) ↣ print
  localClass emptyRecord ↣ (#getX) ↣ print
```

If we run *makeIncrementingPointClass* it returns a closure over x_0. Hence, each time *localClass* is used to create a new instance of *Point* the construction-time computation increments the class variable x_0.

```
ghci> myNestedOOP
1
2
```

5.3.6 Self-referential objects

"A method or an initializer can send messages to self (that is, the current object). For that, self must be explicitly bound, here to the variable s (s could be

> *any identifier, even though we will often choose the name self.) ... Dynamically, the variable s is bound at the invocation of a method. In particular, when the class printable_point is inherited, the variable s will be correctly bound to the object of the subclass.",* section 3.3

OCaml
```ocaml
class printable_point x_init =
  object (s)
    val mutable varX = x_init
    method getX      = varX
    method moveX d   = varX <- varX + d
    method print     = print_int (s # getX)
end;;
```

Thus far we have avoided objects wherein methods refer to each other. The ability to refer to other methods inside the object is an essential feature of OO language and is enabled by a special identifier typically called *this* or *self*. The lack of such a special keyword in our encoding leaves us with the question on how we can provide a class with a reference to itself before it is even constructed? An imperative approach to solving this problem would be to use a mutable reference and leave it undefined to just after the object is constructed when it should be fixed with a proper reference to the object. All under the assumption that the self-reference is not touched during object construction as it would lead to undefined behavior.

We could explicitly write down this process, but fortunately it has already been captured by the *fixIO* combinator [22].

$$fixIO :: (a \to IO\ a) \to IO\ a$$

It takes a function that takes as its first argument expects a value of the type that it itself produces. We will use *fixIO* to provide objects with a self-reference. Because object instantiation now consists of two separate actions – closing record extension, and passing a self-reference – we capture the act of creating a new object instance inside a combinator:

$$new\ o = fixIO\ \$\ o\ emptyRecord$$

We implement the *printable_point* class and have it take a self-reference:

```
printable_point x_init = clazz $ λtail self → do
  varX ← newIORef x_init
  return PrintablePoint {
    getX                = readIORef varX
    , moveX             = λd → modifyIORef varX ((+) d)
    , print             = (self # getX) ≫= putStr ∘ show
    , _printablePointTail = tail
  }
```

Note that we essentially rely on laziness for this construction to work, *self* should only be accessed in safe positions (inside methods) as premature evaluation would cause the program to crash. We can test the self-reference by invoking *print*.

```
mySelfishOOP = do
  p ← new $ printable_point 3
  p # moveX $ 2
  p # print
```

In OO languages it is common practice to call initializer methods on an object inside the constructor. This can be mimicked by wrapping a class and carry out some initialization logic before returning the actual instance:

```
printable_point_constructor x_init tail self = do
  p ← printable_point x_init tail self
  p # moveX $ 2
  p # print
  return p
```

5.3.7 Single inheritance with override

"We illustrate inheritance by defining a class of colored points that inherits from the class of points. This class has all instance variables and all methods of class point, plus a new instance variable color, and a new method getColor.", section 3.7

OCaml
```
class colored_point x (color : string) =
  object
  inherit point x
  val color = color
  method getColor = color
end;;
```

Inheritance is a technique for sharing behavior between objects. It accomplishes sharing by incrementally extending classes – better known as subclassing. Often inheritance is confused with the orthogonal question of *substitutability*. Creating a new subclass is not necessarily the same thing as introducing a new subtype [14]. Although disparate issues our encoding does not permit the separation of the two, i.e. inheritance necessarily implies subtyping.

The correct implementation of inheritance in combination with self-reference is known to be tricky [13]. It is crucial that *self* is *late-bound*, i.e. when a class is extended *self* is bound at the latest possible moment such that a subclass may intercept method invocations on *self* in the superclass by overriding its behavior in the subclass. Late-binding is also referred to as *open recursion* conveying the intuition that the actual type of *self* is left *open* until the *recursion* is closed.

The *colored_point* class takes an initial values for *x*, *color*, and an extension of the record.

```
colored_point x color = clazz $ λtail self →
  printable_point x colored_point'
  where
```

```
    colored_point′ = do
      return ColoredPoint {
        _getColor          = return color
        , _coloredPointTail = tail
      }
```

Compared to our previous attempt at extending records in section 5.3.1 the object now has access to itself through *self*. Note that the self-reference should not be accessed in unsafe positions, i.e. positions where it is evaluated before the actual object is constructed and the self-reference is fixed. The combination of record extension 5.3.1 and self-reference 5.3.6 allows us to model a basic form of inheritance:

```
  myColoredOOP = do
    p ← new $ colored_point 3 "red"
    x ← p # getX
    c ← p # getColor
    print (x, c)
```

```
> myColoredOOP
(3, "red")
```

The above code shows that subclassing works, but we have yet to consider overriding methods. To show what is wrong with the approach to overriding methods shown in OOHaskell (section 2.4, p. 22) we adapt *colored_point* by overriding the *print* method.

```
  colored_point x color = clazz $ λtail self → do
    super ← printable_point x colored_point′ self
    return super {
      print = do putStr "so far - "; super # print
                 putStr "color - ";  putStr (show color)
    }
    where
    colored_point′ = do
      return ColoredPoint {
        _getColor          = return color
        , _coloredPointTail = tail
      }
```

Overriding *print* is done by updating the record that results from constructing a *ColoredPoint*. There are, however, a couple of questionable aspects about this approach. First, referring to the whole record as *super* is not appropriate as *super* should only refer to the parent object. Second, as a side-effect of letting *super* refer to the whole object we cannot refer to *super* in *colored_point′*. Unsatisfied with this approach to overriding methods provided by OOHaskell we continue to explore what it takes to properly model inheritance.

Deriving the inherit combinator

One of the key observations in implementing inheritance is that *super* refers to the fully constructed parent object. It exists as such in the scope of the subclass, allowing methods to refer to the *unmodified* parent object, after which it can be opened up to be extended with additional methods, possibly overriding methods of the *super* object.

For *colored_point* this means that we first instantiate its parent *printable_point* with the *emptyRecord* which results in a binding to *super* that part-takes in the construction of *colored_point'*. Then the *print* method is overridden and the *emptyRecord* is replaced with the *ColoredPoint* extension containing all additional methods and data.

```
colored_point x color = clazz $ λtail self → do
  super   ← printable_point x emptyRecord self
  wrapper ← colored_point' tail super self
  return super {
    print = do putStr "so far - "; super # print
               putStr "color - "  ; putStr color
    , _printablePointTail = wrapper
  }
  where
  colored_point' tail super self = do
    return ColoredPoint {
      _getColor        =
        do x ← super # getX
           putStrLn ("Retrieving color at: " ++ show x)
           return color
      , _coloredPointTail = tail
    }
```

Because we use normal records the process of overriding and extending becomes somewhat entangled.

Cook et al. [13] show how inheritance can be modeled using a combinator ▷ defined in the lambda calculus. They proof its correctness with respect to the operational semantics of a commonly used OO method-lookup algorithm.

$$\rhd : (Wrapper \times Generator) \rightarrow Generator$$

$$W \rhd G = \lambda self.(W(self)(G(self))) \oplus G(self)$$

The combinator takes a wrapper, generator, and builds a new generator by distributing the self-reference to both the generator and wrapper. It passes a generator applied to self as super to the wrapper, and combines the two using ⊕ forming a new generator. That the two instances of the generator applied to self are shared is left implicit, but is made clear by the visualization in figure 5.2.

It turns out that *colored_point* is a concrete instantiation of Cook's ▷ combinator obfuscated by technicalities caused by the use of records and IO monad. To make

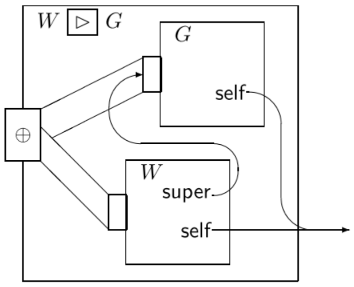


Figure 5.2: A visualization of the inheritance combinator (▷) taken from [13].

this correspondence clear we reify the combinator as the *extends* Haskell function, but first we define a couple of type synonyms that will make the type signature easier to digest:

```
type Class       tail self o     = tail → self → o
type EmptyClass                  = IO ()
type OpenClass  tail self o      = Class (IO tail)    self (IO o)
type SuperClass self sup         = Class EmptyClass self (IO sup)
type SubClass    tail sup self o = tail → super → self → IO sub
```

The *extends* combinator takes a few more parameters than ▷. Since we cannot define a generic operation for record concatenation (as OOHaskell does) we parameterize over it using ⊕. Furthermore, for syntactic purposes we have not combined *override* and ⊕ into a single operation.

```
extends ::
      SubClass    tail sup self sub -- w
  → SuperClass self sup             -- g
  → (sup  → self → IO sup′)         -- override
  → (sup′ → sub  → o)               -- combine subclass and superclass
  → OpenClass tail self o
extends w g override ⊕ = clazz $ λtail self → do
  super    ← g emptyRecord self
  wrapper ← w tail super self
  super′   ← override super self
  return $ super′ ‘ ⊕ ‘ wrapper
```

We can now express *colored_point* in terms of *extends*:

```
colored_point x color =
  (wrapper ‘extends‘ printable_point x) override (λo v → o {_printablePointTail = v})
  where
  override super self = return super {
    print = do putStr "so far - "; super # print
               putStr "color - "  ; putStr color
    }
    wrapper tail super self =
      return ColoredPointClass {
        _getColor = do x ← super # getX
                       putStrLn ("Retrieving color at: " ++ show x)
                       return color
        , _coloredPointTail = tail
      }
```

and demonstrate it through a simple example:

```
myOverridingOOP = do
  p ← new $ colored_point 3 "red"
  p # getColor
  p # print
```

```
>myOverridingOOP
Retrieving color at position: 3
so far - 3 color - "red"
```

Sometimes you might want to override existing methods without adding new ones. This form of anonymous overriding is also possible:

```
colored_point′ x_init color tail self = do
  super ← colored_point x_init color tail self
  return $ super {
    print = putStr "I'm a colored point"
  }
```

5.3.8 Class-polymorphic functionality

Because classes are just values we can parameterize computations over classes.

```
 -- Optional type
myFirstClassOOP ::
  Num a ⇒ (a → IO (PrintablePoint b → ())
              → PrintablePoint b
              → IO (PrintablePoint b))
        → IO ()
myFirstClassOOP point_class = do
  p ← new $ point_class 7
  p # moveX $ 35
  p # print
```

Any subclass of *PrintablePoint* may be passed into *myFirstClassOOP*.

```
ghci>myFirstClassOOP printable_point
42
```

```
ghci>myFirstClassOOP (flip colored_point' "red")
so far - 42 color - red
```

5.3.9 Orphan methods

Orphan methods are methods which can be shared between classes without relying on inheritance – a kind of horizontal reuse.

```
print_getX self = (self # getX) ↣ Prelude.print
```

The *print_getX* function can be applied to any subclass of *PrintablePoint*. In OOHaskell its type would be much more granular and hence work for any class that supports the *getX* method. We can get some of the structural behavior of OOHaskell by introducing a type class per method and overload the method on its object type.

```
class HasGetX o where
  callGetX :: o → IO Int
instance HasGetX (PrintablePoint t) where
  callGetX = getX
print_getX self = (self # callGetX) ↣ Prelude.print
```

On occasions this might be useful, but it requires significant boilerplate and quickly runs into ambiguity problems for methods with polymorphic arguments.

5.4 A type-perspective

5.4.1 Explicit casting

Up to this point we have avoided the issue of ascribing a value a different type based on its relationship in the subtyping hierarchy. Type ascription in the presence of subtyping is commonly known as casting. Given a value of type X, ascribing it a supertype is referred to as an upcast, whereas ascribing it a subtype is referred to as a downcast. The former allows X to be *viewed* as its supertype and can therefore be regarded as a form of abstraction or *elimination*. The latter can be viewed as a form of *introduction* and is arguably more involved since it needs to recover from potentially hidden information.

We will focus on single inheritance where each subtype has a single supertype:

$$\begin{tikzcd}
& \cdots & & \text{supertype} \\
& X \arrow[u] \arrow[dl, dashed] \arrow[d, dashed] \arrow[dr, dashed] & & \\
\cdots & \cdots & \cdots & \text{subtype}
\end{tikzcd}$$

To illustrate why we need casting we show a typical OO scenario where two objects of different types, related by subtyping, are placed inside a list containing only elements of their supertype.

```
let rect   = ... :: IShape (IRectangle ())
    circle = ... :: IShape (ICircle ())
```

Suppose that we insert both shapes into a list:

```
[rect, circle] -- Type error
```

The result is a type error, because the two type element types do not unify. What we actually want is that the type system infers the more general type *IShape a*, but this would imply that it has some notion of subtyping and allow universal quantification over monomorphic values at the covariant position, neither of which are the case.

There are two options to make it work either change the type of list elements, or change the type of the elements inserted into the list. The former leads to more idiomatic Haskell and has two basic options: use *Either*, or an existential envelope.

```
-- Either
[L rect, R circle] :: [Either (IShape (IRectangle ())) (IShape (ICircle ()))]
```

Using *Either* requires tagging *rect* and *circ* with respectively *L* and *R*. It is the simplest solution and leaves the original types intact. That the original types are kept intact is at the same time one of the problems. We can construct the list, but we cannot pass it to a function that expects a list of any shape. Also, when generalizing to n distinct types operations like injection, projection, and mapping become more involved.

```
-- Existential envelope
[rect, cirlce] :: [∃a.IShape a]
```

Using existentials we can take advantage of the tail-polymorphic structure and hide the tail of a record by existentially quantifying over it. Now we can simply insert any subtype of *IShape* inside a list without further ado. However, plain existentials cannot not recover from the existentially quantified information. In section 5.4.2 we will see how in combination with explicit casting we can mitigate the loss of information.

Neither approach is satisfactory. OOHaskell uses yet another approach to insert elements into the list where it circumvents unification problems by explicitely changing the type of list elements before they are inserted. Their type is modified by using a narrowing function that *shops off* the tail.

The narrowing function for shapes is defined as follows:

```
-- Specific narrowing function
up_shape :: IShape a → IShape ()
up_shape o = o {_shapeTail = ()}
```

It takes *at least* a shape, opens up the object, and throws out the tail replacing it with the *emptyRecord'*. We can use *up_shape* as a helper to insert *rect* and *circle* into a list of shapes.

```
[up_shape rect, up_shape circle] :: [IShape ()]
```

Even though it provides a working solution it appears to be even less useful compared to existentials. It accomplishes the same thing with more boilerplate for the necessarily type specific record manipulation, a run-time overhead for performing the actual record manipulation, and requires knowledge of the whereabouts of the narrowing functions. Furthermore, similar to existentials it does not admit downcasting because it simply throws out information that may later be required to perform a downcast.

In the following sections we will improve upon the above techniques by implementing generic functions for up and downcasting. We use type classes for automatically generating the type conversion functions and dynamic typing for hiding the tail instead of deposing it. The use of dynamic types allows a downcast to reconstruct the original types.

Generic upcast

An upcast is a function that when given a source (subtype) and target (supertype) type provides unique directed-path through the subtyping hierarchy from source to target type composed of the smallest possible narrowing steps that allow it to move along the edges of the path.

Although *up_shape* conceptually corresponds to such a path its implementation does not. In order to implement a generic upcast function we need a more compositional approach that is explicit about the constituents of the path.

Suppose we want to upcast a *Cube* to a *Shape* obtaining a more explicit version than:

```
from_cube_to_shape = up_shape
```

requires that we follow the edges in figure 5.3 from *Cube* to *Shape*. The labels on the edges are functions that allow us to transition from one vertex to the other.

```
up_rectangle :: IShape (IRectangle ()) → IShape ()
up_rectangle o = o {_shapeTail = ()}
up_circle :: IShape (ICircle ()) → IShape ()
up_circle o = o {_shapeTail = ()}
up_cube :: IShape (IRectangle (ICube ())) → IShape (IRectangle ())
up_cube o = o {_shapeTail = _shapeTail o {_rectangleTail = ()}}
```

The explicit version of *from_cube_to_shape* mentioning all labels in the path is given by:

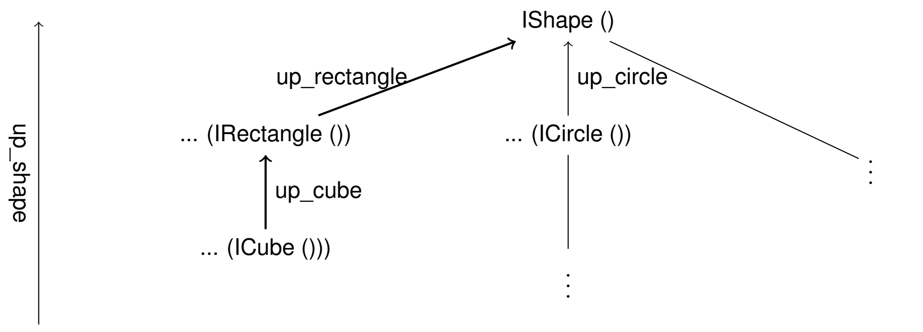


Figure 5.3: The shapes subtyping hierarchy where the edges are annotated with narrowing functions.

$$\mathit{from\_cube\_to\_shape} = \mathit{up\_rectangle} \circ \mathit{up\_cube}$$

Where *up_shape* allowed us to take a shortcut the explicit version does not. It requires N record manipulations, where

$$N = \frac{1}{2}(n_s - n_t)(n_t + n_s - 1)$$

and n_s, n_t are respectively the depth of the source and target type with $n \geqslant 1$. Thus instead of a single record manipulation with *up_shape* it requires 3 record manipulations. Clearly, the explicit approach is less efficient but the effect is somewhat mitigated by the fact that n is bounded by the subtyping depth which in practice is quite limited.

Manually writing functions like *from_cube_to_shape* is tedious, not generic, and hence offers no benefits over just using *up_shape*. The possibility to explicitly specify the path only turns into a benefit when we can craft a function that will automatically create the path for us provided with a source and target type. This is exactly what we aim for. A generic upcast function should therefore: given a source and target type automatically create the path between source and target by decomposing it into individual narrowing steps.

Because the behavior of an upcast depends on both the source and target type we model it using a multi-parameter type class[3]:

```
class α ≺ β where
  upcast :: α → β
```

The type class reads: if α is a *subtype* of β, there exists an *upcast* operation which can be used to cast a value of type α into value of type β.

In order for this proposition to hold we rely on the essential property that each subclass can be interpreted as a new subtype, and thus walking over the type structure implies walking over the subtyping hierarchy.

The rules that decide whether two types are related by subtyping are shown in Figure 5.4. It is well-known that these type rules do not directly translate into an

[3]We use *TypeOperators* solely for stylistic purposes.

$$[\textit{reflexivity}]\ \frac{}{A <: A}$$

$$[\textit{transitivity}]\ \frac{A <: B \quad B <: C}{A <: C}$$

Figure 5.4: Typing rules for the subtyping relation.

algorithmic implementation because it is not clear when they should be applied. In other words, they are not *syntax directed* (see Chapter 16 [52]).

It turns out that if we limit the inhabitants of the $\prec$ type class to first-order values with a tail-polymorphic structure we can obtain a syntax directed version of the typing rules. Instance declarations then correspond to the typing rules which through context reduction provide the compiler with evidence on whether two types are related by subtyping.

We use the 'shapes' example in a first attempt at translating the typing rules to instance declarations on a *per type* basis, i.e. introducing new instance declaration for each new type. The reflexivity rule is trivially implemented by the identity function, because upcasting a value to itself has no effect.

```
-- Reflexivity
instance IShape ()               ≺ IShape ()               where
  upcast = id
instance IShape (IRectangle ()) ≺ IShape (IRectangle ()) where
  upcast = id
instance IShape (ICircle ())     ≺ IShape (ICircle ())     where
  upcast = id
```

Note that we need to lift a Haskell 98 restriction which requires the shape of an instance head to be of the form $C\ (T\ a_1\ ...\ a_n)$, where C is designates the class, T a data type constructor, and $a_1, ..., a_n$ a set of distinct type variables. We lift it to arbitrary nested types by enabling the *FlexibleInstances* extension.

Next, we implement transitivity for *Rectangle* and *Circle*.

```
-- Transitivity
instance (IShape () ≺ β) ⇒ IShape (IRectangle ()) ≺ β where
  upcast = upcast ∘ up_rectangle
instance (IShape () ≺ β) ⇒ IShape (ICircle ())     ≺ β where
  upcast = upcast ∘ up_circle
```

The transitivity instances embed a single narrowing step and delegate further work to other instance declarations. We can now perform a generic upcast without knowing about the specifics of narrowing functions. The solution also allows for easy extension to new object types by simply adding another instance for reflexivity and transitivity.

To illustrate what happens at compile-time we let the system automatically derive *from_cube_to_shape* for us:

```
let cube = ... :: IShape (IRectangle (ICube ()))
in upcast cube :: IShape ()
```

The compiler builds up a proof tree for each application of *upcast* proving that the source and target type are related by subtyping thereby also ruling out silly casts (i.e. casts between two unrelated types). Here is the proof tree derived by the context reduction machinery for the above example:

$$\dfrac{\dfrac{\dfrac{}{\mathit{IShape}\ () \prec \mathit{IShape}\ ()}\,[\text{refl-shape}] \quad \cdots}{\mathit{IShape}\ (\mathit{IRectangle}\ ()) \prec \mathit{IShape}\ ()}\,[\text{trans-rect}] \quad \cdots}{\mathit{upcast}\ \mathit{cube} :: \mathit{IShape}\ ()}\,[\text{trans-cube}]$$

The value produced by the proof should look familiar. It corresponds to the *from_cube_to_shape* function except that at the end it includes the identify function as a residue of the recursion.

```
  id ∘ up_rectangle ∘ up_cube
≡ {- left-identity -}
  up_rectangle ∘ up_cube
≡ {- by definition -}
  from_cube_to_shape
```

There are quite some subtleties involved in the transitivity case. First, we need two additional language extensions *FlexibleContexts* and *OverlappingInstances*. The former does the same for the context as *FlexibleInstances* does for the head. The latter is necessary because β makes the transitive and reflexive case overlap. Fortunately, this is a harmless case of overlapping instances as it does not lead to any difficulties in determining the most specific instance which can still be determined by solely looking at the instance head. Leaving β polymorph allows the recursive call to *upcast* to pick either another transitive instance or bottom out the recursion at the reflexive instance. This is necessarily the case because the choice on which instance to pick is determined by the instantiation of β at the call site.

Pushing it a bit further

While the *instance per new subclass/subtype* approach works its dependence on concrete types requires an unnecessary amount of instances for classes with essentially the same type *structure* (see *Circle* and *Rectangle*). A much better approach would be to abstract over the concrete types. As a consequence the problem of providing instance declarations shifts from *per new subclass* to *per increase in the depth of the subtyping hierarchy*. Fortunately, this is not that much of a problem because the depth of a subtyping hierarchy is in practice often quite limited.

We refactor the previous instance declarations, replacing the concrete types with type variables, but leaving the structure intact.

```
 -- depth 1: reflexivity
instance c () ≺ c () where
  upcast = id
```

For each subsequent increase in depth we add an instance for both reflexivity and transitivity.

```
-- depth 2
instance a (b ()) ≺ a (b ()) where
  upcast = id
instance (a () ≺ x) ⇒ a (b ()) ≺ x where
  upcast = upcast ∘ (? :: a (b ()) → a ())
-- depth 3
instance a (b (c ())) ≺ a (b (c ())) where
  upcast = id
instance (a (b ()) ≺ x) ⇒ a (b (c ())) ≺ x where
  upcast = upcast ∘ (? :: a (b (c ())) → a (b ()))
```

By abstracting over the concrete types we gained some expressive power, but lost some information. We no longer know which narrowing functions should be called at the position of the question mark. In order to capture the specific narrowing functions we introduce a new type class *Narrow*. It allows us to defer the decision on what concrete narrowing function to use to the call site.

The *Narrow* type class captures the set of specific narrowing functions.

```
class Narrow α β where
  narrow :: α → β
```

We augment the transitivity instances with an additional *Narrow* constraint, and replace the question marks with calls to the *narrow* function.

```
-- Depth 2: add a Narrow constraint
instance (a () ≺ x, Narrow (a (b ())) (a ())) ⇒ a (b ()) ≺ x where
  upcast = upcast ∘ (narrow :: a (b ()) → a ())
-- From rectangle to shape
instance Narrow (IShape (IRectangle ())) (IShape ()) where
  narrow = up_rectangle
```

It is important that the type variables in the head of the transitivity instance match those annotating the *narrow* function. We use the *ScopedTypeVariables* extension to bring the type variables in the head into scope such that they may be reused in the body of *upcast*. Furthermore, by adding the *Narrow* constraint the Paterson Conditions no longer hold, i.e. the context is no longer smaller than the instance head[4]. To lift this restriction we must enable *UndecidableInstances* with which termination of context reduction process is no longer guaranteed. Fortunately, this causes no problems because we can tell by our instance definitions that there is no possibility to send the context reduction into an infinite loop.

The subtype type class has some interesting behavior, different from what one might intuitively expect from subtyping:

```
foo c =
  let c′ = upcast c :: A (B (C ()))
```

[4]For more details see: `http://www.haskell.org/ghc/docs/7.2.2/html/users_guide/type-class-extensions.html#instance-rules`

```
    b  = upcast c :: A (B ())
    a  = upcast c :: A ()
  in ()
```

On basis of subtyping one might expect the type of *foo* to correspond to:

$foo :: (a \prec A\ (B\ (C\ ()))) \Rightarrow a \rightarrow ()$

since both b and a are included in $A\ (B\ (C\ ()))$. However, the actual type of *foo* corresponds to:

$foo :: (a \prec A\ (), a \prec A\ (B\ ()), a \prec A\ (B\ (C\ ()))) \Rightarrow a \rightarrow ()$

Each call to *upcast* contributes a constraint to the context. However, the context reduction machinery is not aware of the subtyping rules and therefore cannot reduce the constraint set to $a \prec A\ (B\ (C\ ()))$. This is not erroneous nor does it affect the behavior of *foo*, but for a clean type signature we must perform the context reduction in our head and provide *foo* with an explicit type signature.

Generic downcast

A downcast is the partial inverse of upcast. Its partial because it *attempts* to restore type information whilst admitting the possibility of failure in case the expected type does not correspond to the actual type.

The problem with the current implementation of *upcast* is that the individual narrowing steps throw away the tail making it impossible to later restore it with a downcast. In order to facilitate downcasts the tail needs to be somehow maintained without having it surface in the type. A known technique for hiding types is existential quantification. However, using plain existentials we cannot recover from the hidden information. The *Data.Dynamic* library does allow hidden types to be restored by testing on type equality between the run-time type representations of the hidden value and its expected type. If the value's type matches the expected type it provides proof that the hidden value may safely be recovered from as the expected type.

Before implementing downcast we fix the tail representation by wrapping it inside a new data type *Record* such that an upcast can hide the tail.

```
-- A wrapper for the tail
type Record α = Either α Dynamic
```

Record is represented using the binary sum type *Either* a b. With the tail wrapped inside a *Record* α value we gain the possibility of hiding the tail. We make no distinction between the type of an empty record and that of a hidden tail.

```
-- Smart constructors
record = L
unRecord (L a) = a
hideRecord :: Typeable α ⇒ Record α → Record ()
hideRecord (L a) = R (toDyn a)
```

```
emptyRecord :: IO (Record ())
emptyRecord = return (record ())
```

The *record* and *unRecord* functions wrap and unwrap a record, *hideRecord* hides a record by injecting it into a *Dynamic*, and *emptyRecord* is adapted to the new record representation. Note that the representation requires every interface to be an instance of *Typeable* such that it can be used by the *Data.Dynamic* library.

We now wrap every tail inside a *Record* like this

```
data IShape α = IShape {
  ...
  , _shapeTail :: Record α
}
```

, and revise the narrowing functions to use *hideRecord*:

```
instance Narrow (IShape (IRectangle ())) (IShape ()) where
  narrow o = o {_shapeTail = hideRecord (_shapeTail o)}
```

The effect of the new representation is that for example an upcast from *Cube* to *Shape* still has the same type, but internally looks more like this:

IShape (IRectangle (ICube ()))

where black is the visible and gray the invisible part of the type. By changing the representation we introduced the possibility of restoring the subtypes.

With the correct data structure in place we are ready to implement the downcast function. Since we do not know what types are possibly hidden – it can be any subtypes – a downcast is necessarily a partial function. The best we can do is *try* to perform a downcast. A downcast is a function from a value of type β to a value of type *Maybe* α which succeeds only if β is a supertype of α, and β can be converted into a value of type α.

```
downcast :: β → Maybe α
```

At first, it seems to make sense to place *downcast* inside the $\prec$ type class given that *downcast* is the partial inverse of *upcast* and the instances are governed by the same subtyping rules.

```
class α ≺ β where
  upcast   :: α → β
  downcast :: β → Maybe α
```

This works fine for the reflexivity instances, but unfortunately breaks for transitivity as the type class constraints necessary for *downcast* are not incompatible with those required for *upcast*. An *upcast* forgets information while *downcast* tries to gain information. Hence, we introduce a separate type class for the supertype relation and rely on the library designer to ensure consistency between the two.

```
-- New type class for downcasting
class α ≻ β where
  downcast :: α → Maybe β
```

For reflexivity a downcast always succeeds.

```
-- Depth 1
instance a () ≻ a () where
  downcast = Just
instance a (b ()) ≻ a (b ()) where
  downcast = Just
```

The interesting case is transitivity.

```
instance (a (b ()) ≻ a (b c)) ⇒ a () ≻ a (b c) where
  downcast o = case ? :: Maybe (a (b ())) of
    Just r   → downcast r
    Nothing → Nothing
```

The instance declaration should be read as follows: given a value of type $a\ ()$ we may downcast it to a subtype $a\ (b\ c)$ provided that we can downcast it to $a\ (b\ ())$, if the downcast is successful we delegate the task of resolving c to another instance, otherwise we fail by returning *Nothing*.

Similar to *upcast* we have lost information by abstracting over the actual types. We capture the dual of *Narrow* through a new type class:

```
class Widen β α where
  widen :: β → α
```

and replace the question mark with a call to *widen*:

```
instance (a (b ()) ≻ a (b c), Widen (a ()) (a (b ()))) ⇒ a () ≻ a (b c) where
  downcast o = case widen o :: Maybe (a (b ())) of
    Just r   → downcast r
    Nothing → Nothing
```

By pattern matching on a known part of the type structure and an unknown part c a downcast can incrementally recover from the hidden types. The implementation of *widen* only works correctly if different parts of the library maintain the following invariants:

1. An *upcast* always hides the tail of a record by using *hideRecord*.
2. All interfaces should derive from *Typeable*.
3. Fresh object instantiations always have *emptyRecord′* as tail.
4. A downcast can only encounter a *R* data constructor.

Invariant (1) is not enforced by the type system. Without precautions it is possible for *upcast* to choose *emptyRecord′* which would destroy the invariant. Fortunately, as library designer we may export *Record* as an abstract data type and confine *emptyRecord* to internal use.

Invariant (2) is taken care of by the macros (see section 5.5). However, if a interface does not derive from *Typeable* and (1) holds, using casting operations on that class will fail at compile-time.

Invariant (3) is covered by the *new* combinator.

Invariant (4) follows from (1) and (3). A downcast cannot encounter *L* because widening only starts at a concrete object type. Furthermore, it cannot encounter the empty record because any downcast always bottoms out at a reflexivity instance which itself does not inspect the representation.

All instances of *Widen* are defined in terms of *genericWiden*:

```
instance Widen (IShape ()) (IShape (IRectangle ())) where
  widen o = genericWiden o _shapeTail (λo v → o {_shapeTail = v})
```

The *genericWiden* function is passed the object it should perform widening on and a getter/setter for the object's tail. It discriminates on the object's tail and attempts to recover its tail.

```
genericWiden :: ∀o a b c.Typeable b ⇒
     o
  → (o → Record a)
  → (o → Record b → c)
  → Maybe c
genericWiden o getTail setTail =
  case getTail o of
    R d → maybe Nothing (Just ∘ setTail o ∘ record) (fromDynamic d :: Maybe b)
    L _ → error "invariant (4)"
```

A change in semantics

In section 5.3.1 we informally described the semantics of tail-polymorphic records. The meaning of the tail-polymorphic type structure now slightly changes because we modified the object representation such that the tail of a record may be hidden using a dynamic type. Whereas an object of type $A\ ()$ used to mean *exactly* $A\ ()$ it can now mean *at least* $A\ ()$ even though its type has not changed. Given that now both $A\ ()$ and $A\ a$ can be interpreted as *at least* $A\ ()$ we wonder: can we substitute one for the other? We cover this question of substitutability for the contra- and covariant position.

```
  A () → ...
-- subst
  A a → ...
```

We may substitute $A\ ()$ with $A\ a$ in the contravariant position by changing the type signature. All calls to the function can remain unchanged. However, from a implementation perspective this substitution may prove problematic since we cannot use casting on polymorphic objects. Also, the substitution transitively inhibits other functions from using casts. A better alternative would be to substitude $A\ ()$ with a type variable constrained by the subtype type class:

```
  A () → ...
-- subst
  a ≺ A () ⇒ a → ...
```

Now the function can get any monomorphic subtype of $A\ ()$ as argument, i.e. excluding $A\ a$ or any polymorphic subtype (e.g. $A\ (B\ a)$). Furthermore, before the argument can be used it must first be casted to $A\ ()$.

We may substitute $A\ a$ for $A\ ()$ in the contravariant position by again simply changing the type signature:

```
  A a → ...
-- subst
  A () → ...
```

This change will require all call sites to upcast the function argument to $A\ ()$ unless the argument is already of type $A\ ()$. The function implementation will remain unaffected by the change.

We now consider substitutability in the covariant position. Substituting $A\ ()$ with $A\ a$ in the covariant position is not possible because this would require universal quantification over a value with a monomorphic type. However, what we can do is abstract over its tail using existential quantification:

```
  ... → A ()
-- subst
  ... → ∃a.A a
```

The resulting existential type can then be freely applied to functions expecting an universally quantified value of type $A\ a$. Thus the existential gives us the same behavior we would otherwise get from returning a value of type $A\ a$.

If a function has $A\ a$ in the covariant position it can only have gotten it as an argument. It can never by itself produce a value of type $A\ a$. If a function has $A\ a$ in the covariant position we can substitute it with $A\ ()$ provided that we modify the calling and return context to cast the argument and return value to the appropriate type.

We conclude that moving between $A\ a$ and $A\ ()$ is a delicate business. It does not always go without affecting the implementation, and in particular substituting $A\ ()$ with $A\ a$ may break functions that depend on casting. Using polymorphic objects transitively inhibits other functions from using casts, and when used as argument to a method requires rank-2 polymorphism. The only benefit they bring is that no type conversions are necessary. Hence from a user's perspective it makes sense to stick with monomorphic objects at the cost of some more explicit type conversions.

Combinators

Using the subtype type class we can define two asymmetric combinators *consUb* and *nilUb* that allow the construction of a homogeneous list out of object values related by subtyping. Upon inserting a value it is first cast to the supertype that is provided at the call site and then "consed" on to the list. The supertype should be the common upper bound of all elements in the list.

```
consUb :: ∀a b.(a ≺ b, Typeable a) ⇒ a → [b] → [b]
consUb o xs = (upcast o :: b) : xs
nilUp = [ ]
```

We have used the above combinators in section 5.2.2 to insert different kind of shapes into a homogeneous list of shapes:

```
let scribble :: [IShape ()]
    scribble = consUb s1 (consUb s2 nilUb)
```

The dual, for the *lower bound*, is implemented similarly:

```
consLb :: ∀a b.(b ≻ a, Typeable b) ⇒ b → [a] → [a]
consLb o xs =
  case downcast o :: Maybe a of
    Just x    → x : xs
    Nothing → xs
nilLb = [ ]
```

The definition of *consUb*/*nilUb* and *consLb*/*nilLb* only work for lists. We generalize their definition by overloading them on the container type such that they may be used for any container type that allows incremental construction.[5]

```
class Applicative f ⇒ CastCons f where
  consUb :: ∀a b.(a ≺ b, Typeable b, Monoid (f b)) ⇒ a → f b → f b
  consUb o xs = pure (upcast o :: b) ⊕ xs
  consLb :: ∀b a.(b ≻ a, Typeable b, Monoid (f a)) ⇒ b → f a → f a
  consLb o xs = maybe xs (xs ⊕ ∘pure) (downcast o :: Maybe a)
  nilUb, nilLb :: Monoid (f a) ⇒ f a
  nilUb = ∅
  nilLb = ∅
```

Similarly, we can lift casting operations to work on container types:

```
class Functor f ⇒ Castable f where
  fup :: a ≺ b ⇒ f a → f b
  fup = fmap upcast
  fdown :: ∀a b.(Foldable f, Applicative f, Monoid (f a), b ≻ a) ⇒ f b → f a
  fdown = foldr (⊕ ∘ maybe ∅ pure ∘ (downcast :: b → Maybe a)) ∅
```

[5] ∅: mempty, ⊕: mappend

Not all types that allow mapping also allow deconstruction, hence the *Applicative* constraint is pushed down as a function constraint. We use *fdown* to create a list of rectangles out of a list of shapes:

```
let scribble' :: [IShape (IRectangle ())]
    scribble' = fdown scribble
```

We can also define the familiar *instanceof* operations in terms of *downcast*. It test if a value is of a particular type:

```
instanceof :: ∀a b.(b ≻ a) ⇒ b → a → Bool
instanceof b _ = isJust (downcast b :: Maybe a)
```

Using *instanceof* we can define a function *selectiveDraw* that accepts any Shape, but only draws rectangles.

```
selectiveDraw :: IShape () → IO ()
selectiveDraw shape =
  when (shape `instanceof` (⊥ :: IShape (IRectangle ())))
       (shape # draw)
```

Note that *instanceof* only type checks if it is used for testing if an object actually is some subtype of its current type. There is no need for testing if it is an instance of some supertype because this is intrinsically known.

5.4.2 Self-returning methods

A self-returning method is a method whose return type is the type of self or some other type based on self. It is known that encapsulation by *procedural data abstraction* requires recursive types for the precise typing of self-returning methods in the presence of inheritance (section 3.1 [14]). On the left in figure 5.5 the problem is made explicit in Java (which lacks recursive types). To let the program type check some otherwise superfluous casts must be inserted. On the right there is a hack which uses Java Generics [48] and a special *getThis* function to reify the lost type information [6].

In a first attempt to implement a self-returning method we reuse the shapes example and augment *IShape* with a method *meShape* that returns itself.

```
data IShape α = IShape {
      ...
  meShape    :: IO (IShape α)
  , _shapeTail :: Record α
}
```

In the return type of *meShape* we simply refer to itself. Unfortunately, because α is now used at two positions in interface the *_shapeTail* function becomes less

[6] http://www.angelikalanger.com/GenericsFAQ/FAQSections/ProgrammingIdioms.html#FAQ205

```java
class A {
  public A foo() {
    return this;
  }
}

class B extends A {
  public B bar() {
    return this;
  }
}

B b = new B();
// type check error
b.bar().foo().bar();
// fine
((B) b.bar().foo()).bar();
```

```java
abstract class A<T extends A<T>> {
  public T foo() {
    return (T) getThis();
  }
  public abstract T getThis();
}

class B extends A<B> {
  public B bar() {
    return this;
  }
  public B getThis() {
    return this;
  }
}

B b = new B();
// fine
b.bar().foo().bar();
```

Figure 5.5: On the left: an example in Java where we use self-returning methods in combination with inheritance. On the right: a trick to resolve the need for casting.

general compared to what it would have been had we only used α in a single position.

-- *Without meShape*
$\mathit{\_shapeTail} :: \mathit{IShape}\ \alpha \rightarrow \mathit{Record}\ \beta \rightarrow \mathit{IShape}\ \beta$
-- *With meShape*
$\mathit{\_shapeTail} :: \mathit{IShape}\ \alpha \rightarrow \mathit{Record}\ \alpha \rightarrow \mathit{IShape}\ \alpha$

As a consequence we can no longer change the type of the tail in isolation making record extension highly impractical and impossible to fit into our framework for casting. Alternatively we could try to abstract over the return type by parameterizing over it, similar to what is done on the right in figure 5.5. However, this will also not work because the type parameter now has to unify with itself which is prohibited by the occurs check. Because Haskell does have iso-recursive types OOHaskell uses newtype wrappers to solve this problem. Unfortunately, wrapping self inside a newtype will not work for our encoding as it again requires using the tail type parameter at multiple positions preventing record extension.

Confronted with the impossibilities of the encoding we resort to a less sophisticated version of self-returning methods and take for granted that some casts are required. We start by making the return type of *meShape* concrete:

data $\mathit{IShape}\ \alpha = \mathit{IShape}\ \{$
  ...
  $\mathit{meShape} :: \mathit{IO}\ (\mathit{IShape}\ ())$

```
        ...
    }
    shape newx newy concreteDraw = clazz $ λtail self →
      ...
      return IShape {
        ...
        meShape = return self
        ...
      }
```

The program type checks, but there is a subtle problem. The type inferencer has instantiated *self* to *IShape* () which again inhibits further extension. We should somehow convince the type inferencer that it may temporarily assume *self* to be of type *IShape* () without this knowledge overspecializing the inferred function type. There are two approaches that may help us achieve our goal: upcasting or existential quantification.

```
    -- Upcast
    meShape = return (upcast self :: IShape ())
```

By using *upcast* we make our knowledge that *self* is of at least *IShape* () explicit changing the inferred type to:

```
    -- Inferred type
    shape
      :: IShape a₁ ≺ IShape () ⇒
        Int
         → Int
         → (IShape a₁ → IO ())
         → IO (IShape a₁ → Record a)
         → IShape a₁
         → IO (IShape a)
```

Self is now overloaded by what at first sight may seem like a problematic constraint. However, because objects are always instantiated by using *new* we know that a_1 will eventually be instantiated to a concrete type such that substituting it in *IShape* $a_1 \prec$ *IShape* () will satisfy the constraint.

As an example we show how we can call *meShape* on a *Rectangle*. Calling *meShape* on a *Rectangle* returns a *Shape* which we downcast back to a *Rectangle* and use to invoke *getWidth*:

```
    -- ((Rectangle) s1.meShape()).getWidth()
    mySelfReturn = do
      s₁      ← new $ rectangle 10 20 5 6
      shape ← s₁ # meShape
      let Just rect = downcast shape :: Maybe (IShape (IRectangle ()))
      w ← rect # getWidth
      putStrLn $ show w
```

A different approach would be to existentially quantify over the tail of the record.

```
-- Existential quantification
data IShape α = IShape {
    ...
    meShape :: ∃α.IO (IShape α)
    ...
}
```

This makes it easy to return *self*.

```
meShape = return self
```

But because the tail is existentially quantified over it does not allow casting. We solve this by placing a subtype constraint on the quantified variable.

```
meShape :: ∃α.(IShape α ≺ IShape ()) ⇒ IO (IShape α)
```

Herewith the concrete type can be reified through an upcast.

```
mySelfReturn = do
    ...
    let shape'    = upcast shape :: IShape ()
    let Just rect = downcast shape' :: Maybe (IShape (IRectangle ()))
    ...
```

We gained ease of expression at the return site, but at the same time made the task of the caller more verbose. Also, the use of anonymous existentials is unique to UHC and requires newtype wrappers in GHC which makes it a far less attractive option. For these reasons we prefer to use concrete object types, even though the combination of top-level class definitions, overloading, and parameter hiding can trigger the monomorphism restriction[7].

5.4.3 Parameterized classes

In section 5.3.4 we showed how we could create polymorphic classes by adding type parameters to the interface definition. Parameterization over method types allows for much greater flexibility because the same interface can be reused for instantiations with different concrete types. This is especially useful for container-like classes where operations on the container are independent of the actual contents. In this section we will further explore parameterized classes in combination with inheritance and casting. But before we do we first generalize our previous definition of an *interface*.

Definition 4. *An interface is a record C with the following shape:*

data $C\ a_1, ..., a_n, t = C\ \{m, \_cTail :: Record\ t\}$

[7] `http://www.haskell.org/onlinereport/decls.html#sect4.5.5`

```
class Pair<A,B> {
   private final A a;
   private final B b;

   public Pair(A a, B b) {
      this.a = a;
      this.b = b;
   }

   public A getFirst() {
      return a;
   }

   public B getSecond() {
      return b;
   }
}
```

```
class Triple<A,B,C> extends Pair<A,B> {
   private final C c;

   public Triple(A a, B b, C c) {
      super(a,b);
      this.c = c;
   }

   public C getThird() {
      return c;
   }

   public Triple<B,A,C> swap() {
      return
        new Triple<B,A,C>(
          getSecond(),getFirst(),c
        );
   }
}
```

Figure 5.6: Two generic container types reminiscent of the Haskell tuple.

where C takes $n \geqslant 1$ type parameters, t represents the tail, and may be instantiated to either $()$ or an interface. In the body there is m which expands to zero or more methods that may use any of $a_1, ..., a_n$, and a special method $\_cTail$ where c is the uncapitalized version of C.

Many statically typed OO languages have the ability to parameterize classes. We show that our library can easily deal with parameterized classes that are invariant in their type parameters. As a reference we have implemented a class *Pair* in Java using generics, reminiscent of the Haskell tuple, and let another class *Triple* extend from *Pair* (see figure 5.6). Both *Pair* and *Triple* are parameterized over their contained types.

We perform a stepwise transcription of the Java code to Haskell. First, the *Pair* interface.

```
data IPair a b t = IPair {
  _getFirst    :: IO a
  , _getSecond :: IO b
  , _pairTail  :: Record t
}
```

In Java all classes are subclasses from *Object*. Staying true to the example we also extend from *Object* which we take to be a simple placeholder.

```
pair a b =
  (pair′ ‘extends‘ object) noOverride set_Object_Tail
  where
```

```
pair′ tail super self =
  return IPair {
    _getFirst   = return a
    , _getSecond = return b
    , _pairTail  = tail
  }
```

For casting to work we provide the necessary instances for *Narrow* and *Widen*:

```
type Pair_ a b t = Object_ (IPair a b t)
type Pair   a b  = Pair_ a b ()
instance (Typeable a, Typeable b) ⇒ Narrow (Pair a b) Object where
  narrow = modify_Object_Tail hideRecord
instance (Typeable a, Typeable b) ⇒ Widen Object (Pair a b) where
  widen o = genericWiden o get_Object_Tail set_Object_Tail
```

Notice that we take fruitful use of the fact that the tail is always the last type parameter. If this were not the case we would have been be forced to write down all instances of the sub- and super type classes, for all interface shapes, which would lead to a combinatorial explosion in the number of instances.

We proceed by transcribing the *Triple* class. Interestingly, the *Triple_* type synonym and the Java class declaration look very much alike. In Java the type variables are introduced implicitly by usage whereas in Haskell they need to be explicitly declared before they can be used. In both cases the programmer is responsible for the correctly distributing the type variables.

```
type Triple   a b c   = Triple_ a b c ()
type Triple_ a b c t = Pair_ a b (ITriple a b c t)
data ITriple a b c t = ITriple {
  _getThird   :: IO c
  , _swap      :: IO (Triple b a c)
  , _tripleTail :: Record t
}
```

The implementation follows naturally from the interface definition. Unfortunately, the type inferencer does not infer the correct type for *triple*. It infers that both *a* and *b* should be of the same type because they are used interchangeably at different points in the program (see *_swap* and *pair*). We have to explicitely mark them as distinct by providing a type signature.

```
triple ::
      a
   → b
   → c
   → OpenClass (Record tail) self (Pair_ a b (ITriplet a b c tail))
triple a b c =
  (triple′ ‘extends‘ pair a b) noOverride set_Pair_Tail
  where
```

```
    triple′ tail super self =
      return ITriple {
        _getThird   = return c
        , _swap      = new $ triple b a c
        , _tripleTail = tail
      }
  swapTriple = _swap ∘ unRecord ∘ get_Pair_Tail
```

We also require two additional instances for *Narrow* and *Widen*:

```
  instance (Typeable a, Typeable b, Typeable c) ⇒ Narrow (Triple a b c) (Pair a b) where
    narrow = modify_Pair_Tail hideRecord
  instance (Typeable a, Typeable b, Typeable c) ⇒ Widen (Pair a b) (Triple a b c) where
    widen o = genericWiden o get_Pair_Tail set_Pair_Tail
```

We put the two classes to use by constructing a *Pair* and *Triple*, insert them into a list of pairs, and map over the list projecting out the first component and printing its value.

```
  myOOTriplet = do
    p ← new $ pair  (0 :: Int) (3.0 :: Double)
    t ← new $ triple (0 :: Int) (4.0 :: Double) "Hi"
    let pairs :: [Pair Int Double]
        pairs = consUb t (consUb p nilUb)
    sequence_ $ map (λp → p # getFirst ↣ print) pairs
    t′ ← t # swapTriple
    t′ # getFirst ↣ print
```

```
ghci>myOOTriplet
0
0
4.0
```

With parameterized classes the question of substitutability can be extended to incorporate a class' type parameters. For instance, *Pair Point Point* where *Point* is a subclass of *Object* is intuitively a subtype of *Pair Object Object*, i.e. it is safe to substitute a value of type *Pair Object Object* with a value of type *Pair Point Point* because both *getFirst* and *getSecond* are expected to return a *Point* which can safely be interpreted as an *Object*. This intuition is formalized by *depth subtyping*. Unfortunately, our library is limited to a coarse form of *width subtyping*. With *depth subtyping* casting no longer solely dependents on the top-level type structure, but also needs access to the innards in order to change the type of method arguments and return types. This is exactly what OOHaskell's *deep′narrow* function does (see section 5.9 [38]), leaning heavily on advanced type-level programming to make such generic record traversals possible. This is where our simple OO approach begins to crack in accordance with the predictions of OOHaskell's authors. Hence, we are forced to stick with a less powerful option where the type parameters are left invariant. As a consequence it is impossible e.g. to insert a *Pair Point Point* into a list with elements of type *Pair Object Object*.

Although we cannot use casts at the type parameter position, the typing of # is compatible with both width and depth subtyping similar to OOHaskell where they covered this fact extensively in section 5.9 and 5.10 [38]. We have transcribed their examples without any trouble[8].

5.5 Scraping the boilerplate

Given our decision to no use extensible records and with Haskell not being tailored towards OO programming it is only logical that there are quite a few steps involved to start implementing a new class:

1. Create a new record for representing the class' interface.
2. Make it an instance of some *Typeable* type class.
3. If the class is a subclass
   (a) Make each function available as a method by explicitly unrolling the object representation.
   (b) Make it an instance of both *Narrow* and *Widen*.

The first and second step are easily done manually. The third step is the most painful part where both the declaration of methods and the implementation of *Narrow* and *Widen* require nested record reads and writes which is known to be thorny issue[9].

The goal is the scrape as much boilerplate as possible. We see three possible plans of attack to achieve this goal:

1. Create a DSL for OO programming which translates back to regular Haskell.
2. Use Template Haskell.
3. Use the C pre-processor (CPP).

A DSL will lead to the most elegant solution with minimal input required by the programmer. Furthermore, and admittedly more important it can hide all the idiosyncrasies of the encoding. Creating such a DSL does require more research and we leave it as future work. Template Haskell would be an ideal trade-off, unfortunately we cannot use it as it is GHC specific. Consequently we are left with CPP which allows us to derive some of the boilerplate but not all it. For instance, it cannot generate class methods from an interface definition. What it can do is generate the boilerplate instances and tail manipulation functions with some help from the type class system.

We define two macros for deriving step (2) and (3 b) one for top-level classes `DefineClass` and the other for subclasses `DefineSubClass`. See A.2 for a description of their implementation.

[8]See: `https://github.com/rubendg/lightoo`

[9]For an ongoing discussion see: `http://hackage.haskell.org/trac/ghc/wiki/ExtensibleRecords`. There are also quire some approaches that uses lenses for dealing with records `http://brandon.si/code/haskell-state-of-the-lens/`

5.6 Discussion

5.6.1 Usability

When designing a new language one has maximal flexibility with respect to the syntax and semantics. Because we choose to embed our DSL in Haskell we inherit the limitations of the host language. The fact that implementation details reach the surface directly follows from this decision, even though the combinators and CPP macros help with hiding some of them. In particular our reliance on non-extensible records is a great source of trouble and a major factor in the abstraction leaks. Also, the peculiar combination of subtype constraints with tail-polymorphism does not result in an uniform treatment of subtyping. Uniformity is a key aspect in good language design, it contributes to the predictability of a language – an important factor in the usability of any language. Grasping the subtle details of the library is not for the faint of heart and significantly diminishes its usability.

Besides the limitations of our OO encoding there are also some aspects of Haskell that will trouble any embedding of OO-like code in Haskell. For example, the lack of mutually recursive modules makes properly organizing OO code difficult. When applying the one interface per file scheme it often turns out that method types necessitate that modules importing each other. The lack of recursive modules in Haskell then requires all interface definitions to reside inside a single file, something which is not only cumbersome from a organizational point of view, but also breaks encapsulation at the module level.

Despite these issues we are confident that we have improved the usability of the *"Mutable objects, with tail polymorphism"* approach [38], by providing a set of useful combinators bundled inside a ready to use library.

5.6.2 Efficiency

Contrary to OOHaskell which uses *polymorphic, extensible records of closures* for their object representation we use a much simpler model: *records of closures*. This decision has various implications for the efficiency of our encoding.

Similar to OOHaskell our object representation makes no distinction between an object's data and its methods. For efficiency reasons many OO languages do make such a distinction in order to share methods across all instances of a class. However, separating the two destroys the simplicity of the approach. Because we have not attempted to implement the optimization it remains unclear if it is even possible with our encoding.

In OOHaskell method-lookup is linear in the amount of methods. Our encoding has a more efficient method-lookup which is linear in the subclass depth. Unfortunately, we do not have constant-time record extension, but due to the nested record structure record extension that is linear in the subclass depth.

In an OO language one would typically expect that casting operationally corresponds to the identity function. OOHaskell shows in section 5.7 [38] that they support nominal subtypes by explicit nomination of the types on top of their structural record types. Using the nominal subtyping scheme they are able to implement an upcast which operationally corresponds to the identity function. They also suggest that *some forms* of downcasts can be implemented, but do not provide any further details. Our implementation of casting does, unfortunately, not correspond to the identity function, but requires repeated narrowing of which the complexity is given in section 5.4.1. Without the use of extensible records and type-level programming there appears to be no way around this.

5.6.3 Future work

We have focused on exploring and extending the *Mutable objects, with tail polymorphism* approach. It would be interesting to see whether some of the insights gained by our exploration can be transferred to the other more primitive encodings presented in OOHaskell. Furthermore, in our exploration we have limited ourselves to Haskell with a minimal amount of language extensions, lifting this restriction may yet lead to another OO encoding. Another interesting direction to look into is to see whether there exists a proper translation of Featherweight Java [36], a core calculus embedding the essence of Java, to our encoding.

Chapter 6

wxAsteroids in the web browser

In this chapter we will put the results from the previous two chapters to use by implementing a subset of wxHaskell that runs inside the web browser. The wxHaskell paper [42] explained its design and capabilities by implementing a clone of the classic asteroids game *wxAsteroids* [41]. It is a great showcase of the different key aspects of *wxHaskell*: widgets, graphics rendering, and user input. Furthermore, it provides a good example of how typical *wxHaskell* programs are constructed. Instead of porting the fully featured wxAsteroids we have ported a less feature heavy version due to time constraints. Figure 6.1 shows the original *wxAsteroids* on the left, running on the desktop, next to our port running on the desktop (middle), and in the web browser (right). It shows how LightOO together with the JavaScript FFI can be used to implement the wxWidgets OO design in Haskell in terms of the technologies available in the browser.

In this chapter we will explain the gist of *wxAsteroids*, the design issues, followed by a more detailed explanation of the implementation.

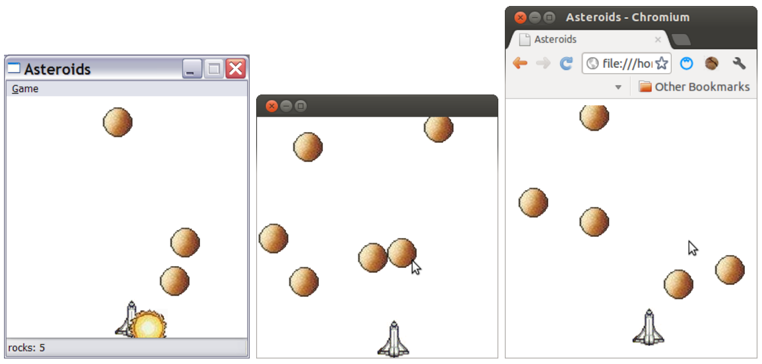


Figure 6.1: The original wxAsteroids on the desktop (left), modified wxAsteroids on the desktop (middle), and in the web browser (right).

6.1 wxAsteroids

In the *asteroids* game the player is tasked with carefully maneuvering its spaceship through an asteroid field making sure it does not get hit. The spaceship can move left and right using the arrow keys. There is an inifite supply of asteroids that move vertically downwards. Whenever a rock hits the spaceship, the rock turns into an explosion. In accordance with the original wxAsteroids hitting a rock does not destroy the spaceship. First, we define some constants:

```
height   = 600
width    = 300
diameter = 24
chance   = 0.1
```

The *height* and *width* values determine the dimensions of the game field. The *diameter* represents the diameter of a rock, and the *chance* determines the chance a new rock appears in a given time frame. The *asteroids* function constructs the user interface, and is run by the *start* function:

```
asteroids :: IO ()
asteroids = do
  vrocks ← varCreate randomRocks
  vship  ← varCreate $ div width 2
  w      ← window Nothing [ ]
  t      ← timer w [interval    := 50
                    ,on command := advance w vrocks
                    ]
  set w [area         := rect (pt 0 0) (sz width height)
        on paint      := draw vrocks vship
        ,on leftKey   := varUpdate vship (λx → max 0     (x − 5)) ≫ return ()
        ,on rightKey  := varUpdate vship (λx → min width (x + 5)) ≫ return ()
        ]
main = start asteroids
```

First two *mutable* variables are created: *vrocks* holds an infinite list containing the positions of all the future rock positions, *vship* contains the current position of the spaceship.

Next, we create a top-level window that serves as a placeholder for the game. The first parameter denotes a potential parent window, the second a list of properties. Subsequently we attach a timer to the window firing every 50 milliseconds. On each tick, it calls *advance*, moving all rocks to their next position and updating the screen.

Finally, we set a few attributes on the window *w*. We assign it an *area* with the given constant dimensions. The other attributes are prefixed with *on* designating event handlers. The *paint* event handler is invoked when a repaint request is made, and draws the current game state to the screen through *draw* (later defined). Pressing the left or right arrow key changes the *x* position of the spaceship.

The *vrocks* variable contains an infinite list of all future rock positions. This infinite list is generated by the *randomRocks* function which depends on random number generation. Because at the time of writing there was no back-end support for random number generation through the standard *System.Random* library we used a more *ad hoc* solution:

```
foreign import js "Math.random()"
  randomNumber :: IO Double
rand _ = unsafePerformIO randomNumber
randoms :: [Double]
randoms =
  let inf = ⊥ : inf
  in map rand inf
```

The *randoms* function provides a infinite list of random numbers in the range [0,1). It works by mapping a random number generator *rand* over an infinite list. Note that this only works because *rand* is not subject to let floating.

```
randomRocks = flatten [ ] (map fresh randoms)
fresh r
   | r > chance = [ ]
   | otherwise = [track (floor (fromIntegral width ∗ r / chance))]
track x = [point x (y − diameter) | y ← [0, 6 ... height + 2 ∗ diameter]]
flatten rocks (t : ts) =
  let now   = map head rocks
      later = filter (¬ ∘ null) (map tail rocks)
  in now : flatten (t ++ later) ts
flatten rocks [ ] = error "Empty rocks list not expected in function flatten"
```

The *fresh* function is mapped over *randoms*. It compares each number against the *chance* constant, and if a rock should appear it generates a finite list of future rock positions that move the rock from the top to the bottom of the screen, otherwise it returns the empty list. Finally, the *flatten* function flattens this list into a list of time frames, where each element contains the position of every rock in that particular time frame.

The *advance* function is called on every timer tick:

```
advance vrocks w = do
  (r : rs) ← varGet vrocks
  varSet vrocks rs
  repaint w
```

It moves *vrocks* to the next time frame (its tail), and request a repaint of the window (w). A repaint causes the *paint* event handler to be triggered which in turn calls *draw* with two parameters: the *graphics context* (gc) and view area $(view)$. The graphics context paints on the window area on the screen, but is in principal independent of its back-end.

```
draw vrocks vship gc view = do
  rocks ← varGet vrocks
  x     ← varGet vship
  let
    shipLocation = point x (height − 2 ∗ diameter)
    positions    = head rocks
    collisions   = map  (collide shipLocation) positions
  drawShip gc shipLocation
  mapM (drawRock gc) (zip positions collisions)
```

The *draw* function reads the current rock and spaceship positions, and positions the spaceship at the current x, and fixed y-position. Then it checks if there are any *collisions* between the spaceship and any of the rocks. Finally, we draw the spaceship and all the rocks onto the screen. The *collide* function simply checks whether a rock has entered the spaceship's comfort zone given their *positions*.

```
collide pos0 pos1 =
  let distance = vecLength (vecBetween pos0 pos1)
  in distance ⩽ fromIntegral diameter
```

Both entities are drawn using the *drawBitmap* function. It takes a graphics context, bitmap, position, transparency mode (not implemented), and a list of properties as arguments. Dependent on whether a collision occured the picture of a rock changes to either a normal rock or a exploded one.

```
drawShip gc pos = drawBitmap gc ship pos True [ ]
drawRock gc (pos, collides) =
  let rockPicture = if collides then burning else rock
  in drawBitmap gc rockPicture pos True [ ]
```

Finally, we specify the resources that we used.

```
rock    = bitmap "rock.ico"
burning = bitmap "burning.ico"
ship    = bitmap "ship.ico"
```

The summary we just gave does not differ much from the one given in [42]. We have simplified the porting effort by omitting features such as the menu, sound effects, and status field. Furthermore, we render the game on top of a *window* instead of a *frame* (which features all the standard decorations such as a title, maximize, minimize, and closing buttons). Besides the ommittances the source has remained largely the same.

6.2 Design

6.2.1 Approach

We took wxHaskell version 0.12.1.4 from hackage and performed a depth first search on the features that needed to be supported in order for wxAsteroids to work. All unused features were commented out and all irrelevant parts that dealt with the implementation details of the C++ back-end were removed. Furthermore, all relevant functions exposed through wxcore were undone from their implementation and replaced by *error* `"to be implemented"`. This allowed us to type check the whole codebase without yet having the implementation at hand. At all times we kept our codebase compatible with *ghci* by using CPP macros to conditionally import modules. With hindsight this turned out to be well worth the effort. Due to its superior error reporting capabilities we could more easily pinpoint programming errors. It also served as a useful reference for improving UHC by occassionally catching errors in its implementation.

Fortunately, none of the implementation details of *wxcore* leak to *wx*. This allowed us to leave the *wx* sources largely untouched. We used the elaborate wxWidgets documentation and source code as a reference for implementing wxcore.

6.2.2 Objects

In order to maintain some type safety when communicating with C++, wxcore assigns phantom types to objects, with as top-level type:

```
data Object a = Object    ! (Ptr a)
              | Managed ! (ForeignPtr (TManagedPtr a))
```

All objects are either a normal or managed pointer to an object that lives in the C++ world. For example, the type of a window (*Window a*) is a type synonym that expands to *Object* (*WxObject* (*CEvtHandler* (*CWindow a*))). The type structure used here is identical to the type structure we used to program OO in Haskell. This turns out the be very useful because it allows the implemention of wxcore functions without (in most cases) altering their interface which makes it very close to a drop-in replacement for the original wxcore implementation.

The *Object* data type is replaced with a record *IObject* with a corresponding implementation:

```
object = clazz $ λtail self → do
  flag ← newIORef False
  return IObject {
    setFlag     = writeIORef flag
    , getFlag     = readIORef flag
    , _objectTail = tail
  }
```

In *Graphics.UI.WXCore.Types* there are quite some methods defined for objects with the assumption that they are implemented as pointers to C++ objects. Some of these methods are replaced whereas others do not make sense anymore. For example, *objectCast* is replaced by *upcast* and *downcast*, *objectIsNull* makes no sense anymore as we always have evidence that an object exists there is no need to check whether it is null, *objectDelete* is also irrelevant we can simply rely on garbage collection.

What has also changes is object identity. Before, object identity boiled down to pointer equality. The idiomatic Haskell approach to test values for equivalence is by structural equivalence. However, objects are black boxes that hide their data. What makes two objects identical becomes subject to interpretation of the programmer. Hence, we implement an equality test similar to pointer equality [52]:

```
sameObject a b = do
  a # setFlag $ False
  b # setFlag $ True
  a # getFlag
```

Every object has a *flag* associated with it. The *sameObject* function tests if changing it in *a* also changes it in *b* essentially testing pointer equality.

6.2.3 Organization

We have set-up wxcore such that every class definition resides in its own module in similar to header files in C++. Implementations of a particular class import the interface definition and re-export it together with a class for constructing instances. Interface definitions typically need other interface definitions to define their type. This is no problem unless both definitions depend on each others types. Unfortunately, this is a quite common scenario in OO languages. Since Haskell does not support cyclic imports there is no other option than placing the interface definitions in the same module breaking the organization scheme.

6.2.4 Mapping to the web browser

The wxcore abstractions at some point need to interface to the target platform. The web browser already offers a great deal of functionality that make it easy create GUIs. We choose the easy route by piggybacking on much of the high-level technology is already present. We let a wxWidget window correspond to a HTML div element. Asteroids draws on a graphics context associated with a window. As back-end for the graphics context we use a HTML5 canvas element, because drawing on divs is not possible without CSS3 hacks. The graphics context is created on demand and covers the whole window. We could just as well have used SVG as a back-end, but it was easier to use the canvas due to its small API.

Event listeners are registered on the div representing the window. When an event is triggered by the target platform its event object properties are read and mapped to a subclass of the *Event* object. This object is then dispatched internally inside the wxWidgets event system which invokes to the appropriate event listeners. Decoupling the native event mechanism from the wxWidgets event mechanism provides the opportunity to trigger custom events not originating from the target platform.

Finally, initialization of the application is done through the *start* function:

```
start io =
  w ← htmlWindow
  -- Wrap a haskell function as if it were a regular JavaScript function
  cb ← wrapFunc io
  -- Set the onload event on the window object
  set "onload" cb w
  return ()
```

The moment the web page is loaded the application starts.

6.3 Implementation details

6.3.1 Subtyping

WxHaskell makes extensive use of a phantom type structure for modelling a type safe interface to foreign objects. In chapter 5, section 5.4.1 we explained that functions consuming abstract objects (i.e. objects with their tail left polymorph) are inhibited from using casts on those objects. Because there are many places where wxHaskell uses polymorphic objects - for attribute classes, as function arguments, inside properties - it make sense to explore whether this can persist if we provide a Haskell implementation, and if changes are required how this affects the interaction of the different parts of wxHaskell.

We start by looking at the *Graphics.UI.WX.Classes* module, part of the *wx* library. It defines a host of type classes for capturing common attributes of widgets. For example, there is a type class that ranges over widgets that can be positioned and sized:

```
class Dimensions w where
  area :: Attr w Rect
   ...
```

Typically, these attribute classes are instantiated by wxHaskell with polymorphic objects:

```
instance Dimensions (Window a) where
  ...
```

Fortunately, this part of wxHaskell is not affected by our implementation because it turns out that in all encountered cases the concerning widget is used solely as the first argument for invoking its own methods. Nevertheless it would have been possible to use these type classes if we were forced to instantiate them with concrete objects, there would simply be more explicit casting compensating for the loss in expressiveness in the types.

However, we have encountered cases wherein we were forced to change the type of a function. For instance, in the case of event processing wxHaskell invokes the *evtHandlerProcessEvent* method passing it any subtype of the *Event* class.

```
evtHandlerProcessEvent :: EvtHandler a → Event b → IO Bool
```

Because an *Event* object ends up as an argument to a callback function, which typically needs to know the event type (e.g. a mouse or keyboard event), there will be some casting involved. Hence we are forced to change the function type making it accept only concrete *Event* objects.

```
evtHandlerProcessEvent :: EvtHandler a → Event → IO Bool
```

Another thing we encountered that sometimes proves useful is to be able to change the type of property (*Prop w*), e.g. from a *Prop Frame* to a *Prop Window*. The *wx* library already defines casting functions for *Attr w a* and *Prop w*:

```
castProp :: (v → w) → Prop w → Prop v
castProp coerce prop =
  case prop of
    (attr := x)  → (castAttr coerce attr) := x
    (attr :~ f)  → (castAttr coerce attr) :~ f
    (attr ::= f) → (castAttr coerce attr) ::= (λv → f (coerce v))
    (attr ::~ f) → (castAttr coerce attr) ::~ (λv x → f (coerce v) x)
castAttr :: (v → w) → Attr w a → Attr v a
castAttr coerce (Attr name getter setter upd) =
  Attr name (λv → getter (coerce v)) (λv x → (setter (coerce v) x))
            (λv f → upd (coerce v) f)
```

Both functions take a coercion function and consistently apply it to the argument (contravariant) positions. The *Prop w* type is actually a contravariant functor with *castProp* as mapping function, because its type parameter *w* is only used in contravariant positions.

```
class Contravariant f where
  contramap :: (a → b) → f b → f a
instance Contravariant Prop where
  contramap = castProp
```

We specialize *contramap* for upcasting the properties:

```
upcastProp :: ∀v w.w > v ⇒ Prop v → Prop w
upcastProp p =
  contramap (handleErr ∘ (downcast :: w → Maybe v)) p
  where
  handleErr = maybe (error $ "Non-existent property: " ++ propName p) id
```

That the *upcastProp* function is implemented in terms of *downcast* may seem somewhat counterintuitive. Upcasting a property does not change the getter and setter stored in the attribute, but wraps them inside a new function accepting the upcasted type which is subsequently coerced back (downcasted) to the old type and applied to the wrapped function. Similarly, we define its dual:

```
downcastProp :: ∀v w.w < v ⇒ Prop v → Prop w
downcastProp p = contramap (upcast :: w → v) p
```

6.3.2 Interfacing with the DOM

For platform dependent features the implementation needs to communicate with JavaScript. In chapter 4 we developed some techniques which we are now going to use for interfacing with JavaScript, in particular the DOM. The DOM is the primary interface to browser functionality. Its interfaces are specified in IDL (Interface Definition Language), here follows an excerpt of the *HTMLElement* IDL definition[1]:

```
interface HTMLElement : Element {
  attribute DOMString title;
  ...
  readonly attribute boolean isContentEditable;
  ...
  [TreatNonCallableAsNull] attribute Function? onkeydown;
  ...
}
```

The top-level declaration provides a name for the interface, optionally followed by a colon and the interface it extends from. The body contains all attributes annotated with their contained type, additional constraints on the interpretation of the contained data (*TreatNonCallableAsNull*), and its usage (*readonly*).

To model the interface types we simply extend *JSObject*:

```
data CHTMLElement a
type HTMLElement_ a = Element (CHTMLElement a)
type HTMLElement = HTMLElement_ ()
data CElement a
type Element_ a = Node_ (CElement a)
type Element = Element_ ()
data CNode a
type Node_ a = JSObject (CNode a)
type Node = Node_ ()
```

The structure is exactly the same as the one used by wxHaskell to model C++ object types in Haskell. It is even a better fit for JavaScript, because its single prototype chain does not naturally allow the modeling of multiple inheritance, of which it is known that it cannot be practically modeled with this type structure [25].

For wxAsteroids it is important to know when the user holds down either the left or right key. We import the *onkeydown* event using the *eventJSRef* function:

```
onkeydown = eventJSRef "onkeydown"
```

It captures the onkeydown property as a read-write *JSRef*, interprets the *TreatNonCallableAsNull* constraint as a *Maybe*, and can be used to import any of the event handlers that are part of an *HTMLElement*.

[1] http://www.whatwg.org/specs/web-apps/current-work/multipage/elements.html#htmlelement

```
eventJSRef :: String → HTMLElement_ a → JSRef ReadWrite (Maybe (Event → IO ()))
eventJSRef id e =
  let get = do
        f ← getProperty id e
        if isNull f
          then return Nothing
          else unwrapFunc1 (unsafeCoerce f)
      set Nothing = setProperty_ id _null e
      set (Just f)  = do
        f ← wrapFunc1 f
        setProperty_ id f e
  in newJSRef get set
```

Internally it creates a new *JSRef* with a getter and setter that respectively read and write the property. Because there is no automatic back and forth conversion from Haskell functions to JavaScript functions we use *wrapFunc1* and *unwrapFunc1* for respectively wrapping and unwrapping 1-argument functions. These functions are part of a larger family of wrapping and unwrapping functions:

```
foreign import js "wrapper"
  wrapFunc :: IO () → IO (JSFunction (IO ()))
foreign import js "wrapper"
  wrapFunc1 :: (a → IO ()) → IO (JSFunction (a → IO ()))
foreign import js "dynamic"
  unwrapFunc :: JSFunction (IO ()) → IO (IO ())
foreign import js "dynamic"
  unwrapFunc1 :: JSFunction (a → IO ()) → IO (a → IO ())
...
```

Unfortunately, the wrapping and unwrapping is not tracked by the RTS. Repeated getting and setting will grow a series of wrapping and unwrapping functions around the original function causing performance problems. Also, the use of Haskell strings for accessing JavaScript properties is a root of performance problems as well. Strings need to be packed before the are in JavaScript format requiring order n time where n is the length of the string. A better solution would be to use overloaded strings[2] with compiler support for representing Haskell strings directly as JavaScript strings. Besides the technical details *eventJSRef* is a definite improvement upon the *laisser faire* attitude of JavaScript where a *TreatNonCallableAsNull* constraint depends run-time type checking, the event handler function can be any type of function making it very easy to let programming errors go by unnoticed.

6.3.3 Implementing wxTimer

We discuss the implementation of the *wxTimer* object, a small and relatively self-contained example that touches many of the aspects discussed thus far. We start at *wxAsteroids* moving stepwise from *wx* to the *wxcore* implementation.

[2]http://www.haskell.org/ghc/docs/7.4.2/html/users_guide/type-class-extensions.html#overloaded-strings

First, in *wxAsteroids*, we create a timer and attach it to a *Window*. We set the timer such that it calls *advance* every 50 milliseconds.

```
...
t  ← timer w [interval    := 50
             ,on command := advance vrocks w
             ]
...
```

The *timer* function is defined in *Graphics.UI.WX.Timer*, part of the *wx* package:

```
type Timer = TimerEx ()
timer :: Window a → [Prop Timer] → IO Timer
timer parent props
  = do t ← windowTimerCreate parent
       timerStart t 1000 False
       set t props
       return t
```

It creates a new timer using *windowTimerCreate*, sets the default resolution to 1 second, and sets some properties on the object. The *interval* attribute is specific to a timer object, and hence has its widget type fixed to a concrete timer.

```
interval :: Attr Timer Int
```

The *command* attribute is overloaded on the widget type as it can be reused by other widgets for setting zero-argument event handlers.

```
class Commanding w where
  command :: Event w (IO ())
instance Commanding Timer where
  command = newEvent "command" timerGetOnCommand timerOnCommand
```

Before we move on the the *wxcore* implementation, we only need to make a minor adjustment to the type signature of *timer* to account for name mismatch:

```
timer :: Window_ a → [Prop Timer] → IO Timer
```

In *wxcore* we find the definitions for *timerGetOnCommand*, *timerOnCommand*, etc. These functions are implemented with the C++ back-end in mind. For example, the *timerGetOnCommand* and *timerOnCommand* functions rely on some C++ wrapper code that allows storing and retrieval of Haskell closures. It ensures that Haskell functions can be safely passed into the C++ world without Haskell garbage collecting them.

```
timerOnCommand :: TimerEx a → IO () → IO ()
timerOnCommand timer io
  = do closure ← createClosure io (λownerDeleted → return ()) (λev → io)
       timerExConnect timer closure
```

```
timerGetOnCommand :: TimerEx a → IO (IO ())
timerGetOnCommand timer
    = do closure ← timerExGetClosure timer
      unsafeClosureGetState closure (return ())
```

Obviously, these implementations make no sense in our situation. We leave the function types in tact, but reimplement their functionality in terms of calls to a timer object. Before we present their new implementation we first provide a sample of the C++ implementation of the timer object:

```
class wxTimerBase : public wxEvtHandler
{
  public:
    wxTimerBase(wxEvtHandler *owner, int timerid = wxID_ANY) {
      Init();
      SetOwner(owner, timerid);
    }

    void SetOwner(wxEvtHandler *owner, int timerid = wxID_ANY) {
        m_owner = owner;
        m_idTimer = timerid == wxID_ANY ? wxWindow::NewControlId() : timerid;
    }

    int GetInterval() const { return m_milli; }
    bool IsOneShot() const { return m_oneShot; }
    ...
```

A timer inherits from *wxEvtHandler*, and is constructed by passing it an owner and optionally an identifier. Normally identifiers are used by wxWidgets to identify windows, but because we could not infer the use case of ids on a timer we left it out of our implementation.

```
timer owner =
  (timer′ ‘extends‘ evthandler) noOverride set_EvtHandler_Tail
  where
  timer′ tail super self = do
     ...
```

The constructor now corresponds to the *timer* function (different from the *timer* defined inside *wx*), and extends from *evthandler*, which we will not present for the sake of brievity. Inside the constructor some variables are brought into scope covering both the *Init* and *SetOwner* function:

```
interval  ← newIORef 0
owner     ← newIORef owner
jsTimerId ← newIORef (−1)
isone     ← newIORef False
isRunning ← newIORef False
return ITimer {
   ...
```

Most variables have corresponding methods for getting their values.

```
, _timerGetInterval = readIORef interval
, _timerGetOwner    = readIORef owner
, _timerIsOneShot   = readIORef isone
 ...
```

The *timerStart* method implements the functionality for starting a timer with a particular frequency (*milli*), and provides the possibility of firing the timer only once (*oneshot*).

```
, _timerStart = λmilli oneshot → do
   let this = upcast self :: Timer
   timingEvent ← new $ timerEvent this
   handler     ← readIORef owner
   let cb = do {
      ; handler # evtHandlerProcessEvent $ (upcast timingEvent)
      ; when oneshot (self # timerStop)
   }
   w        ← htmlWindow
   timerId ← setInterval w cb milli
   writeIORef jsTimerId timerId
   return True
```

The implementation makes fruitful use of the native *setInterval* function for installing a timed callback on the global window object. Inside the callback the *evtHandlerProcessEvent* is invoked on the owning object passing it an instance of a *TimerEvent*. If the timer is a one shot than it stops the timer from preventing any future invocations of the callback. The *setInterval* method returns an identifier which we store inside the *jsTimerId* variable such that we may later use it to stop the timer:

```
, _timerStop = do
   timerId ← readIORef jsTimerId
   clearInterval timerId
, _timerTail = tail
}
```

With the timer implementation we can now implement the *timerOnCommand* and *timerGetOnCommand* functions, which are no methods of the timer class, but helper functions created by wxHaskell.

```
timerOnCommand :: Timer_ a → IO () → IO ()
timerOnCommand t f = do
   owner ← t # timerGetOwner
   (owner # evtHandlerBind) wxEVT_TIMER (const f) idAny idAny
timerGetOnCommand :: Timer_ a → IO (IO ())
timerGetOnCommand t = do
   owner ← t # timerGetOwner
```

```
cb ← do {cd ← (owner # evtHandlerGetHandler) wxEVT_TIMER idAny idAny
   ; return $ maybe (const $ return ()) id cd
   }
return $ cb (error "touched event object")
```

The *timerOnCommand* function simply binds a callback to the owner of the timer, whereas *timerGetOnCommand* tries to retrieve an already bound callback. The *evtHandlerGetHandler* method had to be invented as its not part of the wxWidgets *EvtHandler* class, which can be justified by the fact that wxHaskell also uses wrapper code to implement this functionality.

Finally, the *windowTimerCreate* function simply instantiates a new timer.

```
windowTimerCreate :: Window → IO Timer
windowTimerCreate w = new $ timer (upcast w)
```

We should note that we have changed the type signature such that it takes a concrete window instead of a polymorphic one. We could have left it polymorph, because the owner object is never required as a concrete object inside the timer implementation, but doing this would have required that we made use of parameterized classes in effect complicating the types; we choose not to.

The wx timer now works inside the web browser, completely transparent to the end-user.

6.4 Conclusion

We have successfully ported a feature-light version of wxAsteroids to the web browser. Albeit the modest scope of the port we foresee no intrinsic difficulties in implementing the lacking features. We have discussed the design decisions involved and provided some details on the actual implementation. However, there is still much work to be done making decisions on how to best map wxWidgets features onto the web platform. Also, programming in LightOO feels a bit hacky due to the lack of an uniform treatment of subtyping which has forced us to make slight modifications to the wxcore interface. Furthermore, the lack of recursive modules breaks code organization and we expect that a full implementation of wxWidgets will soon run into performance problems related to the JavaScript back-end.

Chapter 7

Conclusion, Contributions & Future Work

Research question: *how can we make wxHaskell run in the web browser?*

To answer the research question we have explored the different paths that could potentially lead to a proof of concept implementation of wxHaskell for the web. Of these paths we picked the least obvious and most challenging one, developed the necessary tools, and applied them to successfully port a feature-light version of *wxAsteroids* which is *near to* interchangeable with the desktop version[1]. To the best of our knowledge we are also the first to actually implement a real-world OO design in Haskell.

Besides the implementation of a subset of wxHaskell we contribute two independently useful libraries: an extended JavaScript programming prelude[2], and a lightweight approach for OO programming in Haskell inspired by OOHaskell which only requires Haskell 98 plus some lifting of type class restrictions[3]. Almost all code snippets in this thesis can be found at [4].

From the discussions and conclusions of chapter 4, 5, and 6 it should be clear that in order for the wxWidgets implementation in Haskell to be of any real use there is still lots of work to be done. Also, some inherent limitations of the Haskell language such as the lack of first-class language support for extensible records and mutually recursive modules makes OO programming in Haskell feel a bit like a hack. It would be interesting to see if there exists a translation from feather-weight Java to our OO library as it would open up the possibility of creating a small language extension to hide the crufty details, and may even be able to work around the lack of an uniform treatment of subtyping in our encoding by inserting explicit casts at the required places. For all practical purposes, if the JavaScript FFI keeps on improving, design option B remains the most practical approach to implement a fully fledged port of wxHaskell for the web.

[1]https://github.com/rubendg/wxasteroids
[2]https://github.com/UU-ComputerScience/uhc-js
[3]https://github.com/rubendg/lightoo
[4]https://github.com/rubendg/thesis-snippets

Acknowledgements

At the end of my bachelor's study at the Applied Computer Science University Utrecht I had doubts about whether to pursue a master degree or not. Due to the discussions with my American colleges at the time, and my working experience among former university graduates who had greatly impressed me with their technical skills, I decided it was worth a try.

Getting through the courses has been a bumpy ride to say the least. I had to overcome quite a few gaps in my background knowledge. At some point I nearly lost confidence, but thanks to the wonderful Andres Löh I regained trust in that everything would work out fine. I experienced Andres as an enthusiastic, knowledgeable, and excellent teacher with a great knack for understanding what you do not understand, but are not yet able to express. Writing the thesis, doing research, has been a great learning experience. Especially, being more practically minded, it took some time to get comfortable with the mindset necessary for literally spending weeks on something without having a clue whatsoever if it will result in anything to show for.

First of all I would like to thank my supervisors Atze Dijkstra and Doaitse Swierstra for helping me with this project, and Atze for going out of line by improving UHC even when time would not allow it. Without UHC this project would not have been possible! I want to thank Sean Leather for proofreading my thesis and providing useful suggestions and criticism. Many thanks also to Jurrien Stutterheim and Alessandro Vermeulen for the discussions on the JavaScript back-end and their continuous effort to stay involved. Of course my thesis could not have been finished without a healthy dose of saccharine at the Pie-Thursdays, many thanks roomies Paul van der Walt, Ruud Koot, and Sjoerd Timmer. I sure did not intend to leave a mark on your theses the day you ate my rum-drenched tipsy cake! Finally, I would like to thank my family and friends for supporting me and listening to my endless ramblings about Haskell and otherwise unrelated topics.

"Haskell is the world's finest imperative programming language", Simon P. Jones

Appendix A

Appendix

A.1 XLib Hello World

```
#include <X11/Xlib.h>
#include <unistd.h>

#define NIL (0)

static const char text[] = "hello world!";

main()
{
      // Open the display
      Display *dpy = XOpenDisplay(NIL);

      int blackColor = BlackPixel(dpy, DefaultScreen(dpy));
      int whiteColor = WhitePixel(dpy, DefaultScreen(dpy));

      // Create the window
      Window w = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy), 0,
            0, 200, 100, 0, blackColor, blackColor);

      // Register MapNotify events
      XSelectInput(dpy, w, StructureNotifyMask);

      // Make the window appear on the screen
      XMapWindow(dpy, w);

      // Create a new graphics context
      GC gc = XCreateGC(dpy, w, 0, NIL);

      XSetForeground(dpy, gc, whiteColor);
```

```
    // Loop until we get a MapNotify event
    for(;;) {
        XEvent e;
        XNextEvent(dpy, &e);
        if (e.type == MapNotify)
            break;
    }

    XFontStruct *fs = XLoadQueryFont(dpy, "cursor");

    XDrawString(dpy, w, gc, 40, 50, text, sizeof(text));

    // Flush the commands to the X server
    XFlush(dpy);

    sleep(8);
}
```

A.2 LightOO Macros

We define CPP macros for deriving the boilerplate for top-level classes and subclasses. To ease their definition we create a helper type class that allows us to capture the set of tail manipulation functions:

```
class ModTail c where
  getTail :: c t → Record t
  setTail :: c t → Record t′ → c t′
  modifyTail :: (Record t → Record t′) → c t → c t′
  modifyTail = mkMod setTail getTail
mkMod set get f o = set o (f (get o))
```

For example, the instance for *Shape* looks like this:

```
instance ModTail IShape where
  getTail     = _shapeTail
  setTail o v = o {_shapeTail = v}
```

The *ModTail* type class does not yet provide us with access to the tail of a subclass, but we can use it to derive a family of functions for nested tail manipulation by expressing their tail manipulation functions in terms of their parent's.

Here is an example of a family of functions derived for *IShape* () and *IShape* (*IRectangle* ()):

```
 -- Shape
get_Shape_Tail :: IShape a → Record a
get_Shape_Tail = getTail
set_Shape_Tail :: IShape a → Record b → Shape_ b
set_Shape_Tail o v = setTail o v
```

$$
\begin{array}{l}
\textit{modify\_Shape\_Tail} = \textit{mkMod set\_Shape\_Tail get\_Shape\_Tail} \\
\text{-- Rectangle} \\
\textit{get\_Rectangle\_Tail} :: \textit{IShape}\ (\textit{IRectangle}\ a) \to \textit{Record}\ a \\
\textit{get\_Rectangle\_Tail} = \textit{getTail} \circ \textit{unRecord} \circ \textit{get\_Shape\_Tail} \\
\textit{set\_Rectangle\_Tail} :: \textit{IShape}\ (\textit{IRectangle}\ a) \to \textit{Record}\ b \to \textit{IShape}\ (\textit{IRectangle}\ b) \\
\textit{set\_Rectangle\_Tail}\ o\ v = \textit{modify\_Shape\_Tail}\ (\lambda o \to \textit{record}\ \$\ \textit{setTail}\ (\textit{unRecord}\ o)\ v)\ o \\
\textit{modify\_Rectangle\_Tail} = \textit{mkMod set\_Rectangle\_Tail get\_Rectangle\_Tail}
\end{array}
$$

Because the nested types can at times become quite lengthy we mold the types into a type synonym structure. This makes it easier for the programmer to read type errors, provide type annotations when necessary, and for the macros to generate code.

$$
\begin{array}{l}
\textit{type Shape\_}\ t = \textit{IShape}\ t \\
\textit{type Shape} = \textit{Shape\_}\ () \\
\textit{type Rectangle\_}\ t = \textit{Shape\_}\ (\textit{IRectangle}\ t) \\
\textit{type Rectangle} = \textit{Rectangle\_}\ ()
\end{array}
$$

The `DefineClass` macro can be used to derive the boilerplate for top-level classes, `DefineSubClass` for subclasses. Their definition is somewhat complicated by the fact that they are also suited for parameterized classes.

```
#define DefineClass(X,XC,XTAIL,AP,NP)
#define DefineSubClass(X,Y,XC,XTAIL,AP,YP,XP,NP,CONSTR)
```

- X, the name of the class
- Y, the name of the parent class
- XC, the name of the data type representing the class
- $XTAIL$, is the name of the function for manipulating the tail
- AP, all type parameters except the tail
- $YP \subseteq AP$, all type parameters that distribute to Y
- $XP \subseteq AP$, all type parameters that distribute to X
- $NP = |AP| + 1$, the number of type parameters
- $CONSTR$, a listing of *Typeable* constraints on the AP type parameters used for the *Narrow* and *Widen* instances

```
#define DefineClass(X,XC,XTAIL,P,NP) \
type X ## _ P t = XC P t ; \
type X P = X ## _ P () ; \
\
deriving instance Typeable ## NP XC ; \
\
instance ModTail (XC P) where { \
   getTail = _ ## XTAIL ; \
   setTail o v = o { _ ## XTAIL = v } } ; \
\
get_ ## X ## _Tail :: X ## _ P t -> Record t ; \
get_ ## X ## _Tail = getTail ; \
set_ ## X ## _Tail :: X ## _ P t -> Record tt -> X ## _ P tt ; \
set_ ## X ## _Tail o v = setTail o v ; \
modify_ ## X ## _Tail = mkMod set_ ## X ## _Tail get_ ## X ## _Tail ;

#define DefineSubClass(X,Y,XC,XTAIL,AP,YP,XP,NP,CONSTR) \
type X ## _ AP t = Y ## _ YP (XC XP t) ; \
type X AP = X ## _ AP () ; \
\
instance (CONSTR) => Narrow (X AP) (Y YP) where { \
   narrow = modify_ ## Y ## _Tail hideImpl } ; \
\
instance (CONSTR) => Widen (Y YP) (X AP) where { \
   widen o = genericWiden o get_ ## Y ## _Tail set_ ## Y ## _Tail } ; \
\
deriving instance Typeable ## NP XC ; \
\
instance ModTail (XC XP) where { \
   getTail = _ ## XTAIL ; \
   setTail o v = o { _ ## XTAIL = v } } ; \
\
get_ ## X ## _Tail :: X ## _ AP t -> Record t ; \
get_ ## X ## _Tail = getTail . headRecord . get_ ## Y ## _Tail ; \
\
set_ ## X ## _Tail :: X ## _ AP t -> Record tt -> X ## _ AP tt ; \
set_ ## X ## _Tail o v =
  modify_ ## Y ## _Tail (\o -> consRecord $ setTail (headRecord o) v) o ; \
\
modify_ ## X ## _Tail = mkMod set_ ## X ## _Tail get_ ## X ## _Tail ;
```

Bibliography

[1] ECMAScript Language Specification (Standard ECMA-262). Technical report.

[2] The glasgow haskell compiler. http://www.haskell.org/ghc/.

[3] York haskell compiler. http://www.haskell.org/haskellwiki/Yhc.

[4] Inc 280 North. Cappuccino, objective-j. http://cappuccino.org/.

[5] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language, 1989.

[6] Jeremy Allaire. Macromedia flash mx - a next-generation rich client. Technical report, Macromedia, March 2002.

[7] Kenichi Asai. On typing delimited continuations: three new solutions to the printf problem. *Higher Order Symbol. Comput.*, 22(3):275–291, September 2009.

[8] Joel Bjornson, Anton Tayanovskyy, and Adam Granicz. Composing reactive guis in f# using websharper. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 203–216, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] M. Bolin. *Closure: The Definitive Guide*. O'Reilly Series. O'Reilly Media, 2010.

[10] T. Burnham. *CoffeeScript: Accelerated JavaScript Development*. Pragmatic Bookshelf, 2011.

[11] F. Warren Burton. Type extension through polymorphism. *ACM Trans. Program. Lang. Syst.*, 12(1):135–138, January 1990.

[12] Manuel Chakravarty, New South, Wales Sigbjorn, Galois Connections, Fergus Henderson, Melbourne Marcin Kowalczyk, Utrecht Simon Marlow, Cambridge Erik Meijer, Microsoft Corporation, Sven Panne, and et al. The haskell 98 foreign function interface 1 . 0 an addendum to the haskell 98 report. *Interface*, 2003.

[13] William R. Cook, Brian Dalio, Tom Freeman, Craig Hansen-sturm, Victor Law, Leonard Nicholson, James Redfern, Tom Rockwell, and Chris Warth. A denotational semantics of inheritance. Technical report, 1989.

[14] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on*

Principles of programming languages, POPL '90, pages 125–135, New York, NY, USA, 1990. ACM.

[15] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *In 5th International Symposium on Formal Methods for Components and Objects (FMCO*. Springer-Verlag, 2006.

[16] Antony Courtney. Functionally Modeled User Interfaces. *Signals*.

[17] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, August 2003. ACM Press.

[18] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Incorporated, 2008.

[19] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the utrecht haskell compiler. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 93–104, New York, NY, USA, 2009. ACM.

[20] Atze Dijkstra, Jurriën Stutterheim, Alessandro Vermeulen, and S. Doaitse Swierstra. Building javascript applications with haskell. 2012.

[21] Anton Ekblad. Towards a declarative web, 2012.

[22] Levent Erkök, John Launchbury, and Andrew Moran. Semantics of fixio, 2001.

[23] Axel Simon et al. Gtk2hs.

[24] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. *SIGPLAN Not.*, 34(9):114–125, September 1999.

[25] Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. *J. Funct. Program.*, 16(6):751–791, November 2006.

[26] GNOME Foundation. The gtk+ project. http://www.gtk.org/.

[27] A. Fowler. A swing architecture overview. http://java.sun.com/products/jfc/tsc/articles/architecture/.

[28] Emden R. Gansner and John H. Reppy. *A foundation for user interface construction*, pages 239–260. A. K. Peters, Ltd., Natick, MA, USA, 1992.

[29] Jesse James Garrett. Ajax: A new approach to web applications. http://www.adaptivepath.com, 2 2005.

[30] J. Gettys, R.W. Scheifler, and R. Newman. *Xlib: C language X interface (X version 11, release 4)*. Silicon Press, 1990.

[31] A. Ghoda. *Introducing Silverlight 4*. Apress Series. Apress, 2010.

[32] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[33] Google. The dart programming language specification. http://www.dartlang.org/docs/spec/dartLangSpec.pdf.

[34] R. Hanson and A. Tacy. *GWT in action: easy Ajax with the Google Web toolkit*. Manning Pubs Co Series. Manning, 2007.

[35] Jan Rune Holmevik. Compiling simula: A historical study of technological genesis. *IEEE Ann. Hist. Comput.*, 16(4):25–37, December 1994.

[36] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[37] O. Kiselyov. Type-safe functional formatted io.

[38] Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. Draft, 2005.

[39] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell 2004: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[40] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1:26–49, August 1988.

[41] Daan Leijen.

[42] Daan Leijen. wxHaskell: a portable and concise GUI library for haskell. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 57–68, New York, NY, USA, 2004. ACM Press.

[43] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system, release 3.08*. INRIA-Rocquencourt, 2004. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[44] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM.

[45] C. Lindley. *JQuery Cookbook*. O'Reilly Series. O'Reilly Media, 2009.

[46] Tommi Mikkonen and Antero Taivalsaari. Using javascript as a real programming language. Technical report, Mountain View, CA, USA, 2007.

[47] Tommi Mikkonen and Antero Taivalsaari. Web applications - spaghetti code for the 21st century. In *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications*, pages 319–328, Washington, DC, USA, 2008. IEEE Computer Society.

[48] M. Naftalin and P. Wadler. *Java Generics and Collections*. Java Series. O'Reilly Media, 2006.

[49] Rob Noble and Colin Runciman. Gadgets: Lazy functional components for graphical user interfaces. pages 321–340. Springer Verlag, 1995.

[50] Nokia. The qt project. http://qt.nokia.com/.

[51] Johan Nordlander. Polymorphic subtyping in o'haskell. In *APPSEM Workshop on Subtyping and Dependent Types in Programming, 2000*, 2001.

[52] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[53] Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. In *In Proc 20th ACM Symp. Principles of Programming Languages*, pages 299–312.

[54] M. Russell. *Dojo: the definitive guide*. Definitive Guide Series. O'Reilly, 2008.

[55] Robert W. Scheifler and Jim Gettys. The x window system. *ACM Trans. Graph.*, 5:79–109, April 1986.

[56] Erik Van Seters. wxFlashkell: Building Flash based GUI's in Haskell. 2009.

[57] J. Smart, K. Hock, and S. Csomor. *Cross-platform GUI programming with wxWidgets*. Bruce Perens' Open Source series. Prentice Hall PTR, 2006.

[58] Doaitse S. Swierstra, Pablo, and Joao Sariava. Designing and Implementing Combinator Languages. In *Advanced Functional Programming*, pages 150–206, 1998.

[59] Wouter Swierstra. Data types & la carte. *J. Funct. Program.*, 18:423–436, July 2008.

[60] Peter Thiemann. Towards a Type System for Analyzing JavaScript Programs. In Mooly Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, chapter 28, page 140. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.

[61] W3C. Html5. http://www.w3.org/TR/html5/.

[62] W3C. Svg primer. http://www.w3.org/Graphics/SVG/IG/resources/svgprimer.html.

[63] J. Weaver and J.L. Weaver. *JavaFX Script: Dynamic Java Scripting for Rich Internet/Client-side Applications*. Apress, 2007.

[64] S.T. Young, M. Givens, and D. Gianninas. *Adobe AIR programming unleashed*. Unleashed Series. Sams, 2008.

[65] F. Zammetti. *Practical Ext JS Projects with Gears*. Practical Projects. Apress, 2009.