The Monad.Reader Issue 12: Summer of Code Special

by Max Bolingbroke ⟨batterseapower@hotmail.com⟩
and Roman Cheplyaka ⟨roma@ro-che.info⟩
and Neil Mitchell ⟨ndmitchell@gmail.com⟩

November 19, 2008

Wouter Swierstra, editor.

Contents

Editorial

by Wouter Swierstra ⟨wss@cs.nott.ac.uk⟩

I'm happy to announce this year's Summer of Code Special. Three Summer-of-Coders sponsored by haskell.org wrote an article about their projects: Max Bolingbroke gives a tour of GHC's new plugin support; Roman Cheplyaka describes the implementation of a physics engine using Data Parallel Haskell; and Neil Mitchell discusses the past, present and future of Hoogle.

As an editor, it is my responsibility to say a few words about the other projects. Despite my best efforts last year, there were four Summer-of-Coders that didn't write an article for this issue. To make this editorial somewhat more interesting for you to read and for me to write, here's a limerick about each of the other projects:

GHC API Improvements

Student: Thomas Schilling

There was once a hacker called Schilling,
Who found the GHC-API unfullfilling,
On #haskell we know
Him as nominolo,
And his code is absolutely thrilling.

A 'make-like' dependency framework for Cabal

Student Andrea Vezzosi

As some of you might recall,
Packages were once a pain to install.
But thanks to Duncan Coutts,
And the Haskell grass roots,
We now have the Haskell Cabal.

Language.C, a standalone parser/pretty printer library for C99

Student: Benedict Huber

There's a package called Language.C,
Which is quite useful you see,
It takes a long String,
and then does its thing,
and returns a C AST.

Efficient maps using generalised tries

Student: James Brandon

This last limerick's quite hard for me,
If you read on, I'm sure you'll agree.
I struggle and sigh,
For no matter how I trie,
I just don't know how to pronounce trie.

Compiler Development Made Easy

by Max Bolingbroke ⟨batterseapower@hotmail.com⟩

Have you ever wished that you could make Haskell a strict language? Or maybe you've had a great idea for an optimization you could apply to your combinator library that rewrite rules are just not powerful enough to express? Perhaps you're an academic and would love to be able to perform some analysis on the execution of Haskell code? All these ambitions and more can be realised through a new capability of the Glasgow Haskell Compiler [1]: compiler plugins.

The how and what of plugins

To understand how we can accomplish all these things, I first need to talk a little about how GHC compiles Haskell. It is essentially a four step process:

1. Incoming Haskell code is parsed into an abstract syntax tree. This tree captures the full glory of the Haskell language, and is correspondingly complicated: the main data type used to represent this tree has no less than 44 constructors!

2. The code undergoes some processing by the **front end** of the compiler. First the tree undergoes **renaming** to resolve name scoping, then type checking is performed, and GHC finishes off with a process known as desugaring. This desugaring step converts the complicated and large Haskell abstract syntax tree into the much less scary Core language tree. This will be expounded on later.

3. The resulting Core code is transformed by a number of Core-to-Core passes. These are all designed as meaning-preserving transformations that somehow optimize the code they get as input: for example, there is a pass for constructor specialization [2], one for common-subexpression elimination and so on. The series of Core passes executed here are collectively known as the **middle end** or **Core-to-Core pipeline** of GHC.

4. The final Core code is consumed by the back end of the compiler, which deals with generating the actual assembly code that will be run to execute that program. We won't have to worry about the magic that goes on here for the rest of this article.

The compiler plugins feature gives you access to step 3 of this process: you can have your own Core-to-Core passes dynamically loaded into GHC and installed somewhere in the Core pipeline. There are certain limitations to choosing this as the extension point: for example, you would be ill-advised to attempt to generate huge chunks of new code out of nothing as part of your Core pass (please use Template Haskell [3] instead), and it is simply impossible to extend the type checker or the syntax of Haskell. What you can do, broadly, is **optimize**, **analyze** and **check** this Core representation. This article is going to focus on exploring an optimization of sorts, but hopefully this will give you an idea of the sheer power afforded you by these three operations.

A plugged-in Hello World

Let's write the simplest possible plugin to get a feel for how they work: the full code is shown in Figure 1. I think you'll agree that it really couldn't have been easier than that! All it contains is a declaration for the value *plugin*, which is the special name we must give to the plugin that lives in a module for GHC to pick it up. I've assigned this to the default plugin, which is just a plugin that does nothing at all.

```
module SimplePlugin (plugin) where
import GHCPlugins
plugin = defaultPlugin
```

Figure 1: The simplest possible plugin

Since we'd quite like to have confirmation that something is alive, replace the *plugin* declaration with this marginally more interesting one:

```
plugin = defaultPlugin {
  getPasses = putMsgS "Hello, GHC World!" >> return []
  }
```

Right. Let's see it in action!

```
$ ghc -c SimplePlugin.hs -package ghc
$ printf "module Main where\nmain = ...
        putStrLn \"Hello, Runtime World!\"" > TestProgram.hs
$ ghc TestProgram.hs -plg SimplePlugin -o TestProgram
Loading package ghc-prim ... linking ... done.
... lots of "Loading" ...
Loading package template-haskell ... linking ... done.
Hello, GHC World!
$ ./TestProgram
Hello, Runtime World!
```

As you can see, I've used the new *plg* GHC flag to tell it to load the *SimplePlugin* as part of the build process. This is the only new compiler flag you should need to worry about if you're just a consumer of someone else's plugins, but what we've seen so far isn't going to let you produce a wildly popular plugin: let's delve deeper.

Plugins for real: Strict Haskell

Laziness getting you down? Mired in space leaks? Want to give the compiler the opportunity to unbox your program to the hilt? Then the example plugin we're going to develop here is just the thing for you: we're going to change Haskell's evaluation strategy from lazy to strict, all with a simple Core plugin.

Before we go on, let's talk about what this Core language actually looks like. The theoretical basis of Core is a typed lambda calculus known as System FC [4]. GHC's implementation of this System FC as a Haskell data type has just 9 constructors, compared to the 44 of fully-fledged Haskell, which means that you as a compiler plugin author have to handle only a few cases to get the job done.

The full set of data types that make up Core are reproduced in the appendix (Figure 5), but I'm going to try and give you a gentle introduction to the language by means of examples.

The harmony of types and values

The first thing you need to know about Core is that it is a typed lambda calculus: this means, among other things, that it has both the normal "value" lambdas you are used to in Haskell (which might be written as λ) and the more exotic "type" lambdas (which are traditionally written as Λ). Type lambdas capture the notion of parametric polymorphism: wherever you see a *forall* in the type of some function, there will assuredly be a type lambda in the corresponding Core representation of that function as a value, ready to gobble up the actual type you want to use that function at.

As a concrete example, consider the identity function. In Haskell, we can write a rather silly implementation of that as follows:

$$\begin{aligned}&\mathit{my\_id} :: a \rightarrow a\\&\mathit{my\_id} = \lambda x \rightarrow \mathit{id}\ x\end{aligned}$$

However, as you probably know, Haskell implicitly quantifies over all the free type variables (such as a) that we may have written in our types. So the above definition is really the same as this slightly more explicit one:

$$\begin{aligned}&\mathit{my\_id} :: \mathit{forall}\ a.a \rightarrow a\\&\mathit{my\_id} = \lambda x \rightarrow \mathit{id}\ x\end{aligned}$$

As we're still working in Haskell, this code only contains a value lambda: that which captures the variable x, and a value application: that of x to id. If we take a look at the Core representation of *my_id* that we obtain after a desugaring step, we'll see a different story:

$$\mathit{my\_id} = \Lambda\ a.\lambda(x :: a) \rightarrow \mathit{id}\ a\ x$$

Figure 2: The Core language representation of our silly identity function

Suddenly, both a type abstraction (for the type variable a) and an application (of that type variable to the Core representation of id) have sprung up. I've also added an explicit type signature to the binder x to make it clear that, crucially, **the type of x is bound by the type lambda**. The outermost type lambda tells you that in order to use this function you have to first supply a type you wish to use the function at. Apologies if you feel I've belabored the point, but this is the single biggest conceptual change you need to make to get started with the Core language from a Haskell programming background.

If you wish to learn more about the remarkable theoretical properties of System FC and it's relatives, a place to start would be the Wikipedia page on System F [5]. System F is a subset of the full System FC we will work with in GHC, but it turns out that for most Core pass authorship you need only a passing familiarity with the extensions to F provided by FC, which are designed to give the ability to deal with GADTs and other type exotica in a principled way.

Returning to the actual Core data types, we might ask how we would represent the fragment of Core in Figure 2 concretely. The answer comes in two parts: first, a value of type *CoreExpr* that represents the definition of the function:

$$\begin{aligned}
&\mathit{my\_id\_rhs} = \mathit{Lam}\ \mathit{a\_bndr}\ (\mathit{Lam}\ \mathit{x\_bndr}\\
&\quad (\mathit{App}\ (\mathit{App}\ (\mathit{Var}\ \mathit{id\_bndr})\\
&\qquad (\mathit{Type}\ (\mathit{mkTyVarTy}\ \mathit{a\_bndr})))\\
&\qquad (\mathit{Var}\ \mathit{x\_bndr})))
\end{aligned}$$

In this definition, I've assumed we have to hand a number of *_bndr* values of type *Id*. You can think of the *Id* datatype as essentially just consisting of a unique number which is used for comparing *Id*s, a human-readable name and the type of the value it binds: it represents the essence of a (type or value) variable.

I've also made use of the auxiliary function *mkTyVarTy* to lift a type variable (of type *Id*) into an actual type suitable for application (of type *Type*). We won't concern ourselves with the exact representation of types in this article, so will be content not to look at the details of what *mkTyVarTy* is doing.

Although there are really these two types of lambdas floating around, you can see from the above that there is actually only one *Lam* and *App* constructor. This is because *Lam* and *App* are overloaded to mean both type **and** value application. If your plugin needs to tell the difference, you need to either inspect whether the binder is a type or value binder (for *Lam*) or see if the argument of the application is a *Type* (for *App*).

The second part of our rendering into concrete Core is a value of type *CoreBind* which gives that right-hand-side a name (note that a program is just a list of *CoreBind*s!). In our case our binding will look something like this:

$$\mathit{my\_id\_binding} = \mathit{NonRec}\ \mathit{my\_id\_bndr}\ \mathit{my\_id\_rhs}$$

See that use of *NonRec*? In essence it indicates that *my_id* doesn't refer to itself in it's own right-hand-side, and hence is non-recursive. However, it's a bit more than that: the right-hand-side of *my_id* must also not refer to any other value that mentions *my_id* in **it's** right-hand-side, which is the sort of situation that arises with mutually recursive functions:

$$\begin{aligned}
&\mathit{isEven}\ 0 = \mathit{True}\\
&\mathit{isEven}\ (n+1) = \mathit{isOdd}\ n\\
&\mathit{isOdd}\ 0 = \mathit{False}\\
&\mathit{isOdd}\ (n+1) = \mathit{isEven}\ n
\end{aligned}$$

The functions in the this mutually-recursive example would have to be represented in the Core data type as a recursive binding group, like this:

$$\begin{aligned}
&\mathit{isEven\_isOdd\_bindings} = \mathit{Rec}\ [(\mathit{isEven\_bndr}, \mathit{isEven\_rhs}),\\
&\quad (\mathit{isOdd\_bndr}, \mathit{isOdd\_rhs})]\\
&\mathit{isEven\_rhs}, \mathit{isOdd\_rhs} :: \mathit{CoreExpr} \quad \text{-- Omitted}
\end{aligned}$$

To sum it up, the difference between *Rec* and *NonRec* binding is that the variable being bound **is not** in scope in the right hand side of a non-recursive binding, whereas all the identifiers in a recursively bound group **are** in scope in all the right hand sides of that group. (Actually, there's another difference, though it's not important for our discussion: *NonRec* bindings can bind types as well as values, whereas *Rec* cannot, since infinite types are frowned upon!)

As an aside, it's interesting to note that it is always safe to bring something into scope with a mutually recursive *Rec* binding (if we assume variable names are all unique), but it impedes optimization of the resulting Core code. As a result, a part of GHC called the simplifier constantly tries to reduce *Rec* bindings into simple *NonRec* ones if possible. We will discuss the simplifier more later on, as it has an important role in the Core pipeline.

Before I continue with the discussion of how all this relates to Strict Haskell, let's briefly discuss the meaning of three of the Core datatype constructors that won't feature in a major way in this article but which I include for completeness: *Lit*, *Cast* and *Note*.

Remember those System FC extensions to System F that I mentioned? Well, they essentially take the form of the additional *Expr* constructor *Cast*. All this says is that the expression it contains should be coerced according to it's *Coercion*. Like all operations on types, this information is erased at runtime on the understanding that only valid casts have been inserted by the desugaring and Core pipeline stages and hence execution will not "go wrong" despite the lack of dynamic checks.

The *Lit* constructor simply allows primitive values with unboxed types such as *Char#* and *Int#* to appear in the expression language, which are essential but boring features of Core. Finally, the *Note* constructor is used to attach some extra information to arbitrary points in a Core expression, such as which cost center [6] it's execution should be counted against.

As I mentioned, we won't have to worry about these alternatives in this article, though you'll need to understand them if you come to write a plugin yourself. Let's move on to discuss the final two Core constructors, which are the really interesting ones.

The harmony of suspension and evaluation

The last two constructors, *Let* and *Case*, are some of the more powerful constructs of Core. What they have in common is that they allow the binding of new names over other Core expressions, but the way they go about it is rather different.

The *Let* constructor of Core takes one of the binding groups we've already discussed (*CoreBind*s) and binds it over another expression in the obvious fashion. A simple example might be the following code, which adds the result of a call to a function *foo* (with an unboxed argument type) to itself:

```
let x = foo 10 #
in x + x
```

This just turns into the following bit of *CoreExpr*:

```
Let (NonRec x_bndr (App (Var foo_bndr)
                        (Lit (mkMachInt 10))))
    (App (App (Var plusInt) (Var x_bndr))
         (Var x_bndr))
```

The final Core constructor is *Case*. This is rather more involved than *Let*, because as well as binding a value it also performs a control flow action, selecting one of the case alternatives. Accordingly, the expression *Case e b t alts*, which represents a case statement, is read as the rather complicated statement "bind the expression e to the binder b, then do a case split on the outermost constructor of e in order to descend into the right-hand-side of the pattern matching alternatives *alts* (which all have type t) which corresponds to that constructor".

I appreciate that this is quite a mouthful, so let's look at a concrete example incorporating a *Case* construct. Consider the following, rather contrived, Core language expression:

```
case (Just 10#) of wild
  (Just val) → val
  _ → 20 #
```

You may notice that the syntax of a Core **case** construct differs somewhat that of Haskell, as it has an extra variable (called *wild* in our example, and b in our explanation of **case** above) after the **of** keyword. This is bound to the scrutinee (the expression scrutinised by the *Case*) and is included in the language because it turns out to make various optimizations easier to express.

If we were to render this fragment of Core into a *CoreExpr* syntax tree we obtain a value as follows :

```
Case (App (Var $ dataConWrapId just_data_con)
  (Lit $ mkMachInt 10))
  wild_bndr
  intPrimTy
  [(DEFAULT, [], Lit $ mkMachInt 20)
  , (DataAlt just_data_con, [val_bndr], Var val_bndr)]
```

Where the free variables *just_data_con*, *wild_bndr* and *val_bndr* (which are not provided directly by GHC) have been appropriately set up.

One thing you may want to note is that the order of the alternatives in the list of the *Case* does not correspond to the order in which they are matched: actually, they are ordered according to a canonical order on the data constructor being matched by each alternative. This is because in Core a **case** expression cannot perform a nested pattern match, which makes the order in which matches are performed irrelevant.

For the purposes of this article, the most crucial difference between *Let* and *Case* is that when we compile a *Let* GHC generates code to create a **thunk** – a suspended computation – which may be evaluated by other code at a later time. This differs from a *Case* expression, which compiles into code which directly evaluates the expression being scrutinised just as far as it's outermost data constructor (into the so-called "weak head normal form"). The thunks are exactly what give Haskell it's non-strict evaluation strategy, so if we were to systematically replace *Let* with *Case* in the entire program we would just end up with a strict version of the input Haskell program! This is what we're going to explore in the next section of this article.

In which we junk the thunks

Now we're all experts in the facilities provided to us by GHC's Core language, let's put that knowledge to work in our new plugin. We begin by writing a small harness whose purpose is to inject our "strictifying" Core pass into the compiler pipeline: this is shown in Figure 3.

One thing you may not be familiar with in this code is the *PHASE* pragma. This serves to introduce a new compiler phase, and every Core pass must be associated with exactly one of those Core phases so the compiler can work out when in the Core pipeline it should be applied to the program being compiled. The constraints I've put on the *StrictificationPhase* in this example just ensure that it runs quite early in the pipeline, so we can get maximum benefit from any optimizations later in the pipeline. We rather hope that these optimizations will be able to use the new-found strictness of the program as modified by our pass to optimize much more aggressively than they would otherwise.

For further information on the use of the new *PHASE* pragma, which also has applications in the specification of rules and inlining, please see the GHC users guide [7].

Another interesting feature of this code is the reference to a *defaultGentleSimplPass*, which is an instruction to the compiler to invoke the "simplifier" I alluded to earlier. The simplifier is a built-in Core pass that performs many janitorial operations on the Core syntax tree, such as β-reduction, inlining, rule application, and reducing the amount of variable shadowing in the syntax tree. In this instance we are using a "gentle" invocation that instructs the simplifier to

```
{-# LANGUAGE TemplateHaskell #-}
module StrictifyPlugin (plugin) where
import GHCPlugins
import GHC.Prim ( {-# PHASE InitialSimplification #-} )
{-# PHASE StrictificationPhase > InitialSimplification, < 2 #-}
plugin :: Plugin
plugin = defaultPlugin {
  getPasses = do
    Just pass_nm ← thNameToGhcNameM StrictificationPhase
    return [CoreToDo pass_nm
      (CoreDoPasses
        [defaultGentleSimplPass
        , CoreDoPluginPass (BindsToBindsPluginPass
          "Strictify" pass)
        ])
      ]
  }
pass :: [CoreBind] → CoreM [CoreBind]
pass = error "Read the rest of the article!"
```

Figure 3: The basis for the strictifying plugin

do a minimal amount of optimization, but which does make it remove any spurious recursive lets which are binding non-recursive values. The motivation behind this is that the presence of these would impede the applicability of our strictifying pass, as I will discuss shortly.

As you can see, it's now up to us to write a function of type $[\mathit{CoreBind}] \rightarrow \mathit{CoreM}\ [\mathit{CoreBind}]$ that actually performs the strictification. The strictification process essentially requires us to replace every applicable *Let* with a *Case*, and I've chosen to implement this as a generic bottom-to-top traversal over the Core syntax tree. This means we need to write a function that takes an expression whose subcomponents have been strictified and strictifies that new expression.

One wrinkle we need to take account of when designing this function is that since:

$$(e1\ e2) = (\textbf{let}\ x\ =\ e2\ \textbf{in}\ e1\ x)\ \text{[for fresh } x\text{]}$$

We also need to be careful to translate the arguments of applications. Indeed, GHC will transform Core into a slightly different "STG" representation before sending it off to code generation, and one thing done by that transformation is to turn all complex applications (i.e. application of arguments that are not just vanilla variables) into explicit **let**s to make it clear thunks are being introduced. However, as plugins work at the Core level we will have to deal with the fact that thunks may be implicitly introduced by an application for ourselves.

Furthermore, if we went around trying to strictify recursive bindings by using *Case* in some manner we would quickly find one of two things. Firstly, we might find that it is a function binding, in which case trying to evaluate them with *Case* is fruitless as functions are already values. Alternatively, we might find that they weren't functions but rather values defined using value recursion (e.g. $\mathit{ones} = 1 : \mathit{ones}$) – and making such things strict can introduce non-termination!

The resulting pass is presented, with inline commentary, in Figure 4. This is ready for use during compilation in just the same manner as with the simple "hello world" plugin I demonstrated in the previous section.

Discriminating plugins use annotations

In the last section we lovingly crafted a compiler plugin that took whatever module it was compiled with, and made it all strict. Often, you would want your plugins to be a bit more selective, and only apply to a few carefully chosen values. You may even want to attach some extra information to those values to guide the transformation or analysis that the plugin performs: imagine, for example, annotating some functions with preconditions and postconditions in the style of Xu [8] which are then checked by a theorem proving plugin.

```
 {-# LANGUAGE PatternGuards #-}
import Data.Generics
import Data.Maybe
  -- Applies the generic traversal to the entire program,
  -- strictifying it
pass :: [CoreBind] → CoreM [CoreBind]
pass binds = everywhereM (mkM strictifyExpr) binds
  -- The core generic traversal that strictifies
  -- an expression given that its subexpressions
  -- have been strictified already
strictifyExpr :: CoreExpr → CoreM CoreExpr
strictifyExpr e@(Let (NonRec b e1) e2)
   | Type _ ← e1 = return e   -- This can occur if the Let
        -- is binding a type variable
   | otherwise = return (Case e1 b (exprType e2)
                   [(DEFAULT, [], e2)])
strictifyExpr e@(App e1 e2)
   = case e2 of    -- Decides which sorts of arguments to e
                   -- we would like to strictify
      App _ _ → translate
      Case _ _ _ _ → translate
      Cast _ _ → translate   -- May as well, these
      Note _ _ → translate   -- two don't appear on
            -- types anyway
      _ → return e
  where
      translate = do
            -- We must invent a new name to bind the
            -- evaluated thing to. The call to
            -- mkSysLocalM here generates such a new
            -- variable with the same type as e2 and
            -- the name "strict"
         b ← mkSysLocalM (fsLit "strict") (exprType e2)
         return (Case e2 b (exprType e)
            [(DEFAULT, [], App e1 (Var b))])
strictifyExpr e = return e
```

Figure 4: The main worker for the strictifying plugin

We're going to put the new GHC annotations system to work by demonstrating how we can use it to achieve the more mundane of these tasks: selecting some functions to strictify. Let's say that we want to declare an annotation *StrictifyMe* so that programs being compiled can include an annotation of this form:

```
 {-# ANN myfunction StrictifyMe #-}
myfunction = ...
```

This makes use of GHC's new *ANN* pragma that can be used to decorate variables with arbitrary user-defined information. I'll discuss this further later, but for now it suffices to know that if we want to use our new custom *StrictifyMe* annotation, all we need is to have imported a module containing a data declaration like so:

```
 {-# LANGUAGE DeriveDataTypeable #-}
data StrictifyMode = StrictifyMe
  deriving (Typeable)
```

Now, given that our plugin has also imported the annotation-declaring module it becomes very straightforward to write a function to test if a binder has been marked as a candidate for strictification:

```
shouldStrictify :: CoreBndr → CoreM Bool
shouldStrictify bndr = fmap (¬.null) $
  findAnnotations (NamedTarget (getName bndr))
    :: CoreM [StrictifyMode]
```

This check can be incorporated into our original strictification pass straightforwardly, by having the *pass* function of Figure 4 only perform the generic traversal on suitably annotated top-level binders.

As I alluded to earlier, annotations can do a lot more than this: in particular, they can carry data around, and apply to types and modules as well as normal identifiers. Further details on how this works will be available in the fullness of time in the GHC users guide [7], but in the meantime you may find some useful information on the GHC wiki [9].

Conclusions

I hope this sample has given you a bit of a taste for the power of some of the new facilities added to GHC as part of this Google Summer of Code project: not only compiler plugins, but a new phase control and annotations system are now available for GHC users to make use of!

The big caveat with what I've shown you is that none of it is yet merged into GHC, and in all likelihood there will be some further tweaks to how it all fits together before the release. I hope to have the new facilities incorporated the 6.12 release of GHC.

I'm very much looking forward to seeing the uses to which the Haskell community will put the new-found power of compiler plugins, so if I've managed to whet your appetite for a little compiler engineering, please try out the plugin faciliets when they are released! Much more information about how to get started writing plugins is available to support you. Firstly, you can see a version of the sample plugin I developed in this article, along with some others, at *code.haskell.org* [10, 11, 12]. Secondly, as part of the Summer of Code considerable amounts of documentation were made available about the internals of GHC and the GHC API by myself and Thomas Schilling: this will all be available with your GHC release from 6.10 onwards. Lastly, don't forget that you can ask questions about particular things you're stuck on either on the Glasgow Haskell Compiler Users mailing list [13], the developers list [14] or in the *#ghc* IRC channel on *freenode.net*.

Best of luck!

References

[1] `http://www.haskell.org/ghc/`.

[2] Simon Peyton Jones. Call-pattern specialisation for Haskell programs (2007). `http://research.microsoft.com/~simonpj/papers/spec-constr/spec-constr.pdf`.

[3] `http://www.haskell.org/th/`.

[4] Simon Peyton Jones. System F with type equality coercions (2007). `http://research.microsoft.com/%7Esimonpj/papers/ext%2Df/fc-tldi.pdf`.

[5] `http://en.wikipedia.org/wiki/System_F`.

[6] `http://www.haskell.org/ghc/docs/latest/html/users_guide/profiling.html`.

[7] `http://www.haskell.org/ghc/docs/latest/html/users_guide/pragmas.html`.

[8] Dana N. Xu. Extended static checking for Haskell (2006). `http://www.cl.cam.ac.uk/~nx200/research/escH-hw.ps`.

[9] `http://hackage.haskell.org/trac/ghc/wiki/Plugins/Annotations`.

[10] `http://code.haskell.org/strict-ghc-plugin`.

[11] `http://code.haskell.org/cse-ghc-plugin`.

[12] http://code.haskell.org/depth-analysis-ghc-plugin.

[13] http://www.haskell.org/mailman/listinfo/glasgow-haskell-users.

[14] http://www.haskell.org/mailman/listinfo/cvs-ghc.

Appendix

```
data Expr b
  = Var Id
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Note  Note (Expr b)
  | Type  Type
type Arg b = Expr b
type Alt b = (AltCon, [b], Expr b)
data Bind b = NonRec b (Arg b)
  | Rec [(b, (Expr b))]
type CoreExpr = Expr Id
type CoreArg = Arg Id
type CoreAlt  = Alt Id
type CoreBind = Bind Id
```

Figure 5: The Core language datatypes

How to Build a Physics Engine

by Roman Cheplyaka ⟨roma@ro-che.info⟩

This article describes my experience in building physics engine as Summer of Code project mentored by Haskell.org.

About the project

A physics engine is a program or library which predicts evolution of a system according to the laws of physics. It does so by performing a **simulation**; values such as position, velocity etc. are recomputed iteratively at consecutive moments in time.

Physics engines today have many applications, ranging from engineering to animation to virtual reality. For many of these, the speed of the simulation is crucial, while the number of bodies may increase arbitrarily. The natural way to ensure good performance is to make the simulation parallelizable.

Data Parallel Haskell (DPH) [1] is an extension to GHC which allows writing parallel (more precisely, data parallel) programs rather easily. Hence the idea of the project to implement a high-performance physics engine using DPH.

This project got the name Hpysics. From the project's weblog [2]:

> Most of the Summer of Code projects deal with existing projects. However, I'm going to create the new one. And every project needs some name.
>
> So for now I'll pick up the codename hpysics – it's "physics" with two first letters swapped. The first "h" indicates that engine is written in Haskell (this is a convention which lots of Haskell projects use).

The latest source code is available from the darcs repository [3]. Some other information (including a link to the demo video) is available on the project's homepage [4].

I'm going to present my work in two articles. This one will describe general tasks and problems which arise when implementing a physics engine. Another one will dive more deeply into DPH and how it can be used in dynamic simulation.

Limitations

The simulated world is three-dimensional Euclidean space.

The simulated bodies are **rigid**. This means that the shape of the bodies does not change during the simulation. Also, there's currently no support for articulated bodies (although it may be added in future).

All bodies have a convex polyhedral shape. Other convex shapes can be approximated by polyhedra with arbitrary precision. Non-convex shapes can be represented by the union of disjoint convex shapes.

Furthermore, the engine assumes that the only forces that have impact on the bodies are gravity and contact forces which appear when several bodies collide.

Basic simulation

Consider the world consisting of just one body. It has some constant parameters (mass, shape, inertia) and variable parameters (position, orientation, linear and angular velocities). The goal of the simulation is to compute variable parameters at the next moment of time based on their current values.

Since we have a single body, no collisions are possible, so the body moves with the constant acceleration of g.

Linear velocity and position (of the center of mass) evolves as follows:

$$v' = v + g\Delta t,$$

$$x' = x + v\Delta t + \frac{g\Delta t^2}{2}$$

where Δt denotes the step of the simulation.

The above formulas represent the solution to the system of ordinary differential equations $v = dx/dt$, $a = dv/dt$.

Similar equations arise when we consider the rotational component of motion, namely orientation and angular velocity, however, they have no such simple analytic solution. Nevertheless, we can solve the system using approximate numeric methods, e.g. Euler or Runge-Kutta algorithms. Interested readers can read more in Appendix 4 of Mirtich's thesis [5]

The code is in `Hpysics/Simulation.hs`.

Dealing with collisions

We see that in the absence of collisions performing the simulation is a simple task. What changes when we take several bodies and allow them to collide?

Well, two new problems appear. At each step we need to check whether any collisions occurred, and if so, correct each of the colliding bodies' velocities to ensure non-penetration of the rigid bodies. The first task lies in the field of computational geometry and is known as collision detection. The second task lies in the field of physics and is known as collision response.

Collision detection

The main task of the collision detection is, given two shapes in the space, detect whether they intersect. Despite the simplicity of the formulation, this problem is not trivial and was studied by many scientists, and several algorithms were proposed. Most of them deal with polyhedral shapes.

The Lin-Canny algorithm is easy to understand and is quite beautiful. If you have some free time left after reading this TMR issue, I recommend you to download Mirtich's thesis [5] and read his brief overview of the algorithm and the underlying theory (p. 22-24).

On the other hand, there are several problems with it. First, the procedure described above works only if two bodies do not really intersect. This does not seem useful, considering that our goal is the detection of the intersection. To deal with this issue, collision is detected before it occurred by calculating decreasing distances between bodies and using numerical root finding.

The second problem is that the Lin-Canny algorithm heavily relies on the ability to find the closest points of two features, which is not an easy task.

In Hpysics I therefore used another algorithm, V-Clip ("Voronoi Clip") by Mirtich [6]. It does not suffer from both drawbacks of Lin-Canny, at the price of being not so beautiful and being specialized to 3-dimensional space (while Lin-Canny works in n-dimensional Euclidean space). It is worth noting that V-Clip is, in fact, based on the Lin-Canny algorithm.

The code is in `Hpysics/VClip.hs`.

Bounding volumes

Since we need to make many (i.e. $O(n^2)$) one-to-one collision checks, it is natural to make them as cheap as possible. One solution is to construct bounding volumes. The idea is simple: if each of bodies lies in another volume and the volumes do not intersect, then the bodies themselves do not collide.

Of course, the kind of bounding volumes is chosen to simplify collision checks for them. Most popular choices include axis-aligned bounding boxes (AABB), oriented bounding-boxes (OBB) and bounding spheres. In Hpysics, I chose bounding spheres with the center in the body's center of mass, as they are rotationally invariant and thus do not need to be recomputed in each step. The other reason is that they are well-suited for binary space partition trees, which are discussed below.

The code is in `Hpysics/BoundingSphere.hs`.

Broad phase

The bounding volumes optimization, even though it speeds up simulation, does not reduce algorithmic complexity. In order to address this issue, broad phase collision detection was invented. The idea is to divide the space into regions and check only those pairs of bodies which (at least partially) lie in one region.

Again, several algorithms and data structures are available here, including octrees, k-d trees and BSP trees. All of them are trees in the sense that they recursively subdivide space and represent a hierarchy of regions as a tree. They differ in how dividing hyperplanes are chosen and how many children each tree node has.

The code is in `Hpysics/BSP.hs` (incomplete).

Collision response

After we have detected a collision, contact forces should be applied to bodies to ensure their non-penetration. There are several methods to compute these forces: penalty, linear complementarity problem, impulse-based [5] and singular value decomposition [7] (the last two look especially promising, for reasons described in the cited papers).

Hpysics currently employs a very simple strategy for resolving collisions, similar to the one described by Moore and Wilhelms [8]. When two bodies collide, we apply opposite impulses to them (R and $-R$).

Let v_i and ω_i denote linear and angular velocities before collision, and $\bar{v}_i$, $\bar{\omega}_i$ after. From the law of conservation of linear momentum we get three 6 equations (remember, we are in 3D):

$$m_1\bar{v}_1 = m_1v_1 + R,$$

$$m_2\bar{v}_2 = m_2v_2 - R.$$

Similarly, due to conservation of angular momentum we have

$$I_1\bar{\omega}_1 = I_1\omega_1 + \rho_1 \times R,$$

$$I_2\bar{\omega}_2 = I_2\omega_2 - \rho_2 \times R.$$

We introduce a collision frame i, j, k of orthonormal vectors, such that i and j lie in the plane of collision. Two equations follow from the assumption that there's no friction, so the impulse R is perpendicular to the collision plane:

$$R \cdot i = 0,$$

$$R \cdot j = 0.$$

The last equation is the most interesting. Moore and Wilhelms write it as

$$(\bar{v}_2 + \bar{\omega}_2 \times \rho_2 - \bar{v}_1 - \bar{\omega}_1 \times \rho_1) \cdot k = 0,$$

assuming that collision is completely unelastic (here ρ_i is the vector pointing from the center of mass to the collision point). This produced collisions that did not look realistic. Here is how I rewrote it:

$$(\bar{v}_2 + \bar{\omega}_2 \times \rho_2 - \bar{v}_1 - \bar{\omega}_1 \times \rho_1) \cdot k = \alpha(v_2 + \omega_2 \times \rho_2 - v_1 - \omega_1 \times \rho_1) \cdot k.$$

Here α is the coefficient of restitution. Given $\alpha = 0$, we get the Moore-Wilhelms equation. When $\alpha = 1$, we have completely elastic collision. For values of α in between we can get quite realistic results.

We end up with 15 (scalar) equations, and 15 (scalar) unknowns: $R, \bar{v}_1, \bar{v}_2, \bar{\omega}_1, \bar{\omega}_2$. The system of equations is then solved using Gauss-Jordan elimination with maximal pivoting.

The code is in `Hpysics/Collision.hs`.

References

[1] Data parallel haskell homepage. `http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell`.

[2] Hpysics weblog. `http://physics-dph.blogspot.com/`.

[3] Hpysics darcs repository. `http://code.haskell.org/hpysics`.

[4] Hpysics homepage. `http://haskell.org/haskellwiki/Hpysics`.

[5] Brian Mirtich. **Impulse-based dynamic simulation of rigid body systems**. Ph.D. thesis, University of California, Berkeley (1996). `http://www.kuffner.org/james/software/dynamics/mirtich/mirtichThesis.pdf`.

[6] Brian Mirtich. V-clip: Fast and robust polyhedral collision detection. Technical report, ACM Transactions on Graphics (1998). `http://www.merl.com/reports/docs/TR97-05.pdf`.

[7] Brian Mirtich. Rigid body contact: Collision detection to force computation. Technical report, IEEE International Conference on Robotics and Automation (1998). `http://www.merl.com/reports/TR98-01/TR98-01.ps.gz`.

[8] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. In **SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques**, pages 289–298. ACM, New York, NY, USA (1988). `http://www.cs.cornell.edu/Courses/cs667/2005sp/readings/moore88.pdf`.

Hoogle Overview

by Neil Mitchell ⟨ndmitchell@gmail.com⟩

This article gives an overview of the Hoogle tool. We describe the history of Hoogle, the improvements that have been made this summer, and plans for future features. Finally, we discuss the design guidelines of Hoogle 4 – which may be of interest both to budding Hoogle developers and other Haskell projects. This article does not cover the theoretical background of Hoogle.

To try Hoogle online visit http://haskell.org/hoogle*.*

Introduction

To quote from the Cabal description:

> "Hoogle is a Haskell API search engine, which allows you to search many standard Haskell libraries by either function name, or by approximate type signature."

To explore what Hoogle is, and how it can be used, let's expand on some of those phrases:

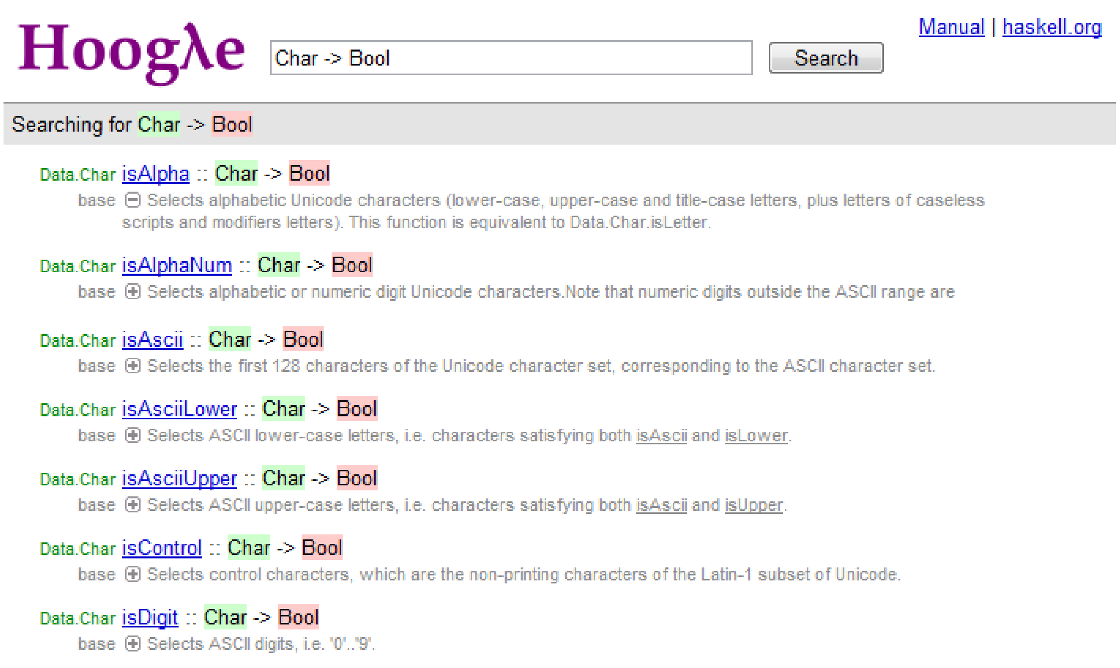
Haskell Hoogle is written in Haskell, and is designed for Haskell programmers.

search engine Hoogle is a tool for searching, in a similar vein to Google. [1]

API Hoogle searches API's, or "Application Programmer Interfaces" – the types and functions provided by a package.

standard libraries By default, Hoogle will search the libraries that are shipped with most Haskell compilers. These libraries include base, array, time, mtl etc.

[1]Hoogle has no affiliation to Google, and the name is intended as a homage.



Char -> Bool

Searching for Char -> Bool

Data.Char isAlpha :: Char -> Bool
base ⊟ Selects alphabetic Unicode characters (lower-case, upper-case and title-case letters, plus letters of caseless scripts and modifiers letters). This function is equivalent to Data.Char.isLetter.

Data.Char isAlphaNum :: Char -> Bool
base ⊞ Selects alphabetic or numeric digit Unicode characters.Note that numeric digits outside the ASCII range are

Data.Char isAscii :: Char -> Bool
base ⊞ Selects the first 128 characters of the Unicode character set, corresponding to the ASCII character set.

Data.Char isAsciiLower :: Char -> Bool
base ⊞ Selects ASCII lower-case letters, i.e. characters satisfying both isAscii and isLower.

Data.Char isAsciiUpper :: Char -> Bool
base ⊞ Selects ASCII upper-case letters, i.e. characters satisfying both isAscii and isUpper.

Data.Char isControl :: Char -> Bool
base ⊞ Selects control characters, which are the non-printing characters of the Latin-1 subset of Unicode.

Data.Char isDigit :: Char -> Bool
base ⊞ Selects ASCII digits, i.e. '0'..'9'.

Figure 1: Hoogle web use.

function name Searches can be performed by name, searching for substrings of function names. One use of Hoogle is as a fast index into Haddock documentation.

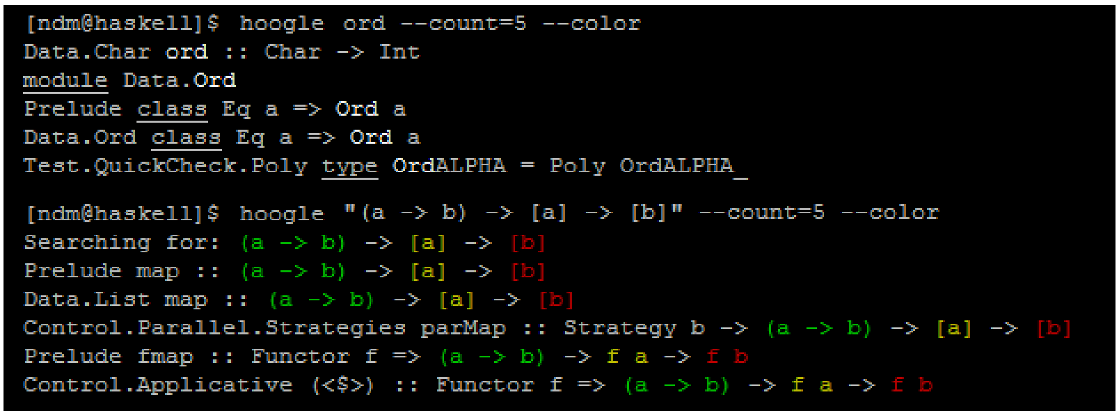
type signature Searches can be performed by type signature, searching for functions of the appropriate type.

approximate Hoogle tries to find the results you want, even if they don't quite match your actual search.

There are three main methods of using Hoogle:

Web Interface The web interface is just like a normal web search engine, requiring no special software or installation. Just visit the website and enter your search terms. An example of the web interface is shown in Figure 1.
URL: http://haskell.org/hoogle/

Command Line The command line tool can be downloaded from Hackage [1], and installed using the standard Cabal commands [2]. The command line tool has more options, and allows searches to be performed while offline. An



```
[ndm@haskell]$ hoogle ord --count=5 --color
Data.Char ord :: Char -> Int
module Data.Ord
Prelude class Eq a => Ord a
Data.Ord class Eq a => Ord a
Test.QuickCheck.Poly type OrdALPHA = Poly OrdALPHA_

[ndm@haskell]$ hoogle "(a -> b) -> [a] -> [b]" --count=5 --color
Searching for: (a -> b) -> [a] -> [b]
Prelude map :: (a -> b) -> [a] -> [b]
Data.List map :: (a -> b) -> [a] -> [b]
Control.Parallel.Strategies parMap :: Strategy b -> (a -> b) -> [a] -> [b]
Prelude fmap :: Functor f => (a -> b) -> f a -> f b
Control.Applicative (<$>) :: Functor f => (a -> b) -> f a -> f b
```

Figure 2: Hoogle command line use.

example session is shown in Figure 2.
URL: http://hackage.haskell.org/cgi-bin/hackage-scripts/package/hoogle

Lambdabot Interface When using Lambdabot [3], you can invoke the Hoogle plugin with `@hoogle`.

Version History

Hoogle is now over four years old, and has undergone four complete rewrites. This section describes each version.

Version 1

I started work on the original version of Hoogle before I started my PhD, with only basic Haskell knowledge, and long before I had ever encountered a Monad. Realising that my PhD was likely to be dominated by Haskell, I decided to develop a tool to help beginners (such as myself) find some of the useful functions located in the standard libraries. While learning Haskell I had watched experienced programmers take my code and simplify it significantly, by using some clever function whose existence I was unaware of. I wanted to perform the same tricks!

The initial version of Hoogle was web based, and relied on client-side Javascript to perform searches. The use of Javascript was unavoidable for a client-side web program, but even with my limited Haskell experience I longed for proper algebraic data types, pattern-matching and type-safety. The list of functions was obtained from the ZVON Haskell Reference [4], who kindly provided all their function data in XML, for anyone to use. Hoogle worked, but suffered from huge page load times (it had to transfer 8MB of data), and wasn't particularly easy to use.

Version 2

Once I started my PhD I was exposed to lots of Haskell, and decided to rewrite Hoogle in Haskell. Originally version 2 was a direct port of version 1 to Haskell, with the searching logic moved to a server-side CGI program. However, once Hoogle was written in Haskell it became much easier to explore type searching, treating types as algebraic data structures. As soon as version 2 was placed on the web, members of the `#haskell` IRC channel [5] began to use and discuss it. The feedback and encouragement provided by `#haskell` resulted in many improvements.

Version 2 was my first real experience at programming a large Haskell project, and it showed. The code was poorly organised. The parser could easily be crashed by entering malformed searches. Many features were littered throughout the code, without clear isolation. Much of the code failed to make use of the standard functional idioms. Eventually I reached the stage where every improvement became an increasingly large amount of work, and became more likely to conflict with an existing feature.

Version 3

The goal of version 3 was to improve the code so that features could be added easily. Hoogle 3 was always intended to be the definitive version, which would be modified, but never rewritten from scratch. The ZVON function list was replaced with information extracted from Haddock [6], which allowed all of the hierarchical libraries to be searched. The experience of writing version 2 provided many insights, which were incorporated. A log of all the searches performed was used to determine where Hoogle didn't match the users expectations. The end result was a more polished tool.

However, version 3 was still insufficient in many ways – the most obvious design flaw was the inability to search for higher-kinded type classes, of which Monad is by far the most common. The other problem was scalability, Hoogle 3 scaled linearly in the number of functions available, which worked fine on a small function database, but became a problem when attempting to search more libraries.

Version 4

After submitting my PhD, I spent the summer working on Hoogle, sponsored by the Google Summer of Code. Once again, version 4 was a complete rewrite. The largest change is that instead of using a text file containing a list of functions, version 4 uses a binary database. This change allows Hoogle to perform searches faster, and to scale better as more functions are added to the database.

By storing some precomputed information, searches can be made faster. For example, if the database included the answer for all queries of length 5 and below, these searches could be answered very quickly. However, the database would also grow unacceptably large. Version 4 required many trade-offs, choosing the right representation to maximise search speed and minimise database size. For example, text searching in version 3 has time complexity $O(m \cdot n)$, where n is the number of functions and m is average length of a function name. Early releases of version 4 used a trie and required only $O(m)$ to find all the results, but at the cost of a large database. The current version requires $O(m \cdot \log n)$ to find the answers which match the prefix of the search, then uses heuristics to find additional answers quickly – although with a time complexity of $O(m \cdot n)$.

The Future

I hope that version 4 will be the last ever rewrite of Hoogle. Now there is a stable base to work from, the hope is that additional features can be added neatly. Some of the planned features are given in this section, but none have any timescale given.

Index Hackage

Hoogle currently doesn't index all the packages available on Hackage, but it should. The work done for version 4 has enabled Hoogle to scale to the necessary number of packages, so hopefully Hoogle requires no changes. However, before a package can be indexed by Hoogle it must have documentation generated by Haddock, and this has proved a stumbling block. To install all of Hackage is a challenge, and to do so on my ailing Windows machine is an impossibility. Hackage already generates Haddock documentation for all packages, and once this process has been revamped, hopefully Hoogle information can be generated at the same time.

Hoogle Local

Hoogle Local is a graphical user interface to Hoogle, giving the same interface as the web version, but operating offline. Hoogle Local allows users to locally install API databases, customise Hoogle to a greater degree, and doesn't require an internet connection – but provides the same user friendly interface as the web version. This feature has been partially implemented, making use of Firefox 3 as an XULRunner host. The development version is useable but lacks the necessary polish to release more widely.

Multilingual Hoogle

Hoogle is currently focused on Haskell, with support for most GHC type system extensions. But internally, Hoogle does not support all of Haskell's advanced type features – multi-parameter type classes are not supported directly, but are translated into single-parameter type classes. A more accurate description might be that Hoogle supports searching over a core type language, which Haskell's type language is translated into. We suspect that other programming languages could also be translated into Hoogle's type system. There are three classes of programming languages that Hoogle might support:

- Languages based on the Hindley-Milner type system. Some of these languages have type systems which are a subset of Haskell. These languages should permit a fairly straightforward translation – obvious examples include ML and Clean.
- Strongly typed languages, typically object oriented. F# has shown that a functional language can interface with object-oriented languages in a reasonably natural way, by adding some features and by translating others. Hoogle could use some of the same ideas, and expand to search languages such as Java and C#.
- Untyped languages, typically scripting languages. Languages such as Perl, Python and Javascript don't have any formal types in their interfaces, but often there is some notion of what subset of values should be passed to which function – sometimes encoded as a runtime check. This information could be used to give approximate types to functions, and allow Hoogle searching.

Hopefully one day Hoogle will be a general purpose programming language search engine, that works well for both Haskell and other programming languages.

Design Guidelines

This section explains the guidelines used for organising the Hoogle codebase. This information is intended to serve both as a reference to budding Hoogle developers, and as my current view of best practices in large-scale Haskell development. Hoogle has been rewritten from scratch four times, each time incorporating knowledge gained from previous attempts, and iteratively improving the code layout. Some of these lessons may apply to other projects, and help avoid painful rewrites!

Structure your code as a library

Hoogle is a program, but is structured as a library, with client programs which make use of the library. Any code within the Hoogle module tree is part of the library, and is carefully checked to expose a sensible interface. Each client which makes use of Hoogle has its own top-level module. For the purposes of deployment, there is no library, but if the need arises an explicit library can easily be added. By forcing a split between the underlying functionality and the user interface, and by imagining other potential users of the library, we gain a cleaner separation of concerns.

Put types in their own module

All the data type definitions are placed in a module of their own, at the bottom of the import hierarchy. For example, type signatures are defined in the module Hoogle.TypeSig.Type. This module also contains basic utility functions (isTypeApp, fromTypeApp) and instances (Eq, Show). I have found that by separating out data type definitions, it is much easier to avoid mutual recursion between modules.

Group operations on a type

All basic operations on a type share the same module prefix. For example, operations on type signatures are given module names such as Hoogle.TypeSig.Parse and Hoogle.TypeSig.Render, each responsible for one particular operation. For modules wishing to use type signatures there is Hoogle.TypeSig.All which imports and re-exports all the modules within Hoogle.TypeSig, usually with a more restrictive export list. The intention is that no module outside of Hoogle.TypeSig should ever import a module other than .All. Many internal details can be hidden from the users of type signatures, which are useful to expose to the operations on type signatures.

Originally the .All module was simply called Hoogle.TypeSig – which seems like a more natural choice. However, having the .All module in the same directory as the other related modules is beneficial, and makes it easier to keep the modules in sync. Additionally, it becomes easy to spot when a module from a different module prefix imports something in violation of the guidelines.

Use a hierarchy

Hoogle is structured as a library, with Hoogle.All exporting all the definitions that a client may wish to use. Anything exported from this module is intended as a permanent interface, and is relatively stable. Unfortunately, often clients of the

Hoogle library need access to more specific details – details that are semi-stable, but which are not ready to form part of the standard interface. By letting clients import modules such as Hoogle.TypeSig.All, the official interface avoids getting polluted, but the features can be still implemented.

Over time I hope that clients importing modules other than Hoogle.All will decrease, as proper thought is given to the interface, and the right abstractions are identified. By allowing greater flexibility the hope is that long-term maintenance will not be hampered by the pressing need to add one particular feature.

Provide one executable

Version 3 had four executable programs – one to generate ranking information, one to do command line searching, one to do web searching, and one to do regression testing. Version 4 has one executable, which does all the above and more, controlled by flags. There are many advantages to providing only one end program – it reduces the chance of code breaking without noticing it, it makes the total file size smaller by not duplicating the Haskell run-time system, it decreases the number of commands users need to learn. The move to one multipurpose executable seems to be a common theme, which tools such as darcs and hpc both being based on one command with multiple modes.

Conclusion

Hoogle is not yet finished. In addition to the future tasks given in this article, there are plenty of suggestions and bugs outstanding, most of which are documented in the bug database (http://code.google.com/p/ndmitchell/issues/list). Some bugs are marked as beginner, meaning they can be easily tackled by someone new to Hoogle – and in some cases someone new to Haskell. If anyone wants to help, please email me at ndmitchell@gmail.com.

As the task of programming goes from one of painting on a blank canvas, to one of plumbing together existing components, Haskell has a distinct advantage with its high-level abstractions. As the number of libraries increases, finding the right functionality becomes harder. Hoogle aims to help by providing a simple way to find the right functions.

Acknowledgements

Hoogle has benefited greatly from feedback, encouragement, technical contributions, bug reporting and more. Versions 2 and 3 were helped substantially by many members of the functional programming group at York University, and the users of `#haskell`. While writing version 4 I was mentored by Niklas Broberg, received lots of help from Duncan Coutts, and was funded by Google through the haskell.org project. Hoogle has benefited from substantial contributions from David Waern, Don Stewart, Emily King, Esa Ilari Vuokko, Gaal Yahas, Ganesh Sittampalam, Gwern Branwen, Henk-Jan van Tuyl, Mike Dodds, Thomas Davie, Thomas Jäger, Tillmann Rendel and Udo Stenzel.

References

[1] Hackage. `http://hackage.haskell.org/packages/hackage.html`.

[2] Cabal. `http://haskell.org/cabal/`.

[3] Lambdabot. `http://www.haskell.org/haskellwiki/Lambdabot`.

[4] ZVON Haskell Reference. `http://www.zvon.org/other/haskell/Outputglobal/index.html`.

[5] Haskell IRC Channel, `#haskell`. `http://www.haskell.org/haskellwiki/IRC_channel`.

[6] Haddock. `http://haskell.org/haddock/`.