The Monad.Reader Issue 1

by Andrew J. Bromage ⟨ajb@spamcop.net⟩
and Autrijus Tang ⟨autrijus@autrijus.org⟩
and Kenneth Hoste ⟨kenneth.hoste@UGent.be⟩
and Simon D. Foster ⟨u1sf@dcs.shef.ac.uk⟩
and Sven Moritz Hallberg ⟨pesco@gmx.de⟩

The Monad Reader

Shae Erisson, editor.

Contents

Pseudocode: Natural Style

By Andrew J. Bromage – email: ajb@spamcop.net

Welcome to Pseudocode!

This this series of articles (hopefully monthly, assuming that I get around to it) serves two main purposes. Firstly, I'll be presenting some "recreational programming" problems. Some of these problems will to illustrate something specific. Others will be just because it ticked my fancy, so to speak. Secondly, since I don't keep a blog, this is my virtual soapbox. You see, dear reader, I subscribe to the Usenet Theory of Finding Things Out: Don't ask, because people will ignore you. Post wrong information instead, and people will rush to correct you. So read on for this month's wrong information.

Programming and Writing

I have a confession to make. I used to be a Perl hacker.

It's not something that I'm especially proud of. But, as Alan Perlis put it, if a language doesn't change the way you think about programming, it's not worth knowing. In that sense, being a reformed Perl hacker is something that I'm not ashamed of, either.

One of the things that I learned from Perl is the relationship between programming languages and natural languages. In particular: **Programming is writing.**

This should be obvious, but all the best ideas are.

Back when I tutored undergraduate students, I was often asked what constitutes "good style". It's a hard question to answer, especially about a language like Haskell which, at the time, didn't have a huge corpus of software to compare against. There was GHC, but it wasn't exactly a shining example of readability at the time. (A lot of it arguably still isn't.) A lot of it was written in monadic style, but was written before the advent of constructor classes, let alone do-notation. First-year students find recursion hard enough to understand, let alone that.

But for programming in a more common language, like C, the best analogy – and I wish I could remember who taught me this – was to writing in a native language. You, dear student, are being taught to write software. If we were teaching you to write novels, we would expect you to have read a few novels written by other people first. So read other peoples' programs, but read them critically. Only this way will you learn "good style".

In my humble opinion, we don't push this analogy far enough, though to our credit, we do teach students some of its more important aspects. For example, we drum into

them the idea that your program must be readable above all else. Your program, dear student, will be read by someone else, so try to make sure that they can. The typical student, of course, only learns this valuable lesson by **being** the next person, several months later. That was certainly true in my case.

The writing analogy holds for professionals, too. Henry James famously said: "All writing is rewriting." That's also true of programming. All programming is reprogramming, only we tend to use other terms, like "cleaning up", "maintaining" or "refactoring".

Refactor aggressively

A phenomenon which you occasionally find on the Haskell mailing lists is what I call "micro-refactoring by committee".

It starts when a newbie asks: "How can I improve this program?" Over the next couple of days, suggestions arrive, each more interesting than the previous.

A conversation might start something like this:

```
Hi.  Can someone critique this code for me?  Thanks!

reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

To which someone will point out:

```
Accumulator recursion is better in this case:

reverse xs = rev' xs acc
  where
     rev' [] acc     = acc
     rev' (x:xs) acc = rev' xs (x:acc)
```

To which someone will reply:

```
You should use foldl!

reverse xs = foldl snoc [] xs
  where snoc xs x = x:xs
```

And on the conversation will go until the program looks something like this:

```
reverse = foldl (flip (:)) []
```

In the end, we'll end up with an extremely short program, which the original poster, being a newbie, requires a nontrivial amount of effort to decipher.

On one hand, this is a useful exercise. Acknowledging the multitude of ways to do something is good. On the other hand, the end result is usually quite silly. Yes, it's

great if we want to know the Kolmogorov complexity of `reverse`, but in our rush to explore the solution space, I wonder if we're sometimes forgetting the rule that we drum into students: In writing code, readability is the most important thing!

Or maybe there are just more ex-Perl hackers out there in Haskell land that we care to admit...

Natural transformations

If I may be so bold, I'm going to propose what I consider the most important rule of programming style. It doesn't trump other rules, but rather, it's the rule which helps you decide which other rules to apply.

The rule is: **Be natural.**

The biggest problem with the final version of `reverse` is that it's unnatural. If I had to write that function from scratch, and assuming that it wasn't in the Prelude and I didn't already know the answer by heart, I would not do it that way.

Partly, my native language flows from left-to-right across the page, and I tend to view any code which contains `flip` with deep suspicion, since it reverses the "natural" order for me.

Mostly, I strongly suspect that most code of this type is not written, but rather it's **translated into**. I tend to see many uses of "point-free style" in the same light. It's also one of the reasons why I've resisted using arrows: You don't program in arrow style; you program in diagrams on paper, then translate that into arrow style.

Any time you find yourself translating your code into some style, you increase the chances that the "next person" (who, you will recall, may well be yourself) will have to translate back into what you wrote originally.

Of course I don't mean "keep what you first wrote". If you're a fallible human, then the code that you write first certainly won't be precisely the same as the code that gets released. But it does suggest that every time you modify, refactor or otherwise transform your code, you should be careful to maintain naturality. At any point, your program should be something that you **could** have written first time, assuming you were a more perfect programmer than you are.

Programming Challenge

Okay, enough ranting. On with this month's problem.

Take a list S. Delete some elements from the list. What you have left is a **subsequence** of S. For example, $[1, 3, 2]$ is a subsequence of $[2, 1, 2, 3, 2, 1, 3]$, because you can obtain the former by deleting elements from the latter.

Consider the list $[1, 2, 3, 1, 2, 3, 1]$. This string contains all permutations of the list $[1, 2, 3]$ as subsequences. It is also minimal, in the sense that there is no shorter subsequence which will do (though there are potentially many minimal subsequences). we will call such a list a **shortest supersequence** over the alphabet [1..3].

Let $S(n)$ be the length of the shortest supersequence over the alphabet $[1..n]$. The task is to find out some interesting things about $S(n)$. Any interesting thing will do, but as a suggestion:

- Write a Haskell function to test if xs is a supersequence over the alphabet $[1..n]$. There's an obvious algorithm which is $O(n!length(xs))$. Can you do better?
- Write a Haskell function to produce the shortest supersequence over the alphabet $[1..n]$.
- Find bounds on S(n). Clearly n is a lower bound. An upper bound is $n^2 - n + 1$. (Why?) Can you do better? How about a recurrence?
- For the alphabet $[1..n]$, find bounds on the **number** of shortest supersequences. (It must be a multiple of $n!$, for obvious reasons.)

Email your findings to me by the 22nd of March, 2005, for credit in next month's The Monad.Reader.

Pugs Apocryphon 1 – Overview of the Pugs project

Autrijus Tang – email: autrijus@autrijus.org

The Pugs Apocrypha are a series of documents, written in question/answer format, to explain the design and implementation of Pugs. This document (PA01) is a higher-level overview of the project.

What is this document about?

The Pugs Apocrypha are a series of documents, written in question/answer format, to explain the design and implementation of Pugs. This document (PA01) is a higher-level overview of the project.

What is Perl 6?

Perl 6 is the next major revision of Perl, a context-sensitive, multi-paradigmatic, **practical** programming language, designed by a team led by Larry Wall. The Pugs project has been enthusiastically welcomed by the Perl 6 team.

Has Perl 6 been specified?

By December 2004, most of Perl 6 has been specified as a series of Synopses. Although not considered final, it is now stable enough to be implemented. Many of the Synopses are based on Larry's Apocalypses. Sometimes the design team releases Exegeses, which explain the meaning of Apocalypses. Pugs adheres to the Synopses, referring to Apocalypses or Exegeses when a Synopsis is unclear or imprecise.

What does "Apocrypha" mean?

The word Apocrypha, from the Greek ἀπόκρυφος, "hidden", refers to religious works that are not considered canonical, or part of officially accepted scripture. The proper singular form in Greek is **Apocryphon**.

What is the relationship between Apocrypha and the Perl 6 design documents?

Apocalypses and Synopses cover the Perl 6 language in general; Apocrypha are specific to the Pugs implementation. Like Parrot Design Documents, Apocrypha will be constantly updated according to the status of Pugs.

Will Pugs implement the full Perl 6 specification?

Yes. Pugs always targets the latest revision of Perl 6 Synopses. As soon as a new revision or a new Synopsis is published, incompatibilities between Pugs and the new version will be considered bugs in Pugs.

Is Pugs free software?

Yes. It is available under both GPL version 2 and Artistic License version 2.0b5. Once the final version of Artistic 2.0 is released, Pugs will adopt it.

Is Pugs funded by the Perl Foundation?

No. After receiving three Perl Foundation grants on various projects, Autrijus decides it would be more helpful to donate time to the Perl 6 project by hacking Pugs, rather than asking TPF for money to do the same thing.

Where can I download Pugs?

For the very latest version of Pugs, check out the source from Subversion or darcs repositories. Periodic releases are available on CPAN under the Perl6-Pugs namespace. (By the way, if you'd like offline working with the Subversion repository, the svk client may be of interest. But using vanilla svn is fine.)

How do I build Pugs?

Pugs uses the standard Makefile.PL build system, as detailed in the README file. Since Pugs is written in Haskell, you will need Glasgow Haskell Compiler (GHC) 6.2 or above. Please download a binary build for your platform; compiling GHC from source code can take a very long time.

What is Haskell?

Haskell is a standardized, purely functional programming language with built-in lazy evaluation capabilities. While there are several different implementations available, currently Pugs needs to be built with GHC, because it uses several GHC-specific features.

What is GHC?

GHC is a state-of-the-art compiler and interactive environment, available under a BSD-style license. Itself written in Haskell, GHC can compile Haskell to bytecode, C code, and machine code on some platforms. GHC has an extensive library, numerous language extensions, and a very capable optimizer (with some help from a Perl 5 program). As such, it provides an excellent platform to solve Perl 6's **bootstrapping problem**.

What is the Perl 6 bootstrapping problem?

The goal of the Perl 6 project is to be **self-hosting**: The Perl 6 compiler needs to be written in Perl 6 itself, and must parse Perl 6 source code with Perl 6 Rules, which is a subset of the Perl 6 language. The generated code must also contain an evaluator that can execute Perl 6 code on the fly. The only way to break this cycle of dependency is by first implementing some parts in other languages, then rewrite those parts in Perl 6.

What was the initial bootstrapping plan?

According to the Parrot FAQ, the initial plan was to bootstrap via Perl 5: First we extend Perl 5 to run on the Parrot virtual machine (via `B::Parrot` or Ponie), and then implement the Perl 6 compiler in Perl 5, which will be translated to Perl 6 via a p5-to-p6 translator. However, although part of the Rule system was prototyped in Perl 5 as `Perl6::Rules`, it was not mature enough to build a compiler on. As such, the plan was revised to bootstrap via C instead.

What was the revised bootstrapping plan?

According to an early 2005 proposal, the plan is to first implement the Rule engine in C (i.e. PGE), use it to parse Perl 6 into Parrot as an abstract syntax tree (AST), and then implement an AST evaluator as part of Parrot. Ponie and p5-to-p6 are still in progress, but they are no longer critical dependencies in the bootstrapping process.

How can Pugs help Perl 6 to bootstrap?

In a bootstrapping process, there are often many bottlenecks, which prevent people from working on things that depend on them. For example, one cannot easily write unit tests and standard libraries for Perl 6 without a working Perl 6 implementation, or work on an AST evaluator without an AST interface. Pugs solves such deadlocks by providing ready substitutes at various level of the process.

How can Pugs help the Perl 6 language design?

Inconsistencies and corner cases in the specification are very hard to spot without a working implementation. However, if a design problem is found late into the implementation, it may require costly re-architecture for everything else. By providing a working Perl 6 implementation, Pugs can serve as a proving ground to resolve problems as early as possible, as well as encourage more people to exercise Perl 6's features.

Why did you choose Haskell?

Many Perl 6 features have similar counterparts in Haskell: Perl 6 Rules corresponds closely to Parsec; lazy list evaluation is common in both languages; continuation support can be modeled with the ContT monad transformer, and so on. This greatly simplified the prototyping effort: the first working interpreter was released within the first week, and by the third week we have a full-fledged `Test.pm` module for unit testing.

Is Pugs a compiler or an interpreter?

Similar to Perl 5, Pugs first compiles Perl 6 program to an AST, then executes it using the built-in evaluator. However, in the future Pugs may also provide a compiler interface that supports different compiler backends.

Which compiler backends do you have in mind?

If implemented, the first compiler backend will likely generate Perl 6 code, similar to the `B::Deparse` module. The next one may generate Haskell code, which can then be compiled to C by GHC. At that point, it may make sense to target the Parrot AST interface. We can also add other backends (such as Perl 5 bytecode) if people are willing to work on them.

Do you have a roadmap for Pugs development?

The major/minor version numbers of Pugs converges to 2π; each significant digit in the minor version represents a milestone. The third digit is incremented for each release. The current milestones are:

- 6.0: Initial release.
- 6.2: Basic IO and control flow elements; mutable variables; assignment.
- 6.28: Classes and traits.
- 6.283: Rules and Grammars.
- 6.2831: Role composition and other runtime features.
- 6.28318: Macros.
- 6.283185: Port Pugs to Perl6, if needed.

How portable is Pugs?

Pugs runs on Win32, Linux and many flavors of Unix systems. See GHC's porters list and download page for details. Starting from 6.2.0, the Pugs team will also provide binary builds on selected platforms.

How fast is Pugs?

The parser part of Pugs is very fast, due to its robust underpinning in Parsec. However, the Pugs evaluator is currently not optimized at all: dispatching is around 1000 operators per second on a typical PC, which is nearly 100 times slower than Perl 5. Still, it is fast enough for prototyping language features; if you need fast operations in Pugs, please consider helping out the Compiler backend.

Is there a CPAN for Perl 6 modules?

Currently, Pugs searches for Perl 6 libraries under the `Perl6::lib` namespace in the Perl 5 search path. For example, the Test module is installed as `Perl6/lib/Test.pm` in Perl 5's site library path. This is a temporary measure; we expect more robust solutions in the future.

Can Pugs work with Perl 5 libraries?

Not yet. However, we may write a `Inline::GHC` module in the future, allowing interaction between Perl 5 modules and Haskell modules, similar to Autrijus' previous work on `Inline::MzScheme`; if that happens, then it is trivial to build `Inline::Pugs` on top of it. Alternatively, we may implement a Perl 5 source code parser that emits Pugs AST code, which will make pure Perl modules work on Pugs. Finally, it is also conceivable to compile Pugs AST into Perl 5 AST, but that is even more speculative.

Can Pugs work with Haskell libraries?

Currently, you can statically link Haskell libraries into Pugs primitives, by modifying a few lines in Prim.hs. We are considering writing a simple interface to hs-plugins, which will let Pugs dynamically load Haskell libraries, even inline Haskell code directly within Perl 6.

Can Pugs work with C libraries?

Not yet. However, HaskellDirect seems to provide an easy way to interface with C, CORBA and COM libraries, especially when combined with hs-plugins described above.

I know Perl 5, but not Haskell. Can I develop Pugs?

Sure! The standard libraries and unit tests that come with Pugs are coded in Perl 6, and there is always a need for more tests and libraries. All you need is basic familiarity of Perl 5, and a few minutes to get acquainted with some small syntax changes. You will likely pick up some Haskell knowledge along the way, too.

I know Haskell, but not Perl 5. Can I develop Pugs?

Sure! Perl 6 and Haskell have many things in common, such as type-based function dispatch, first class functions and currying, so picking up the syntax is relatively easy. Since there are always some TODO tests for features in need of implementation, it is never hard to find something to do.

I have learned some Perl 6. What can I do with Pugs?

Look at the examples/ directory to see some sample programs. Some people are already writing web applications and report systems with Pugs. If you run into a missing feature in Pugs, please let us know so we can implement it.

Where can I learn more about Haskell?

The Haskell homepage and the Wiki are good entry points. Of the many online tutorials, Yet Another Haskell Tutorial is perhaps the most accessible. Due to the ubiquitous use of Monad transformers in Pugs, All About Monads is also recommended. For books, Algorithms: A Functional Programming Approach, Haskell: The Craft of Functional Programming and The Haskell School of Expression are fine introductory materials. Finally, the `#haskell` channel on freenode is full of helpful and interesting people.

Where can I learn more about Perl 6?

The Perl 6 homepage provides many online documents. Every week or two, a new Perl 6 list summary will appear on Perl.com; it is a must-read for people who wish to follow Perl 6's progress. For books, Perl 6 and Parrot Essentials and Perl 6 Now are both helpful.

Where can I learn more about implementing programming languages?

Types and Programming Languages is an essential read; Pugs started out as a self-initiated study of the text, and it continues to be an important guide during the implementation. Its sequel, Advanced Topics in Types and Programming Languages, is also invaluable. It may also help to get acquainted with other multi-paradigmatic languages, such as Mozart/Oz, Curry and O'Caml. Finally, the detailed GHC commentary describes how GHC itself was implemented.

I'd like to help. What should I do?

First, subscribe to the perl6-compiler mailing list by sending an empty mail to perl6-compiler-subscribe@perl.org. Next, join the `#perl6` IRC channel on irc.freenode.net to find out what needs to be done. Commit access is handed out liberally; contact the Pugs team on `#perl6` for details. See you on IRC!

An Introduction to Gtk2Hs, a Haskell GUI Library

By Kenneth Hoste – email: kenneth.hoste@UGent.be

This article is an introduction to Gtk2Hs, one of many Haskell GUI libraries. We introduce the library by means of a small example, which we will build from scratch during this article. The emphasis is on the Gtk2Hs related code.

Introduction

Since this article is an introduction to **Gtk2Hs**, we will start with some words on the library itself: the structure of the library, its advantages and disadvantages, and how to install and use it.

In order to make the explanation of the basic principles behind Gtk2Hs a little bit easier, we use an example application: the Memory game. The code needed to run the application was written to serve as example code for this article, and thus is not meant to be fully working, or even bug-free.

Because one of the big advantages of the Gtk2Hs library is the support of Glade, we show how we can access a GUI description created with Glade. The main part discusses the Gtk2Hs related functionality of the Memory application. This includes setting up communication between the different parts of the program. We show how we can ensure the interaction with the user runs smoothly.

To conclude the article, we review the important items we discussed, and try to see what the future could bring us.

What is Gtk2Hs?

As we mentioned above, Gtk2Hs is a **GUI library for Haskell**. It is based on **Gtk+** (version 2.6), a multi-platform toolkit for creating graphical user interfaces. Some of its features we use in this article include nearly complete coverage of the Gtk+ toolkit, API documentation (still in development), bindings for several Gnome modules (more specifically: libglade for loading GUIs from xml files at runtime, GConf for storing application preferences and SourceView, a source code editor widget with syntax highlighting), support for GNU/Linux, Mac OS X and Windows platforms, ...

The library has reached version 0.9.7, and thus is still largely under development. Support for using the library can be found on the Gtk2Hs mailing list. When this article is published, a fully working 0.9.7.1 build for Windows should be available. Instructions on how to install Gtk2Hs on Windows are available on the website (`http://gtk2hs.sourceforge.net/archives/2005/02/17/installing-on-windows`). The API is quite useful already, although it doesn't contain all the desired information.

When talking about a GUI library, it is often useful to first explain some of the used terminology, as we do not expect everybody to be familiar with several of the UI terms we use in this article, such as (cfr. Wikipedia):

- widget: a component of a graphical user interface that the user interacts with. Examples: button, label, scroll bar, ...
- container: a widget which is able to contain other widgets
- box: a container which aligns all of its widgets either in a vertical or horizontal way

To use Gtk2Hs in a Haskell program, you should install Gtk2Hs and import the Gtk2Hs modules you need. For the Windows platform (and maybe others too), this installation will require Gtk+ to be installed (to install Gtk+ on the Windows platform, see *gladewin32.sourceforge.net*), and to be able to use the Glade functionality, you should also have Glade installed (more information, see the 'Using Glade' section).

Once everything is installed properly, it should suffice to add an import statement at the beginning of the program code, i.e.:

```
import Graphics.UI.Gtk
```

This lets the Haskell compiler know we want to use the functionality provided by the Gtk2Hs package.

Some people would argue that there are dozens of Haskell GUI libraries out there, so why would this one be any better than the rest? To point out the advantages of Gtk2Hs over other Haskell GUI libraries, we give a brief overview of what Gtk2Hs does better than most of the other libraries:

- **Glade support** First of all, using Glade you can design a GUI visually rather than having to write code. This allows one to follow the HIG (the Gtk/Gnome Human Interface Guidelines: `http://developer.gnome.org/projects/gup/hig`) much more easily. It also allows Gtk2Hs to read the GUI definition at runtime. When the user wants to change some small things about the GUI, or even completely re-design it, no recompiling of the Haskell code is needed. Just make sure the new *.glade file has the same name as the last one, and contains the same widgets (with the same names) as the last GUI definition.
- **API reference documentation** An API is a tool which every serious developer needs. The Gtk2Hs tool which is available now, isn't complete yet, but several people are putting a lot of effort into it. The Gtk+ API (`http://gtk.org/api`) could also be of some help, since Gtk2Hs is a mapping of the Gtk+ functionality to Haskell.
- **Unicode support**

- **Memory management**
- **Bindings for the Mozilla browser rendering engine**

To conclude this section, we list some of the other Haskell GUI libraries available.

- **wxHaskell** (*wxhaskell.sourceforge.net*) - built on top of wxWidgets, a comprehensive C++ library across all major GUI platforms, including Gtk, Windows, X11 and Mac OS X
- **FranTk** (*haskell.org/FranTk*) - a declarative library for building GUIs in Haskell, running on top of Tcl-Tk (working via TclHaskell)
- **HToolKit** (*htoolkit.sourceforge.net*) - a portable Haskell library for writing graphical user interfaces. To ensure portability, the library is built upon a low-level interface PORT, which is currently implemented for Gtk and Windows.
- **Other** (*haskell.org/libraries/#guigs*) - check here for information about other Haskell GUI library projects

The example program: Memory, the game

Before we show how to code the example we're using, let's see what it is supposed to do.

The Memory game is a simple card game. The goal of the game is find all pairs of equal images on the cards, which are all upside down when the game starts. We will allow the user to set the game level between 1 and 9. The level indicates how many pairs the deck of cards has: level 1 means just one pair (very easy), level 9 means nine pairs (quite 'hard'). When the user has set a level, he should be able to start a new game. We will use a control panel to provide this functionality.

In order to play the game, the player must be able to flip cards over, and see what picture is on them. Once he has flipped over two cards (and thus has tried to find a pair), the program should decide whether the cards match or not. When the player flips over another card, the game should visualise the match by means of some picture (in our case, a smiley icon), or just flip the cards back again when there was no match.

As a teaser, we show a screenshot of the Memory game which we have written. Some cards have been matched, two cards are flipped over (but no match is found), and the rest of the cards are upside down. On the right, the control panel is shown where the user can adjust the game level and/or start a new game.

The GUI: Using Glade

Glade is a tool to create XML descriptions of a graphical user interfaces. Using Glade, it is not necessary to write application code to construct GUIs. Moreover, it lets us redesign the GUI appearance without having to touch the code. No knowledge of Glade is needed to be able to understand this article. It was used to demonstrate one of many features of Gtk2Hs.

If you want to find out more about the use of Glade, check out the following urls: `http://glade.gnome.org` for GNU/Linux, or `http://gladewin32.sourceforge.net`

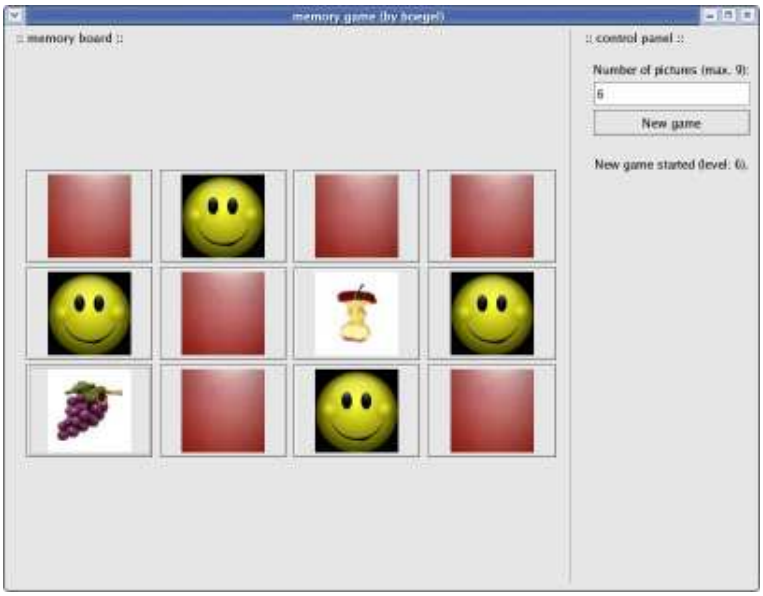

Figure 1: Screenshot of the Memory game

for the Windows platform).

The code: reading the glade description

After using Glade to create the GUI, it is necessary to load the description of the various widgets in the GUI before they can be used, e.g. to define the communication between widgets, in an interactive application written in Haskell.

To make the Glade functionality available in Haskell, we should import the Glade module defined in the Gtk2Hs library first:

```
import Graphics.UI.Gtk.Glade
```

The first thing to do before loading the widgets defined in the Glade description, is to load the description itself. Because the *.glade file is needed to run the application, we include a check too see whether the description file can be found where it is expectedx. As such, we can inform the user with an appropriate error message if no conforming Glade file is available.

The load the file, we use the `xmlNew` function provided by the Glade module in the Gtk2Hs library. The `Maybe` module makes an excellent tool to help us check if the file was available. If it was, we use the description it contains. Otherwise, we throw an error, which will inform the user of what went wrong.

```
windowXmlM <- xmlNew "memory.glade"
let windowXml = case windowXmlM of
```

```
  (Just windowXml) -> windowXml
  Nothing -> error "Can't find the glade file \"memory.glade\"
                   in the current directory"
```

Now we are able to access the GUI description from within our code. This allows us the load the widgets defined in the description, using the `xmlGetWidget` function. When using Glade, one should try to define the widget as it should appear, only using Glade. That way, no adjustments have to be made when loading the widgets in Haskell.

```
window <- xmlGetWidget windowXml castToWindow "window"
   onDelete window deleteEvent
   onDestroy window destroyEvent
controlPanel <- xmlGetWidget windowXml castToVBox "control panel"
entry <- xmlGetWidget windowXml castToEntry "number of pictures"
button <- xmlGetWidget windowXml castToButton "new game button"
label <- xmlGetWidget windowXml castToLabel "message label"
   labelSetText label "\nNothing set."
boardAlignment <- xmlGetWidget windowXml castToAlignment "board content"
board <- xmlGetWidget windowXml castToVBox "cards"
```

As you can see, some extra code is needed besides simply loading the widgets.

Because a window doesn't react on click actions by default, we have to define what has to happen when the user closes the window. E.g. when a window is closed, the delete-event is thrown by Gtk+. For handling this event, the `onDelete` function is used to define which function should be executed when the window is closed.

We also have to define the `deleteEvent` function, which is called whenever a delete-event is thrown. Our implementation is quite simple: there is nothing we need to do when the user closes the window, besides killing the process showing it. To achieve this, we can simply return `False`, which will result in a destroy-event being thrown for the window. This behavior is the same as the default behavior, so if we wouldn't provide this, the window would be destroyed anyway. We mention it here explicitly to illustrate how the system works.

```
deleteEvent :: Event -> IO Bool
deleteEvent _ = do return False
```

In order to catch this destroy-event, we should use the `onDestroy` function (see above), in combination with the definition of the `destroyEvent` function (analogous to the delete-event). The latter just executes `mainQuit` (provided in the General module of Gtk2Hs), which will result in exiting the main loop (and thus killing the application).

```
destroyEvent :: IO()
destroyEvent = do mainQuit
```

The last line of extra code, is quite straightforward. It sets the default text on the messagelabel to "Nothing set.", to show the user the application has just started, and no game level is set yet (which we will need in order to start a new game).

The code: what's after Glade

When all the widgets are loaded, there's still some work to do before we can show the GUI. Because we are building an interactive game application, we should be able to track what the user is doing (i.e. keep track of the game state). Haskell provides a handy 'tool' for that purpose: `IORef`. Because we are building a GUI, most of our code will be executed in the IO monad. Therefore, we can't just pass along some variable and expect every function to notice the change. The IORef module (available in the `Data` package of Haskell), provides a solution for this problem. We can read and write from/to a IORef, and thus share the variable with different functions.

The use of IORef is what distinguishes low/medium level libraries like Gtk2Hs and wxHaskell from high level libraries like FranTk and Fruit. The latter have various abstractions to avoid or hide the use of IORefs.

Throughout the program, we use a single IORef variable to keep track of the state. At the start of the program, we put this IORef in a well known state:

```
state <- newIORef (State Nothing Nothing 0)
```

In the IORef, we use a new data type `State`, that contains all the information we need about the game state, and which is declared as follows:

```
data State = State (Maybe ToggleButton)
                   (Maybe (Bool,ToggleButton, ToggleButton)) Int
```

The first of its arguments may contain a card which is flipped (or nothing when no card is flipped, that's why we use the Maybe monad). The second argument may contain a pair of cards which was tested for a match the last time (or again, nothing when no pair is available). The result of the match test is also available is this argument. The last argument contains the number of pairs of cards left to find, in order to be able to detect when the game is finished (i.e. when no pairs are left to find).

To complete the definition of `main`, we should add the following code:

```
onClicked button ...

widgetShowAll window
mainGUI
```

The `onClicked` part is discussed in the next section. The two last lines of code are needed to show all the widgets in the window, including the window itself (`widgetShowAll window`) and preparing the GUI for user interaction (`mainGUI`).

The code: setting up communication

The only piece of code of the `main` function we have not discussed yet, is the `onClicked` part mentioned earlier. This part defines what should happen when the "*New game*"

button is pressed. Obviously, when there's more than one button in the GUI, the function `onClicked` should be defined for every button.

We first give the entire definition of the `onClicked` function, and then dissect is piecewise.

```
onClicked button $ do
  entryText <- entryGetText entry
  let text  = filter (not.isSpace) entryText
    check = (not $ null text) && null (tail text)  && isDigit (head text)
  if (check) then
    do board <- vBoxNew True 1
       cards <- startNewGame (read text ::  Int) label state
       gameBoard <- createBoard cards state
       boxPackStart board gameBoard PackNatural 0
       children <- containerGetChildren boardAlignment
       if (not $ null children) then containerRemove boardAlignment
                                     (head children)
                        else return()
       containerAdd boardAlignment board
       widgetShowAll window
  else
    do children <- containerGetChildren boardAlignment
        if (not $ null children) then
          containerRemove boardAlignment (head children)
        else return()
        widgetShowAll window
```

Before the level entered by the user is used, it is paramount to check if the text entered in the entry field is correct.

We could avoid this with using a spin box (which would allow only values between 1 and 9) instead of an entry field. This would be much easier, but using an entry field allows us to show some additional aspects of Gtk2Hs (removing widgets from a box, what to do when unexpected user input is given, ...).

First of all, we filter out all the spaces (to avoid unnecessary error messages).

```
entryText <- entryGetText entry
let text  = filter (not.isSpace) entryText
```

To check if the entered text is correct, we use the boolean variable `check`. First of all, we check if the text (without spaces) is 1 character long (because we only allow a level between 1 and 9). Then, we check if the character is a digit, using the `isDigit` function provided in the `Char` module.

```
check = (not $ null text) && null (tail text) && isDigit (head text)
```

Depending on the value of `check`, we decide what to do.

When the text entered is not correct (for example when a word is entered, or some non-numeric symbol), we should clear the game board (because it is possible another game was being played when the new game is started, and the current game should be removed from the board).

```
labelSetText label ("\nPlease enter a level between\n1 and
                     9 to set the game level.")
children <- containerGetChildren boardAlignment
if (not $ null children) then
    containerRemove boardAlignment (head children)
else
    return()
widgetShowAll window
```

To clear the board, we simply remove all the widgets on it. In our implementation, the game board itself is created in a frame (a Gtk widget). On this frame, we have put a box container, in which all the cards are aligned. To clear the game board, it is thus sufficient to remove (and redraw) that box. For obtaining said box, the `containerGetChildren` function is used. This function returns a list of all the widgets that have been added to a container (an argument of the function). As shown above, the box is contained in a frame, and thus the box can be found among the children of the frame alignment. Since removing the box from the alignment actually equates to clearing the board, we must make sure the alignment is not empty before we try to obtain the first element of its children. Obviously, if that case, the code would try to take the head of an empty list, which would result in an abnormal termination of the program, or at least in a runtime error. Finally, to show the change, we add a call to the `widgetShowAll` function, with the entire window as its argument. The new widgets are created in a hidden state, so a function should be called in order to make them visible. We could have called `widgetShowAll` only on new widgets, but to avoid clutter, we chose the implementation shown above.

To inform the user what went wrong, we also show a suitable message on the provided label, using the `labelSetText` function.

When the entered text was correct, we need to show a new game board ready to play the game with the desired number of cards. In order to do this, we have to clear the board (analogous to the case above) and add the new deck of cards.

```
let level = (read text :: Int)
labelSetText label ("\nNew game started (level: "++(show level)++").")
cards <- buildNewGame level state
cardsBox <- fillBox cards state
children <- containerGetChildren boardAlignment
if (not $ null children) then
    containerRemove boardAlignment (head children)
```

```
else
    return()
containerAdd boardAlignment cardsBox
widgetShowAll window
```

To ensure our code remains readable, we implemented the process of showing a new board in several separate functions. First we describe what happens, then we take a closer look at the steps taken to get the desired result.

The game level is entered as text, i.e. a string, and thus must be converted to an integer value. Next to that, we want to show a nice message on the label we provided for this purpose. The `buildNewGame` function yields the requested deck of cards. This deck is then used to build the box, which was mentioned earlier and which will contain the game cards. Building this box is done by the `fillBox` function. It is paramount that we pass along the IORef representing the game state, because this state will be needed by the function that deals with 'card clicks'.

Now, let's look at the definition of the `buildNewGame` and `fillBox` functions.

```
buildNewGame :: Int -> IORef State -> IO Board
buildNewGame n state = do
    writeIORef state (State Nothing Nothing n)
    let imagesPart = take n getImages
        images     = imagesPart ++ imagesPart
    case n of
        1 -> return (Board 2 1 images)
        2 -> return (Board 2 2 images)
 ...
        9 -> return (Board 5 4 images)
```

The `buildNewGame` function returns all the information needed to visualize the game board with a certain level (the number of different card pairs). For that purpose, we defined a `Board` datatype, containing the number of rows and columns of the game board, and a list of the names of the images for every card on the game board.

```
data Board = Board Int Int [String]
```

First, `buildNewGame` sets the game-state to a well-known default value, and builds a list of image-names, as many as needed according to the chosen game level. To keep the example simple, we use a function which returns a static list of image-names.

```
getImages :: [String]
getImages = ["1.jpg","2.jpg","3.jpg","4.jpg",
             "5.jpg","6.jpg","7.jpg","8.jpg"," 9.jpg"]
```

Depending on the chosen game level, the board layout will be chosen (the number of rows/columns).

The other function we need to define, is the `fillBox` function.

```
fillBox :: Board -> IORef State -> IO VBox

fillBox (Board w 1 list) state = do
                                    ...
fillBox (Board w 2 list) state = do
                                    ...

fillBox (Board w 3 list) state = do
    vBox <- vBoxNew True 1
    addHBoxToVBox w vBox (take w list) state
    addHBoxToVBox w vBox (take w (drop w lis t)) state
    addHBoxToVBox w vBox (drop (2*w) list) s tate
    return vBox

fillBox (Board w 4 list) state = do
                                    ...
```

This function will use the list of image names in the `Board` datatype to fill the box that represents the game board. The definition of `fillBox` depends on the number of rows needed to represent the game board. Every board consists of a vertical box containing a number of horizontal boxes of equal width. We show the definition of `fillBox` for a game board with 3 rows. A new vertical box is created, and then the horizontal boxes are added using the `addHBoxToVBox` function. We have to make sure we provide the right sublist containing the image-names.

```
addHBoxToVBox :: Int -> VBox -> [String] -> IORef State -> IO()
addHBoxToVBox w vBox list state = do
    hBox <- hBoxNew True w
    fillRow hBox list state
    boxPackStartDefaults vBox hBox
```

The `addHBoxToVBox` function just creates a new horizontal box of given width `w`, fills it with togglebuttons (using the `fillRow` function), and adds it to the vertical box also provided.

```
fillRow :: HBox -> [String] -> IORef State -> IO()
fillRow box [] state = do return ()
fillRow box (l:ls) state = do
    button <- toggleButtonNew
    widgetSetName button $ l
    containerSetBorderWidth button 2
    image <- imageNewFromFile "back.jpg"
    containerAdd button image
    onToggled button (buttonToggled button state)
    boxPackStartDefaults box button
    fillRow box ls state
```

The `fillRow` function creates a new togglebutton for every string in the given list with image-names. The name of the button is set to the image name (using `widgetSetName`), so when the togglebutton is clicked, we are able to show the image 'hidden' behind it. To start with, a default image is added to the togglebutton. We also define which function should be executed when the button is toggled (`buttonToggled`, see below), and of course the button is added to the horizontal box provided.

The code: playing the game

```
buttonToggled :: ToggleButton -> IORef State -> IO()
buttonToggled button stateRef = do
    state <- readIORef stateRef
    name <- widgetGetName button
    pressed <- toggleButtonGetActive button
    treatClick stateRef name button state pressed
```

The `buttonToggled` function just collects some information about the button which was toggled: the name of the button and the state of the button (pressed/unpressed). We also need the current game state. The actual actions which need to be executed when a button was toggled, are defined in the `handleClick` function.

```
handleClick :: IORef State -> String -> ToggleButton -> State -> Bool -> IO()
```

Because this function has several cases, we will treat them one by one.

```
handleClick _ _ _ (State _ _ (-1)) _ = return ()

handleClick _ _ _ (State Nothing (Just (True,p1,p2)) 0) _ =
  do
    showImageOnButton p1 "found.jpg"
    showImageOnButton p2 "found.jpg"

handleClick _ "found" _ _ _ = return ()
```

In `handleClick`, we distinguish several cases, handled separately by using pattern matching on the arguments of the function. The first case, where the total number of cards left in the game state is set to -1, is used when the game board should not react to any clicks. This is necessary when one button toggle results in 'un-toggling' another button, otherwise the un-toggling would trigger another execution of `handleClick`. When the the number of pairs to match is zero, and the last attempt to match succeeded, the game is finished. Here, we just make sure that the last pair of images are also replaced by smiley faces, but other actions can be added (showing a dialog box, adjusting the message in the control panel, ...). The third case is executed when a button is toggled which contains a card that had already matched. Here, no changes must be made to the game state or the button which was clicked.

```
handleClick ref name button (State (Just lastButton) _ tot) False =
  do
    writeIORef ref (State Nothing Nothing tot)
    showImageOnButton button "back.jpg"
```

When a button is clicked, but the user decides to choose another card to start with, i.e. he clicks the same button again, the state should be set to the state the game was in before to the first click. Obviously, this case must be handled before the others, otherwise a match will be found, which is clearly wrong.

```
handleClick ref name button (State Nothing Nothing tot) _ =
  do
    writeIORef ref (State (Just button) Nothing tot)
    showImageOnButton button name
```

This case occurs when a button is clicked while the game is in the 'empty' state. When this happens, the game is adjusted (it should show which button is currently clicked), and the image which belongs to the clicked button is shown, using the `showImageOnButton` function. This last function is discussed at the end of this section.

```
handleClick ref name button (State Nothing (Just (False,p1,p2)) tot) _ =
  do
    writeIORef ref (State (Just button) Nothing tot)
    showImageOnButton button name
    showImageOnButton p1 "back.jpg"
    showImageOnButton p2 "back.jpg"

handleClick ref name button (State Nothing (Just (True,p1,p2)) tot) _ =
  do
    writeIORef ref (State (Just button) Nothing tot)
    showImageOnButton button name
    showImageOnButton p1 "found.jpg"
    showImageOnButton p2 "found.jpg"
```

When the last attempt to match two images failed, and a new button was clicked, the first case will be executed. The game state is adjusted, so the currently clicked button is in it, and the last attempt to match is removed. The image of the clicked button is shown, and the images of the last two buttons which were clicked are reset to the default image.

When the last attempt did succeed, the same actions will be executed, but instead of resetting the images of the clicked button to the default one, the images are set to a 'found'-image (i.e. a smiley face).

```
handleClick ref name button (State (Just prevBut) _ tot) True =
  do
```

```
    last <- widgetGetName prevBut
    let matched = (name == last)
    writeIORef ref (State Nothing Nothing (-1))
    toggleButtonSetActive False button
    toggleButtonSetActive False prevBut
    prevName <- widgetGetName prevBut
    showImageOnButton button name
    if (matched) then
      do widgetSetName button "found"
         widgetSetName prevBut "found"
         writeIORef ref (State Nothing (Just (matched,button,prevBut))
                         (tot-1))
         if (tot-1 == 0) then
            toggleButtonSetActive True button
         else return ()
    else
        writeIORef ref (State Nothing (Just (matched,button,prevBut))
                        tot)
```

When some button has already been clicked, and a second is clicked, the definition of `handleClick` above is used. Because the toggle of both buttons is set to `False`, we have temporarily adjusted the game state, so no clicks will be accepted. The image of the second button is shown, so the user can see if the match succeeded or failed. Then, using the names of the buttons, we check if the match succeeded. When it did, the name of the button is adjusted to 'found', and the state is adjusted accordingly. When this matched pair was the last pair (which we can check using the number of pairs left to match), we force a button click, to make sure the 'found'-image is shown on both buttons. When the match failed, all we have to do is adjust the game state. Because the name of the buttons wasn't changed, the images will be reset to the default image when the next button is clicked.

To conclude the discussion of the code, we show the function, which is used to show a certain image on a button.

```
showImageOnButton :: Button -> String -> IO()
showImageOnButton button file = do
    children <- containerGetChildren button
    containerRemove button (head children)
    image <- imageNewFromFile file
    containerAdd button image
    widgetShowAll button
```

Because another image is already added to the button, we will remove it first, analogous to the way we removed the game board when a new game is started. Using the

string which contains the path of the new image to be shown, we obtain the desired image, add it to the button, and show the new widgets added to the button using `widgetShowAll`.

The code: playing with efficiency

Because the example application we have written is quite small, efficiency isn't a real issue. Still, we would like to show how to improve the efficiency of our application.

The problem is that each time a button is pushed, the image the button hides is loaded from disk. Because our images are small, no real delay can be noticed. To avoid the images being loaded from disk every time, we could load them once, when the application is started. Then, when a button is pushed, we only have to replace the current image with another image already loaded.

Another thing we can improve, is the way how we change an image shown on a button. In the implementation above, we explicitly remove the current image, and add a new image to the button. A better way would be to change the image, rather than replacing it.

To kill two birds with one stone, we use the `Pixbuf` datatype, which contains all the information needed to create an image.

Loading all the images before they are needed, requires a way to make the loaded images accessible when needed. One way is to use a map, which is built at the beginning of the program:

```
type Images = [(String, Pixbuf)]
```

Now, when we need to 'load' an image, we can use the `lookup` function, which is defined in the Haskell Prelude.

Because we are using `Pixbuf`, there is no need to replace the image on a button, we can just change it. To illustrate, we show how the `showImageOnButton` function could look like:

```
showImageOnButton :: ToggleButton -> Images -> String -> IO ()
showImageOnButton button images imageName = do
    let Just image = lookup imageName images
    imageWidget <- liftM castToImage $ binGetChild button
    imageSetFromPixbuf imageWidget image
```

This implementation is both simpler and cheaper in terms of resources. Credits go to Duncan Coutts for this suggestion.

The game: really playing it

As you may have noticed when trying to play the game, it is quite easy to play, too easy really. The reason is simple: the list containing the image-names, and thus the

list which determines the sequence of the cards on the board, is never shuffled. In order to implement the game as it is meant to be played, we should provide a `shuffle` function, which simply shuffles the list containing the image-names. We didn't bother to implement such a function, because it has nothing to do with the Gtk2Hs functionality.

Conclusion

Gtk2Hs is a powerful, user-friendly Haskell GUI library. The use of Glade to create a GUI allows the developer of the application to concentrate on the interaction part of the application, instead of making sure it looks good (just try to create the same layout as in the memory game, using only Haskell code, you'll see what I mean). The only drawback when using Glade, is that the *.glade file should always be available, which makes distribution of the application more difficult.

The current API available is very useful already, and since Gtk2Hs hasn't reached 1.0 yet, it will only improve. This article could serve as a Gtk2Hs tutorial for people who are not familiar with Gtk2Hs, and hopefully is a stimulation for other people working with Gtk2Hs to write a tutorial of their own. This way, the support for Gtk2Hs will increase, which will stimulate the growth of the library.

As a disclaimer, I would like to state the code isn't meant to be the most efficient code possible, neither to be bug free. Improvements, suggestions and comments are always welcome, and the code is free to use in any way.

I hope this article has convinced the reader of the benefits of using Gtk2Hs as a Haskell GUI library, and has contributed to its popularity. Special thanks to Shae Matijs Erisson (the editor), Duncan Coutts (one of the Gtk2Hs people who made some suggestions) and Andy Georges (who proofread this article, and suggested a lot of improvements, mostly language related).

Implementing Web-Services with the HAIFA Framework

By Simon D. Foster – email: u1sf@dcs.shef.ac.uk

*Distributed Computing is no longer a luxury within the sphere of Computer Science, but rather a necessity. Languages which are not able to interoperate with their peers are unlikely to ever achieve any great role in the future. The purely-functional programming language **Haskell** encapsulates a large and unique feature set, which would seem have many applications in this new world of Web-Services. Indeed, the **HAIFA** project itself was motivated by the need for a more suitable paradigm on which to base the composition of Web-Services to fulfill specific tasks. In this article we look at Web-Service interoperability, and how the **HAIFA** framework can be used to build **Haskell**-driven services, filling a void to which the existing non-functional languages have no elegant and composable solutions.*

Introduction to HAIFA

The **Haskell Application Interoperation Framework (HAIFA)** project is the culmination of the last 18 months work in attempting to enhance the interoperability capabilities of Haskell by introducing a framework for developing and accessing Web-Services. The project has gone through a large number of evolutions as the design was continually refactored to work around various design issues in the original project specification. However, we are now finally reaching the stages where **HAIFA** can be used to build services from vanilla **Haskell** code requiring minimum rewriting, with the ultimate goal of allowing a completely decoupled interface to be attached to packages for distribution of **Haskell** functionality.

The primary motivation behind **HAIFA** has been the need for a more well suited paradigm for developing Web-Services than the current technologies can provide. Our goal with **HAIFA** is to build an elegant and clean Web-Service API, taking advantage of **Haskell's** unique characteristics to enable users to solve instances of a particular problem domain with minimum difficulty. We believe that Haskell provides an ideal platform on which safe, reliable, yet concise and highly composable applications can be built. Composability is becoming an increasingly important factor, as Web-Services are no longer discrete application parts consumed by monolithic application code, but are increasingly subject to direct composition, with minimal 'glue code', and defined

hierarchically by smaller-grain compositions.

However, before we can look at these issues, it is necessary to look at building the tools for achieving basic interoperability. We first examine the components which have so far been developed for **HAIFA**, with an overview of how they work and what they can be used for. Then, we shall look to how **HAIFA** can be used to bind **Haskell** functions to **SOAP** [1] operations in order to produce a full Web-Service. Finally we look at the future of the framework, how we intend to use it in future projects and draw some conclusions.

Components of HAIFA

The Generic XML Serializer

XML and Haskell

The first, and perhaps greatest challenge of the **HAIFA** project has been devising a method for converting **Haskell** data into the primary data serialization language, **XML**. At the beginning of the project, 2 years ago, such tools in **Haskell** were sparse and largely incomplete, providing only the very basic features for parsing and producing **XML**. Due to its then larger feature set, we chose the **Haskell XML Toolbox (HXT)** [2] as the basis library for our project, largely because **HaXML** [3] did not, at that time, provide **XML Namespaces** [4], nor did there appear to be any movement to add such features (although of course things are different now). However, even **HXT** on its own is purely a parser library, with a large set of filters and tools for processing **DTDs**. Due to time constraints, we first simply developed an ad-hoc parser for **SOAP Envelopes**, which was used to aid in communication with **SOAP** services. However, this method was, once more time became available, abandoned in favour of a more generic approach.

The **Generic XML Serializer** (or simply **GXS**) is the corner-stone of **HAIFA**, in that it provides a method of making the use of **XML** virtually transparent to the user. The premise is that users are only bothered about the data at each end of the link, and not at what happens in between, thus **GXS** allow the serialization of **Haskell** algebraic data-types with, where possible, minimal intervention by the user. Of course, if a particular application has particular serialization requirements, **GXS** also provides for this with a completely customizable interface.

GXS has been developed using the latest version of Ralf Lämmels famous '**Scrap Your Boilerplate**' [5][6] Generics library (which we simply refer to as **SYB**), which builds on his type-case methodology with the integration of type-class based cases (see [7]). This is of course precisely the right paradigm in which **XML** should be serialized, with a general case for serializing most algebraic data-types, and possible specialization for other data-types. We also make use of **Template Haskell** [8] (or **TH**) for building derivers for our own **GXS** classes in order to take the maximum amount of work off the user.

Further, **GXS** is highly modular and extensible, with the aim of allowing serialization rules to be distributed over a large number of modules, such that they can be used together to form complex serializers for different tasks. That said, GXS is still very

much in its infancy, the latest version having only been developed in recent months, and much of the deserializer part is currently very naive. However it currently suites our foremost purpose, the implementation of **SOAP/1.1** for Web-Services.

Using GXS

As with all **SYB**-based solutions, **GXS** is based around the `Data` class, which now has two parameters to allow for context customization via John Hughes' method [9]. We use `Data`'s reification properties to enable the extraction of type meta-data in an attempt to automatically build type serializers. For example a data-type with field labels will be serialized to a sequence of elements using the names of the fields (the algorithm for doing this will eventually be customizable).

The class which we 'pass' as context to the `Data` class is our own class, `XMLData` as shown below (for the purposes of this article, we only provide a brief overview)

```
class (Data (DictXMLData h) a) => XMLData h a where
    -- Custom encoder
    xmlEncode :: DynamicMap -> h -> a -> [[XmlFilter]]
    -- Monadic Decoder
    xmlDecode :: h -> ReadX a
    -- Type meta-data
    toXMLConstr :: h -> a -> XMLConstr
```

With `DictXMLData` being our dictionary for this class. The two parameters of the class, `a` and `h`, indicate the type being serialized, and the 'hook' which is being applied to the serialization respectively (we'll consider hooks later). The operations of the two main functions, `xmlEncode` and `xmlDecode` should be relatively obvious, but the third function needs further explanation. The `XMLConstr` data-type contains a number of properties related to serialization of the given type, such as the 'default' name to be assigned to it and how the sub-terms should be serialized (if relevant). The current structure of the `XMLConstr` data-type is shown below[1]

```
data XMLConstr = XMLConstr { xmlFields      :: [FieldProp]
                           , isInterleaved  :: Bool
                           , isMulti        :: Bool
                           , elementNames   :: [String]
                           , attributeNames :: [String]
                           , forceDefault   :: Bool
                           , defaultProp    :: Maybe FieldProp
                           } deriving Show
```

Currently this data-type stores;

[1]Please note that this data-type is currently in a state of flux, and is highly subject to change as the design is refined.

- A list of field descriptors for the sub-terms of the type, specifying how the term should be serialized. For example, whether it should be an element or an attribute (specified by the `FieldProp` data-type).
- Flags for whether the data-type is interleaved (unordered) and whether it usually has multiple particles, as is the case for data-types like `[a]` and `Array`.
- Default element and attribute names.
- A flag indicating if the default serialization method is forced, so for example it will always be given the same name, no matter what its parent type defines.
- The default serialization method.

The `XMLConstr` should, in most instances, provide all the information required to serialize a type. The default definitions of the core functions in `XMLData` use this data-type extensively to serialize a type, which equates to our most general-case. This is a feature which the previous version of **SYB** could not easily provide, since all serialization data had to come from the reified type meta-data, or via cumbersome type-indexed tables which had to be passed to the various functions.

Both the deserialization and serialization functions carry around a `DynamicMap` data-type, which is essentially a `FiniteMap` whose value type is `Dynamic`. However, to enable type homogeneity over the domain each key holds a default value, which must be monomorphically typed, such that the integrity of the data can be maintained. The `DynamicMap` is available for essentially any purpose, and is especially useful for allowing hooks to store and access variables.

This class therefore provides us with most of the features we require to perform serialization. On the one hand, with the aid of the `Data` class and our default methods, a powerful general-case serializer is available, and on the other hand the encoder and decoder functions can be specialized to enable fully customized encoders should the user require them. However, we don't yet deal with the middle ground; suppose we don't want to fully write our serializers, but we do want to alter our `XMLConstr` for a particular type without writing a whole class instance. This is where **Template Haskell** comes in handy. In the most specific case, a simple splice can be built which generates a list of '`instance XMLData h MyDataType`' declarations, to make the code concise. Building on that, we can put together a number of filters which alter the parameters stored in `XMLConstr`, in order to change the default serialization rules with minimal difficulty. For example, the most basic **TH** function, `xmlify`, works like so

```
$(xmlify [''MyDataType1, ''MyDataType2, ''MyDataType3] [])
```

which makes the three given data-types serializable by deriving `Typeable`, `Data` and `XMLData`. We can further supply flags to this function which allow us change the rules for how to serialize these data-types. For example, the standard rule for element names is simply to assign a type's element names corresponding to the names of the type's constructors, which naturally begin with uppercase letters. The actual name we require may begin with a lower-case letter, so with the aid of the flag `decapE`, this change can be effected;

```
$(xmlify [''MyDataType1, ''MyDataType2, ''MyDataType3] [decapE])
```

This may seem a trivial example, but it displays the relative ease with which serialization rules can be customized.

Hooks

So far, only the actual encoding of data has been discussed, but beyond this **GXS** provides a powerful system for encoding meta-data into the serialization tree. Such hooks can be used to encode **XML Schema** [10] type data into the tree, which is a vital feature of **SOAP**, in particular, and Web-Service interoperability in general. A hook is essentially a function which can, at every node in the tree, perform some transformation, such as adding an extra attribute. Currently hooks are only relevant on the encoder side, in that they only allow the production of meta-data and not the interpretation. For the most part this is not a problem, since deserialization can be performed without meta-data, but certainly this is an area for future expansion in order to allow for proper data validation.

Each hook consists of a type-code[2], with instances for the two type-classes; `XMLHook`, which actually encodes the meta-data into the tree and `InitXMLHook` which prepares the `DynamicMap` with any required data.

```
-- Add data to the global DynamicMap
class InitXMLHook a where
    hookDM :: a -> (DynamicMap -> DynamicMap)

{- XMLHook creates a filter for the XML Tree based on
   type-code and the type of the data being serialized.
-}
class InitXMLHook a => XMLHook a b where
    encodeHook :: DynamicMap -> a -> b -> [[XmlFilter]]
```

As an example, a simple encoder for adding **Haskell** type data to encoders might look like this;

```
data XSITypeHook = XSITypeHook

instance Typeable a => XMLHook XSITypeHook a where
    encodeHook dm _ x = let ty = show $ typeOf x in
                            [[attr "type" $ txt ty]]

instance InitXMLHook XSITypeHook where
    hookDM _ = id -- Don't need any extra data
```

[2] A type whose internal structure is irrelevant, since we only use the type to point to class instances.

Summary

GXS is finally reaching the state which we always envisaged; a fully extensible XML Serializer capable of fully customising the serialization process. With the aid of **SYB3**, type-classes can be utilized to fully customize encoders and decoders as required, and hooks aid in encoding arbitrary meta-data into the serialization tree. However, much of this library is still very young and untested, and due to more pressing projects, we have concentrated on the parts of the library needed for **SOAP**. Nevertheless, it should be possible to adapt it to most applications without difficulty.

SOAP/1.1

SOAP/1.1 is a popular, if somewhat deprecated, protocol for the exchange of messages between link-partners, primarily as an aid in interoperability. One of the most common uses of SOAP is to perform remote-procedure invocations over the Internet. The client sends a request message, encapsulating an operation name and some parameters, and the server answers with a reply message of the same form, encapsulating the return values.

The availability of an **XML** Serializer makes the implementation of the basic **SOAP/1.1** Envelope structure a breeze. Due to **Haskell's** parametric types, a very elegant syntax for SOAP has been adapted, currently the Envelope structure looks like this;

```
data Envelope a =
         Envelope{ header        :: [XmlTree]
                 , body          :: Body a
                 , encodingStyle :: Maybe String
                 } deriving (Eq, Show)
```

Which perfectly captures the concept of an envelope encapsulating a message. With the help of **GXS**, we provide encoders for this, as well as appropriate namespaces. For the purposes of this project, we have adapted the hierarchical library structure in order to adopt similar naming to the **Java**-style namespaces, making the location of **URI**-qualified data-types easier to derive. So for example, the **SOAP** Envelope module is qualified as `Org.Xmlsoap.Schemas.Soap.Envelope`. So far, only a basic structure for the **SOAP** Envelope is available, without some of the more detailed features such as Arrays, but these should be relatively trivial to implement with **GXS**.

Web-Service Publisher

The ability to encode and decode **SOAP** Envelopes is only half the story. Once data-types can be encapsulated and serialized, it is necessary to be able to wrap up **Haskell** functions as **SOAP** Invocations. Since we are primarily dealing with functions of the form `InputMessage -> OutputMessage` where both the input and output messages are **GXS** serializable, the process of wrapping the messages up in **SOAP** Envelopes is trivial. The next thing to do is to wrap up the function as a **HTTP** [11] handler. We utilize Warrick Gray's HTTP/1.1 library [12] along with code from **HWS-WP** [13][14], to build a **HTTP**

server shell, which simply takes a series of handlers, which are in reality just functions of the type `MonadIO m => Request -> m Response`, with a few extra parameters for dealing with the configuration. It then uses these handlers to build a **HTTP** server, which we utilize here for serving out **SOAP**.

The **HAIFA Web-Service Publisher** then takes vanilla functions of various types and converts them into **HTTP** handlers which can then be inserted into the server. The publisher is based around the `Publish` type-class;

```
class Service s m where
    publish :: s m -> (XmlTree -> m XmlTrees)
```

consisting of a single function, which takes a type encapsulating some sort of functionality and parameterised over the monad which the web server is working in (i.e. any `MonadIO` monad). This is necessary for monadic operations depending on state, although, of course, not all types need to actually use the monad. As an example, the type for encapsulating the simplest type of function, one from an input message to an output message is;

```
data MonadIO m => SimpleFunc a b m =
                  SimpleFunc { sfunc :: (a -> b) }
```

The instance of `Service` for this data-type simply deserializes the incoming Envelope, applies the encapsulated message to the simple function and serializes the output, after wrapping it in another Envelope. The now homogenous function of type `XmlTree -> m XmlTrees` can easily be converted to a **HTTP** handler and bound to a URI on the server. Finally we wrap up our functions in an existentially quantified type, to enable different types of functions to be inserted as operations, and produce a list of pairs, linking operation names to actual `Request -> m Response` functions.

Putting it all together

In this section we draw all the tools together and show how a Web-Service can be implemented in **Haskell** with the **HAIFA** framework. We also demonstrate a simple example which can be used as a template for putting together more complicated services. The basic process of converting a bunch of functions in a module into a Web-Service is as follows.

1. The module must have the following pragmas
   - `-fglasgow-exts`
   - `-fallow-undecidable-instances`
   - `-fallow-overlapping-instances` (if the default, easy to use rules are required)
   - `-fth` (if using **TH** instance derivers).
2. Import `Text.XML.Serializer`, `Network.Service` and `Network.Server.HTTP` (assuming of course **HAIFA** is installed).

3. All the data-types involved in the function should be made serializable, either by creating `XMLData` instances or by using the **TH** functions, for example `$(xmlify ...)`.
4. Message data-types should be created for the functions; two for each, for input and output. These should carry the parameters of the functions, and will, for much of the time, have field labels, to make deriving serializers easier. The name of the constructors should reflect the names of the messages.
5. Either create instances of `Service` for the function types, which perform the wrapping up as **XML or** wrap the functions up in functions of the form `InputMessage -> OutputMessage`.
6. Pair each function with an appropriate name, and wrap them all up in a list, with the help of the existentially quantified `PubFunc` type.
7. Pass the list of qualified functions to the `buildWS` function, which will produce a single HTTP handler.
8. Create a main function, which runs the **HTTP** server and binds the handler just created to a **URI** on the server.
9. Compile, run the server (making sure an appropriate config file exists) and the Web-Server is ready for action!

Appendix demonstrates how a sample Factorial Web-Service can be constructed. For simplicity, a function called `factorial` has been created which takes the input message, which encapsulates an `Int`, and returns the output message which encapsulates another `Int`. Both `InputMessage` and `OutputMessage` are then made serializable with `xmlifyQ`, a variant of `xmlify` which also namespace qualifies the data-type. The function `decapE` decapitalises the first character of the message names, given by the constructor names. This function is then wrapped up as described above, passed to the `buildWS` function and finally to the **HTTP** server itself in the `factorialService` function. This then gives us a simple Web-Service with a simple function which can be invoked remotely via **SOAP**.

Future Components

XML Schema

The ability to be able to correctly type **XML** literals is an essential part of programming **XML** based applications. The next large project in **HAIFA** will be the development of an **XML Schema** type-mapper, although the first task will be to produce a set of suitable of data-types for parsing a schema. We have had some success in parsing **XML Schema**, specifically with the older version of **GXS** (pre-**SYB3**). We were able to parse the complete **XML Schema** syntax, and built a very basic type-mapper for it. The conversion of **Haskell** data-types to **XML Schema** will actually not be particularly complicated, since most of the data can be gleaned from the `XMLData` class in **GXS**. However, mapping schema data-types to **Haskell** will be a much greater challenge.

WSDL

Once **XML Schema** is developed, or at least partially developed, the next stage will be to put together a **WSDL** [15] processor, which can produce a description of a **Haskell** Web-Service, and create accessors for existing Web-Services. To be able to describe the interface of a **Haskell**-driven Web-Service is of paramount importance, since without such a description it is very difficult for other programs to communicate with it. As with **XML Schema** though, actually generating **WSDL** will not be too hard, since we can set the limits on how we encode our types and operations.

Composite Web-services

As stated the primary motivation behind **HAIFA** was not merely the implementation of Web-Services in **Haskell**, since arguably there already exist adequate solutions in other languages for doing this. To be able to realise solutions to realistic problems, it is unusual that a single Web-Service can provide all the required facilities. In fact with different vendors providing different services, such as Banks, Travel Companies and General Stores, it is important to consider how a number of Web-Services can be combined to form solutions to more demanding questions. Functional programming is naturally very well suited to solving composition problems, and we believe that **Haskell** could play a very large role in providing the semantics for workflow-based orchestrations. At the time of writing the **CASheW-s** project [16] at the University of Sheffield is beginning to fully realise a workflow engine, which will, when completed, be capable of orchestrating composite Web-Services.

Conclusion

We have looked at the work on the **HAIFA** framework so far in the areas of **XML**, **SOAP** and Web-Services and demonstrated a clear need for the implementation of such technologies in **Haskell**. Further, we demonstrate a much larger, and still somewhat open problem of compositionality in Web-Services, which we think clearly demonstrates that the current movement in the world of distributed computing and component technology neccesitate the use of the functional programming paradigm. All of the library components are still very immature, and do not yet provide all the features required for the more advanced **XML** technologies, in particular **XML Schema**. We also acknowledge that there are still many shortcomings in **GXS**, particularly in deserialization. Nevertheless, we believe the **HAIFA** framework encapsulates a significant step forward in the use of **Haskell** for building industrial strength interoperability applications.

The HAIFA project page on Savannah can be found at
`http://savannah.nongnu.org/projects/haifa`
Darcs repositories for HAIFA and its dependencies can be found at

`http://www.dcs.shef.ac.uk/~u1sf/darcs`[3]
HAIFA code available under the terms of the GNU General Public License

Acknowledgments

We would like to thank Matt Fairtlough, Barry Norton and Andrew Hughes, for their suggestions and support in this work. We would also like to especially thank Ralf Lämmel, for putting together the latest and greatest version of SYB, without which this project would not have been possible. A large section of this work was conducted as part of a third-year dissertation at the University of Sheffield.

References

[1] Martin Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (SOAP) 1.1. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/ (2000).

[2] Uwe Schmidt. The Haskell XML Toolbox Website. http://www.fh-wedel.de/~si/HXmlToolbox.

[3] Malcolm Wallace and Graham Klyne. HaXml: Haskell and XML. http://www.cs.york.ac.uk/fp/HaXml/.

[4] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML. http://www.w3.org/TR/REC-xml-names/ (1999).

[5] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. **ACM SIGPLAN Notices**, 38(3):pages 26–37 (mar 2003). Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[6] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In **Proceedings; International Conference on Functional Programming (ICFP 2004)**. ACM Press (sep 2004). 12 pages; To appear.

[7] Ralf Lämmel. Modular generic function customisation. http://homepages.cwi.nl/ ralf/syb3/ (2004).

[8] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty (editor), **ACM SIGPLAN Haskell Workshop 02**, pages 1–16. ACM Press (October 2002).

[9] John Hughes. Restricted Data-types in Haskell (1999).

[10] W3C XML Schema. http://www.w3.org/XML/Schema.

[3]The version of SYB3 found in these repositories should neither be considered official nor final, and is only provided as a convenience.

[11] R. Fielding, UC Irvine, and J. Gettys. Hypertext Transfer Protocol – HTTP/1.1. Technical report (1999).

[12] Warrick Gray and Bjorn Bringert. Haskell HTTP Library. http://www.dtek.chalmers.se/ d00bring/haskell-xml-rpc/http.html.

[13] Simon Marlow. Writing high-performance server applications in haskell, case study: A haskell web server. In **Haskell Workshop**. Montreal, Canada (September 2000).

[14] Martin SjÃ¶gren. Dynamic loading and web servers in haskell (October 2000).

[15] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Service Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl (2001).

[16] CASheW-s Engine Project. http://savannah.nongnu.org/projects/CASheW-s-engine.

Listing of Factorial Web-Service

```
{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances
            -fallow-overlapping-instances -fth #-}
-- A Sample Factorial Service.
module Network.Service.Factorial where

import Text.XML.Serializer
import Network.Service
import Network.Server.HTTP
import Network.HTTP
import Network.URI
import Data.Generics2
import Text.XML.HXT.Aliases
import Text.XML.HXT.Parser
import Org.Xmlsoap.Schemas.Soap.Envelope

-- The Input Message
data IntMessage = IntMessage {value::Int} deriving Show

-- The Output Message
data FactMessage = FactMessage {fact::Int}

-- The Factorial Function
factorial :: IntMessage -> FactMessage
factorial (IntMessage i) = FactMessage (f i)
    where f 0 = 1
          f 1 = 1
          f n = n * f (n-1)
```

```
-- Make Input and Output message serializable
$(xmlifyQ [''IntMessage, ''FactMessage]
          [decapE] "urn:FactorialService")

factorialService = ("factorialService",
   \x -> \y -> buildWS  [("intMessage",
               PubFunc $ SimpleFunc factorial)])


-- Run the HTTP Server
runFactorial :: IO ()
runFactorial = httpServer "config.xml" [factorialService] []
```

Code Probe - Issue one: Haskell XML-RPC, v.2004-06-17 [1]

By Sven Moritz Hallberg – email: pesco@gmx.de

Greetings, Recipient! Welcome to Code Probe, The Monad.Reader*'s code critique column. My name is Sven Moritz, also known as Pesco on the IRC, and I hope to regularly use this column to review some interesting Haskell programs from a* ***literary*** *perspective.*

Being a believer in Knuth's famous saying that

> *programs are meant to be read by humans and only incidentally for computers to execute*

I think that there should be a culture of programming criticism, much like there is a culture of literary criticism. This column is my contribution to such a culture.

I owe inspiration to a talk by Bogk et al. [2] who held a public code critique session and, in essence, suggested:

> *Read some code from time to time. It's fun. And you never know what it turns up...*

Nota bene, among other things they displayed, to their audience of several hundred programmers, a rather blatant buffer overflow condition in the source code of a certain ***very*** *popular database system. Funny feeling...*

XML-RPC

Everyone has probably at least heard about XML-RPC [3]. Used on the WWW for remote-controlling web services such as Google, it is a simple XML format for encoding a remote procedure call (and the reply). In fact, it is **extremely** simple. A procedure call consists of a method name (a character string) and some number of arguments. The result is a single value. For the argument and result values, there are six primitive data types (`int`, `boolean`, etc.) and two kinds of complex data types (`array` and `struct`). Without going into detail, the "arrays" are actually heterogenous lists, and a `struct` is a kind of associative list, mapping field names (strings) to field values.

Haskell XML-RPC

Björn Bringert has written a pretty complete implementation, for both making as well as accepting XML-RPC calls. The interface is very easy to use, there is virtually no boilerplate code needed for simple cases. The provided CGI servant also supports the de-facto standard introspection methods automatically.

Literate Programming, almost

When I first glanced into Haskell XML-RPC as a candidate for Code Probe one, I thought it was a literate program that just did not make a big deal about being one. I was looking at the "report" [4] which I would describe as the library's user manual. It includes, in pretty LaTeX typesetting, complete definitions of Haskell functions and data types used by the library.

Looking closer though, the report only quotes these from the actual source code. It does so **almost** completely and **almost** accurately. The single inaccuracy is that the report only mentions the old, non-hierarchical module names which means that a user reading the report only (for its good explanations) will be surprised to find his program incorrect with the current release. As for incompletness, there are one or two recent developments in the code which are not (yet) mentioned in the report. Apart from that, it explains very well how the library works and how to use it. It is also not overly verbose. Good!

Short Reference

In addition to the report, a Haddock[5]-generated API reference is available on the Web which, by the way, does mention the up-to-date module names. It is complete although a bit short on details.

The latter, in my experience, seems to be a common effect especially with auto-generated reference documentation. A reference should provide an already-experienced user with full details on the subject. I **suspect** that programmers hesitate to "clutter" their program source with these details (just look at some man-pages to see how much there can be to say).

For concrete example, the Haddock entry for the type `XmlRpcMethod` is "The type of XML-RPC methods on the server" which tells me next to nothing. Looking at the definition I can partly guess at its purpose but only discovering its usage in `cgiXmlRpcServer` really made it clear to me.

Any way, the reference documentation only really neglects very few entries, all important functions have documentation, and it is sufficient. Generally, not bad at all.

One omission that surprised me, though, was the missing documentation for the classes `Remote` and `XmlRpcFun`. They are used to implement the polyvariadic functions `remote` and `fun`. The trick used by Bringert here was to my knowledge first mentioned by Oleg Kiselyov on the Haskell mailing list [6]. It is not trivial to grasp at first (basically, a

polyvariadic functions is a method of a classes with a recursive instance declaration) so documentation would be helpful. The report does contain a short explanation of the technique so my guess is that Haddock just runs into a loop on the recursive instances. Pity.

Low-Level Structs

Although the basic interface of the library is **very** straight-forward and clean, there are some minor limitations. Most notably `struct`s require low-level handling unless all their fields are of the same type – they are mapped to the Haskell type `[(String,a)]`.

Since recently, there is a Template Haskell module (called `THDeriveXmlRpcType`) to generically map `struct`s to appropriate Haskell records. It is neither mentioned in the on-line API reference, nor in the report but the code does contain Haddock comments and should be readable to those familiar with TH.

Meat of the Matter

Having read all that documentation, is the code itself readable? Yes it is. While it does not seem to be written with specifically literate ambitions, it **is** well-structured and easy to understand.

One could complain about one or two seemingly superfluous definitions (`post_` and `doCall`) or suboptimal function names (`handleResponse`), but those are hardly distracting.

More importantly, the internal comments quote the XML specification in several places, making nicely clear what the code is supposed to do. Also, identifiers are generally readable and meaningful, and last but not least, all functions are no longer than a couple of lines.

Also worth noting is the fact that the library relies on external packages for dealing with XML and HTTP. Thus, there is no extraneous clutter or interweaving of the RPC facility with the encoding and transport code.

Conclusion

In summary, the Haskell XML-RPC library, apart from bringing some very useful functionality to Haskell, was a pleasant read which might even teach some readers a new trick – polyvariadic functions. In addition, even though the user manual is **very slightly** outdated and the API reference could use some more detail, the provided documentation classifies as “very good”.

Probe Rating: Quite Enjoyable

References

[1] Björn Bringert. Haskell xml-rpc (June 2004). http://www.dtek.chalmers.se/~d00bring/haskell-xml-rpc/.

[2] Andreas Bogk **et al.** Das literarische code-quartett (December 2004). http://www.ccc.de/congress/2004/fahrplan/event/97.en.html.

[3] D. Winer. Xml-rpc specification (June 1999). http://www.xmlrpc.com/spec/.

[4] Björn Bringert. Haskell xml-rpc "report" (January 2004). http://www.dtek.chalmers.se/~d00bring/haskell-xml-rpc/haskell-xml-rpc.pdf.

[5] Simon Marlow. Haddock: A haskell documentation tool. http://haskell.org/haddock/.

[6] Oleg Kiselyov. Functions with the variable number of (variously typed) arguments (June 2004). http://okmij.org/ftp/Haskell/vararg-fn.lhs.