The Monad.Reader Issue 7

by Duncan Coutts ⟨duncan.coutts@comlab.ox.ac.uk⟩
and Dimitry Golubovsky ⟨golubovsky@gmail.com⟩
and Yaron Minsky ⟨yminsky@janestcapital.com⟩
and Neil Mitchell ⟨ndm@cs.york.ac.uk⟩
and Matthew Naylor ⟨mfn@cs.york.ac.uk⟩

April 30, 2007

Wouter Swierstra, editor.

Contents

Editorial

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

After the success of the last issue, I was very happy indeed to receive four excellent submissions: Matthew Naylor introduces a combinator library for describing the behaviour of digital circuits; Yaron Minsky recounts how functional programming is moving to Wall Street; Duncan Coutts provides an in depth review of Graham Hutton's new book, Programming in Haskell; and finally, Dimitry Golubovsky, Neil Mitchell, and Matthew Naylor give a tutorial on the secrets of Yhc's core language. I do hope that these sterling contributions indicate that it is feasible to continue to release a new issue of The Monad.Reader every quarter. Finally, I'd like to thank Andres Löh for his help in continuing to perfect the LaTeX style files.

As always, I would like to welcome your submissions and feedback. For the moment, however, enjoy the latest issue!

A Recipe for controlling Lego using Lava

by Matthew Naylor ⟨mfn@cs.york.ac.uk⟩

*The Haskell library **Lava** is great for describing the **structure** of digital circuits: large, regular circuits can be captured by short, clear descriptions. However, structural circuit description alone – and hence Lava – has a somewhat limited application domain. Many circuits are more appropriately expressed in terms of their desired **behaviour**.*

*In this article, I present a new module for Lava – called **Recipe** – that provides a set of behavioural programming constructs, including mutable variables, sequential and parallel composition, iteration and choice. Mutable state is provided in the form of rewrite rules – normal Lava functions from state to state – giving a powerful blend of the structural and behavioural styles.*

The approach taken here is also applicable to software-based embedded systems programming. Indeed, I have developed a simple C backend for Lava, and my final example – a program that controls a brick-sorter robot – runs on a Lego Mindstorms RCX microcontoller.

Overview

This article is structured in three parts:

- An introduction to structural circuit description in Lava.
- Implementation and discussion of the Recipe module.
- Example applications of Recipe, including a sequential multiplier and a controller for a Lego brick-sorter.

Structural Circuit Description in Lava

Lava is a library for hardware design, developed at Chalmers University [1]. In essence, it provides an abstract data type `Bit`, along with a number of common operations over bits. To illustrate, the following Lava function takes a pair of bits and sorts them into ascending order.

```
bitSort         :: (Bit, Bit) -> (Bit, Bit)
bitSort (a, b) =  (a ==> b) ? ((a, b), (b, a))
```

Here, two Lava functions are called: implication (`==>`), which is the standard ordering relation on bits, and choice (`?`), which in hardware terms is a multiplexor, and in software terms is a C-style if-expression.

Lava allows any function over some (nested) structure of bits, be it a tuple, a list, a tree, or whatever, to be:

- simulated by applying it to sample inputs,
- turned into logical formula that can be verified for all inputs of a given, fixed size by a model checker, and
- turned into a VHDL netlist that can be used to configure an FPGA.

For example, to simulate our bit sorter, we can just say:

```
Lava> simulate bitSort (high, low)
(low, high)
```

Looks correct, but to be sure, we can formulate a correctness property:

```
prop_bitSort (a, b) = c ==> d
  where
    (c, d)          = bitSort (a, b)
```

and verify it **for all inputs**:

```
Lava> smv prop_bitSort
Valid
```

In this case we have requested to use the SMV verifier [2]. Now that we are sure our circuit is correct, we can turn it into a VHDL netlist, ready for circuit synthesis:

```
Lava> writeVhdlInputOutput "BitSort" bitSort
        (var "a", var "b") (var "c", var "d")
Writing to file "BitSort.vhd" ... Done.
```

This command generates the file `BitSort.vhd`, and requests that in the VHDL code, the inputs be named `"a"` and `"b"`, and outputs `"c"` and `"d"`.

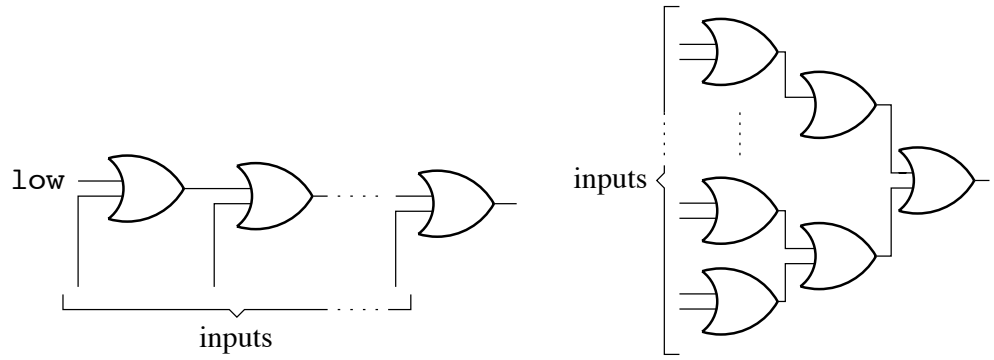

Figure 1: Linear reduction (left) and tree reduction (right).

Larger Circuits

The `bitSort` circuit is of fixed size: it always takes two inputs and produces two outputs. However, in Lava, circuit descriptions can be parameterised by the numbers of inputs and outputs. For example, logical disjunction over an arbitrary number of bits can be expressed as:

```
orl        :: [Bit] -> Bit
orl []     =  low
orl (a:as) =  a <|> orl as
```

As physical circuits have a fixed size, the exact number of inputs to a Lava description must be decided when calling a function like `writeVhdl`. This is because Lava, in effect, expands out the recursion in a description. For example, the `orl` description corresponds to the circuit structure shown on the left in Figure 1.

It is clear from Figure 1 that the circuit structure of `orl` has a linear shape: the time taken to compute the output is linear in the number of inputs. With parallelism, and the fact that disjunction is commutative and associative, we can do much better using a tree-shaped structure:

```
tree            :: (a -> a -> a) -> [a] -> a
tree f [a]      =  [a]
tree f (a:b:bs) =  tree f (bs ++ [f a b])
```

The structure of the description `tree (<|>)` is shown on the right in Figure 1. To be sure that it computes the same function as `orl`, we can verify the following correctness property:

```
prop_OrTree = forall (list 8) $ \as ->
                orl as <==> tree (<|>) as
```

Notice that we have only stated that the equivalance holds for size 8 input lists. In fact, this property does not hold in general, since `tree` is undefined on the empty list.

Sequential Circuits

Sequential circuits are circuits that contain clocked memory elements and feedback loops. They can be described in Lava using `delay` elements and value recursion. The `delay` primitive takes a default value and an input. It outputs the default value on the initial clock cycle, and from then on, the value of the input from the previous clock cycle. For example:

```
Lava> simulateSeq (delay low) [high, high, high, low, low]
[low,high,high,high,low]
```

The `delay` element can be thought of as having an internal state, and can therefore remember its input value from the previous clock cycle.

Value recursion, also known as "circular programming", is used to express feedback loops. For example, the following description implements a delay element with an input-enable line. The internal state is only updated when the enable line is active.

```
delayEn init (en, inp) = out
  where
    out                = delay init (en ? (inp, out))
```

The structure of `delayEn` is shown on the left in Figure 2. This function will prove useful later, in the implementation of Recipe.

Another function that will prove useful later is the set-reset latch. It takes two inputs, set and reset. A pulse on the set or reset line causes the internal state of the delay element to go high or low respectively. For our purposes, it is unimportant what happens when the set and reset lines are pulsed at the same time.

```
setReset (s, r) = out
  where
    q           = delay low (and2 (out, inv r))
    out         = or2 (s, q)
```

The structure of `setReset` is shown on the right in Figure 2.

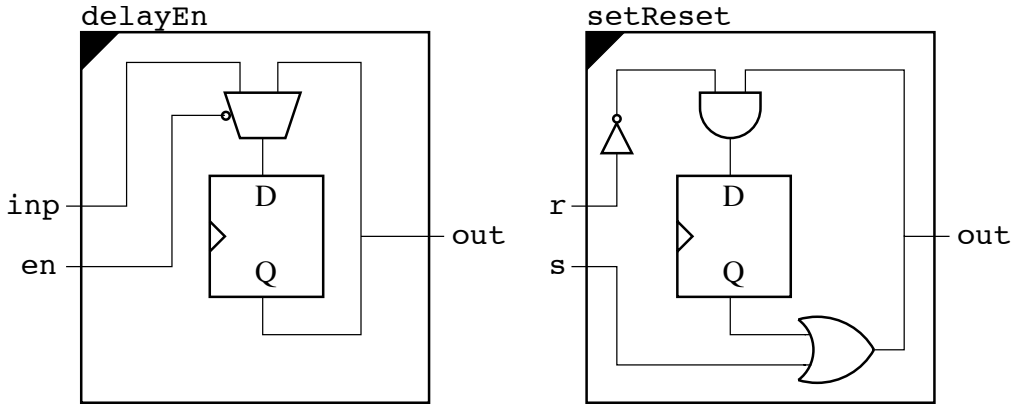

Figure 2: Delay with input enable (left) and a set-reset latch (right).

Recipe – A Library for Behavioural Description

The structural approach is ideal for many kinds of circuit – see the Lava tutorial [1] and Mary Sheeran's web page [3] for more examples. However, when it comes to circuits with multiple feedback loops, structural descriptions can become rather awkward to write and hard to understand. Even the simple feedback loops, like those shown in Figure 2, can take a while to understand.

What we need is a more abstract notion of state and control-flow. One such abstraction, which will be used in Recipe, is Page and Luk's hardware variant of Occam [4]. It provides basic program statements such as skip and assignment, which take one clock cycle to complete, and powerful combining forms including sequential and parallel composition, choice and iteration.

But how can such a behavioural language be integrated with Lava? Koen Claessen and Gordon Pace give us a big hint: they roughly say "just describe the language syntax as a Haskell data type, and write an interpreter for that language in Lava". This approach, which may seem obvious, turns out to be very neat. Indeed, Claessen and Pace have already developed behavioural langauges on top of Lava, but none are quite as general as the one which I present in here.

I will take a slightly different approach to Claessen and Pace here. For a more direct presentation, Recipe will be implemented directly as a set of monadic functions as opposed to a single evaluation function over an abstract syntax. In practice however, an abstract syntax is preferred because optimisations can be applied.

The Recipe Monad

In Page and Luk's Occam variant, each program construct can be thought of as a black box which takes a **start** signal and produces a **finish** signal. So, my Recipe monad is a function that takes a start bit and returns a finish bit:

```
data Recipe a = Recipe { run :: Bit -> Env -> (Bit, Env, a) }
```

You will notice that it also takes an environment (of type `Env`) and produces an environment. For now, the purpose of the environment is unimportant – it will be explained later. The `Recipe` data type can be made a monad as follows:

```
instance Monad Recipe where
  return a = Recipe $ \start env -> (start, env, a)
  m >>= f  = Recipe $ \start env ->
               let (fin0, env0, a) = run m start env
                   (fin1, env1, b) = run (f a) fin0 env0
               in  (fin1, env1, b)
```

This is really no more than a state monad, but for those who are less familiar with monads, here are the important points:

- `return` doesn't read or write any environment information. It just connects the start signal directly to the finish signal.
- `>>=` sequentially composes two recipes. It passes the start signal to the first recipe, which returns a finish signal. This finish signal is then passed as the start signal to the second recipe.
- `>>=` also threads the environment through the two recipes.

This completes the definitions of our first two behavioural constructs – the circuit which does nothing, and sequential composition!

Skip and Wait

The `return` construct is great for doing nothing, but why stop there when we can do nothing **and** take one clock cycle to do it! This is what `skip` does:

```
skip :: Recipe ()
skip =  Recipe $ \start env -> (delay low start, env, ())
```

The finish signal simply becomes the start signal delayed by one clock cycle. Note that the idea of these start and finish signals is that they carry single-cycle pulses to indicate the start or finish event.

The great thing about making Recipe a combinator library is that recipes are just Haskell values. New ones can be defined in terms of existing ones:

```
wait   :: Int -> Recipe ()
wait 0 =  return ()
wait n =  skip >> wait (n-1)
```

Now we can do nothing and use up an arbitrary number of clock cycles to do it. Despite my sarcasm, `skip` and `wait` are actually rather useful, and important.

Parallel Composition

Recipes can be composed in parallel using the `|||` operator, defined as follows:

```
(|||)   :: Recipe a -> Recipe b -> Recipe (a, b)
p ||| q =  Recipe $ \start env ->
             let (fin0, env0, a) = run p start env
                 (fin1, env1, b) = run q start env0
                 fin             = setReset (fin0, fin)
                                       <&> setReset (fin1, fin)
             in  (fin, env1, (a, b))
```

Parallel composition behaves according to a fork/join semantics. This means that an expression of the form `p ||| q` starts `p` and `q` at the same time, and finishes only when **both** `p` and `q` have finished. To implement this blocking behaviour, we feed each of the finish signals of `p` and `q` into the set-line of a set-reset latch, and combine the outputs with an and-gate. This final finish signal is then fed back into the reset-line of each latch, so that the control circuitry is left in a reuseable state, ready for the next time it is executed (for example, if it occurs in the body of a loop).

Choice

An if-then-else construct over recipes is defined as:

```
cond       :: Bit -> Recipe a -> Recipe b -> Recipe ()
cond c p q =  Recipe $ \start env ->
                let (fin0, env0, _) = run p (start <&> c) env
                    (fin1, env1, _) = run q (start <&> inv c) env0
                in  (fin0 <|> fin1, env1, ())
```

It takes a condition bit `c`, and two recipes, `p` (the then-branch) and `q` (the else-branch). Notice that `c` is of type `Bit`, meaning that any normal Lava function can be used to describe the condition. The conjunction of the start signal and `c` is used to trigger `p`, and the conjunction of the start signal and the inverse of `c` is used to trigger `q`. The the final finish signal is equal to the disjunction of the finish signals of `p` and `q`.

Iteration

A construct that repeatedly runs a recipe while a condition holds is defined as:

```
iter     :: Bit -> Recipe a -> Recipe a
iter c p =  Recipe $ \start env ->
              let (fin, env', b) = run p (c <&> ready) env
                  ready          = start <|> fin
              in  (inv c <&> ready, env', b)
```

The loop body `p` is said to be "ready" when the start signal is active, or its own finish signal is active. However, it is only triggered when it is both ready and the loop-condition holds. If it is ready, and the loop-condition does not hold, then the overall finish signal is triggered.

When using loops, the programmer must take care not to make a loop-body that takes no clock-cycles to complete – this would result in a combinatorial loop in the circuit! If a loop body can take zero clock cycles under certain conditions, then a `skip` should be inserted for safety. This is why I earlier claimed that `skip` is important.

Again, we can define other useful combinators on top of existing ones:

```
forever     :: Recipe a -> Recipe a
forever p   =  iter high p

waitWhile   :: Bit -> Recipe ()
waitWhile a =  iter a skip

waitUntil   :: Bit -> Recipe ()
waitUntil a =  iter (inv a) skip
```

The Environment

We have progressed reasonably far without needing an environment, i.e. any global information about the circuit we are constructing. However, the only remaining

constructs that we wish to define are dependent on each other – constructs to create, read and write mutable variables. To define them separately, we need an environment so that they can share information.

In hardware, a mutable variable corresponds to a delay element with an input-enable (recall `delayEn`). The state of the delay element is updated only when an assignment occurs that activates the enable line. So, an input to a mutable variable is defined as:

```
type Inp = (Bit, Bit)
```

The first element of the pair is the input-enable and the second is the actual input value being assigned to the variable.

The purpose of the environment is to the provide a mapping between variables and their inputs and outputs:

```
type Var = Int

data Env = Env { freshId    :: Var
               , readerInps :: Map Var [Inp]
               , writerInps :: Map Var [Inp]
               , outs       :: Map Var Bit }
```

A variable, which is represented by a unique integer, may have multiple inputs (one for each assignment to that variable in the program), but only one output. Notice that there are **two** mappings between variables and their inputs in the environment. The idea is that the `writerInps` mapping is used like a log, to **record** which variables have which inputs as we go. And the `readerInps` mapping is used to **lookup** the inputs to a variable. Ultimately, when it comes to running the `Recipe` monad (or "following the recipe"), the `writerInps` mapping will be passed circularly as the `readerInps` mapping:

```
follow              :: Bit -> Recipe a -> (Bit, a)
follow start recipe =  (fin, a)
  where
    (fin, env, a)   =  run recipe start initialEnv
    initialEnv      =  Env 0 (writerInps env) empty empty
```

The idea is similar to that of tying the output of a reader-writer monad to its own input, as presented by Russell O'Connor in the last edition of The Monad.Reader[5]. The idea was explained to me by Emil Axelsson, who uses something similar in Wired [6]. Apparently Koen Claessen has also used the idea in an old version of Lava, so it certainly seems quite useful!

It should not be too surprising that the environment is circular as the input to a variable could depend on its own output.

Before we're ready to implement mutable variables, we need a couple of helper functions over environments. The first of these, `createVar`, obtains a fresh variable from the environment and records a given value as that variable's output:

```
createVar       :: Env -> Bit -> (Var, Env)
createVar env a =  (v, env')
  where
    v           =  freshId env
    env'        =  env { freshId = v + 1
                       , outs    = insert v a (outs env) }
```

The second, `addInps`, takes a list of variable/input pairs and adds them to the `writerInps` mapping.

```
addInps         :: Env -> [(Var, [Inp])] -> Env
addInps env as =  env { writerInps = inps }
  where
    inps       =  unionWith (++) (fromList as) (writerInps env)
```

Mutable Variables

Recall that in hardware, a mutable variable corresponds to a delay element with an input-enable (`delayEn`). As `delayEn` is just a normal function, we first need to know what input to pass it. So we create a fresh variable `v` and then ask the environment for the inputs to `v`. We then combine all these inputs into one as follows:

- The overall input-enable line is equal to the disjunction of the enable lines of all the inputs. We assume that only one input-enable line is active at any one time.
- The overall input value is selected from all the inputs according to which input-enable line is high, using a kind of multiplexor.

Finally, the output from the delay element is recorded in the environment as being the output of `v`. So, the function to create a mutable variable, `newVar`, is defined as:

```
newVar :: Recipe Var
newVar =  Recipe $ \start env ->
            let (v, env') = createVar env out
                inps      = readerInps env ! v
                out       = delayEn low (enable, value)
                enable    = orl (map fst inps)
                value     = orl (map and2 inps)
            in  (start, env', v)
```

Obtaining the value of a variable is just a case of looking it up in the output mapping:

```
readVar   :: Var -> Recipe Bit
readVar v =  Recipe $ \start env -> (start, env, outs env ! v)
```

Writing a value to a variable is achieved by using a more general assignment function called `assign`:

```
writeVar     :: Var -> Bit -> Recipe ()
writeVar v a =  assign [(v, a)]
```

The more general function is capable of performing a number of assignments at the same time. It works by pairing the bit in each assignment with the start signal (which acts as the input-enable), and adding the resulting assignments to the `writerInps` mapping. Notice that like `skip`, assignment takes one clock cycle to complete:

```
assign    :: [(Var, Bit)] -> Recipe ()
assign as =  Recipe $ \start env ->
               let as' = map (\(a, b) -> (a, [(start, b)])) as
               in  (delay low start, addInps env as', ())
```

This interface to mutable state – with `newVar`, `readVar`, and `writeVar` – will be familiar to programmers who have used the `IORef` and `STRef` types in Haskell. However, to completely copy those interfaces, a `modifyVar` function is required:

```
modifyVar     :: (Bit -> Bit) -> Var -> Recipe ()
modifyVar f v =  do a <- readVar v ; writeVar v (f a)
```

A slight generalisation of `modifyVar` is `rewriteVar`, which gives the feeling that functions are being used as rewrite rules:

```
rewriteVar       :: (Bit -> Bit) -> Var -> Var -> Recipe ()
rewriteVar f v w =  do a <- readVar v ; writeVar w (f a)
```

The `rewriteVar` function provides a nice abstraction. It gives mutable state a more consistent and higher-level interface than seperate `readVar` and `writeVar` functions. Unfortunatly, rewriting is limited to "single variables at a time".

Generalised Rewriting

Rewrite rules would be much more powerful if they could be applied between arbitrary structures of variables, e.g. tuples and lists, rather than just single variables. This can achieved quite easily using a type class.

```
class Same a b | a -> b, b -> a where
  smap :: (Var -> Bit) -> a -> b
  szip :: a -> b -> [(Var, Bit)]
```

The idea is that `Same a b` holds between any two types `a` and `b` which have an identical structure – the only difference is that `a` contains variables at its leaves, whereas `b` contains bits. Figure 3 contains example instances of the `same` class.

Now we can define a generalised read function over structures of variables:

```
read   :: Same a b => a -> Recipe b
read a =  Recipe $ \s env -> (s, env, smap (outs env !) a)
```

And a generalised rewrite function:

```
rewrite       :: (Same a b, Same c d)
              => (b -> d) -> a -> c -> Recipe ()
rewrite f a b =  do x <- read a ; assign (szip b (f x))
```

It is common that the source and destination of a rewrite rule are the same:

```
apply f a = rewrite f a a
```

It is also common to initialise variables with a constant value:

```
set a b = rewrite (\() -> b) () a
```

Futhermore, it is useful to generalise `cond` and `iter` so that their conditions are functions over variables:

```
ifte f a p q = do b <- read a ; cond (f b) p q

while f a p  = do b <- read a ; iter (f b) p
```

The functions `rewrite`, `apply`, `ifte` and `while` provide a neat interface between structural Lava functions and behavioural recipes.

In this section, we have generalised over explicit reading and writing of individual variables, but not creation of variables. This is not a problem – `newVar` is what we want – but it can also be useful to create a list of variables of a given size, called a register:

```
newReg   :: Int -> Recipe [Var]
newReg n =  sequence (replicate n newVar)
```

This completes the definition of the Recipe module.

```
instance Same Var Bit where
  smap f a = f a
  szip a b = [(a, b)]

instance Same a b => Same [a] [b] where
  smap f a           = map (smap f) a
  szip [] bs         = []
  szip (a:as) []     = szip a (smap (const low) a) ++ szip as []
  szip (a:as) (b:bs) = szip a b ++ szip as bs

instance Same () () where
  smap f  () = ()
  szip () () = []

instance (Same a0 b0, Same a1 b1) => Same (a0, a1) (b0, b1) where
  smap f (a, b)          = (smap f a, smap f b)
  szip (a0, a1) (b0, b1) = szip a0 b0 ++ szip a1 b1

instance (Same a0 b0, Same a1 b1, Same a2 b2) =>
           Same (a0, a1, a2) (b0, b1, b2) where
  smap f (a, b, c)  = (smap f a, smap f b, smap f c)
  szip (a0, a1, a2)
       (b0, b1, b2) = szip a0 b0 ++ szip a1 b1 ++ szip a2 b2
```

Figure 3: Example instances of the `Same` class

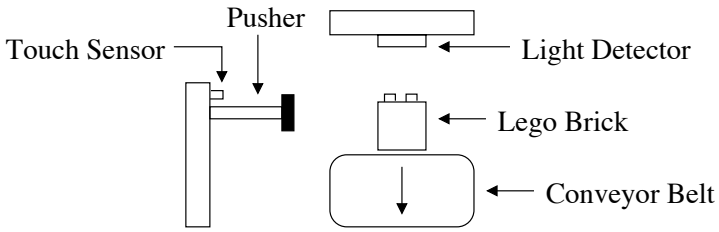

Figure 4: Lego brick-sorter (the brick is moving towards you)

Examples

In this section we use Recipe to implement a few short but useful programs: a sequential multipler, and a controller for a Lego brick-sorter.

A Sequential Multiplier

On some FPGA devices, multiplication in a single clock cycle is an expensive operation. It can therefore be useful to use a sequential multipler instead – one that takes several clock cycles to complete, but which has a much shorter critical-path delay, and which uses less FPGA resources. A sequential multiplier is usually implemented using the shift-and-add algorithm.

In Lava, numbers are typically represented as lists of bits, with the least significant bit coming first. In fact, such lists can be defined to be an instance of Haskell's `Num` class. Lava contains most of the required definitions already, in it's `Arithmetic` library.

The other ingredients we need, further to binary addition from the `Num` class, are functions for shifting numbers left and right:

```
shl       :: [Bit] -> Int -> [Bit]
a `shl` n =  drop n a ++ replicate n low

shr       :: [Bit] -> Int -> [Bit]
a `shr` n =  reverse (reverse a `shl` n)
```

Here, for simplicity, we assume the standard binary encoding of non-negative numbers i.e. we're not dealing with two's complement, so we just pad numbers with zeros rather than doing proper sign-extension.

The next step is to define the core of the algorithm as a rewrite-rule. If we're multipling `a` by `b` with an accumulator `acc`, then this step is as follows:

```
step              :: ([Bit], [Bit], [Bit]) -> ([Bit], [Bit], [Bit])
step (a, b, acc) =  (a `shr` 1, b `shl` 1, head b ? (acc+a, acc))
```

Now, to implement the multiplier, we just need to apply the `step` rule while `b` is not equal to zero:

```
mult        :: ([Var], [Var]) -> Recipe [Var]
mult (a, b) =  do acc <- newReg (length a)
                  set acc 0
                  while orl b (apply step (a, b, acc))
                  return acc
```

This recipe takes $n + 1$ clock cycles to complete in the worst case, where n is the bit-length of the multiplicand.

A Lego Brick-Sorter

As part this year's "Reactive Systems Design" module at the University of York, Jan Tobias Mühlberg designed a Lego brick-sorter that students would build and program during the practical sessions. As a demonstrator on this module, I couldn't resist developing a Haskell solution to the problem!

The basic structure of Tobias's brick-sorter is shown in Figure 4. It is intended to operate as follows:

- Initialise: the pusher-arm is moved into its resting position – the position in which it is touching the touch-sensor. Note that the pusher arm is either "moving" or "not moving" – there is no notion of direction. Mechanically, the pusher will automatically flip direction when it reaches as far as it can strecth, or when it reaches the resting position.

- Sort: when the light-detector dectects a reflective, light-coloured brick then the pusher should push the brick off the conveyor belt, and return to the resting position.

A program to control the brick-sorter according to this behaviour is shown in Figure 5. Since the Lego Brick-Sorter is controlled using the Mindstorms RCX micro-controller, I needed to develop a C backend for Lava [7] before this program could actually be used. Although Lava is more suited to describing large parallel circuit structures, Recipe programs are suitable for sequential, CPU-based archiectures too. The C code generated by Lava for a Recipe program will typically be fairly efficient.

Concluding Discussion

I have shown how a subset of Page and Luk's variant of Occam can be defined as a monadic library in Lava. In particular, I provided a combinator for applying normal Lava functions as rewrite rules over arbitrary structures of mutable variables. This results in a powerful combination of the structural and behavioural styles.

There is one issue in Page and Luk's Occam that leaves me slightly unsatisfied: when a variable is assigned two different values in parallel (in the same clock cycle), the behaviour of the circuit is undefined. In Recipe, I would like to investigate the use of **atomic** rewrite rules to obtain a more useful behaviour in such situations. This would potentially allow constructs such as communication channels to be defined within the language. The language BlueSpec already provides atomic actions, but it is quite different to Occam.

```
sorter                 :: (Bit, Bit) -> Recipe (Var, Var)
sorter (touch, light) =  do belt <- newVar
                            push <- newVar

                            set push high
                            waitUntil touch
                            set push low

                            set belt high
                              ||| forever (do waitUntil light
                                              set push high
                                              waitWhile touch
                                              waitUntil touch
                                              set push low)

                            return (belt, push)
```

Figure 5: Recipe to control brick-sorter

Acknowledgements

Finally, thanks to Jan Tobias Mühlberg for helping me get my Lava-generated C programs running on the Lego Mindstorms RCX. Without this goal, I may not have been encouraged to polish Recipe, which I initially developed over two years ago! Thanks also to Emil Axelsson, Neil Mitchell, Tobias (again), and Tom Shackell for their comments on the first draft of this article.

About the Author

Matthew Naylor is a member of the programming languages and systems group at the University of York.

References

[1] Koen Claessen and Mary Sheeran. A Tutorial on Lava. `http://www.cs.chalmers.se/~koen/Lava/tutorial.ps`.

[2] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131 (1992). `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[3] Mary Sheeran. Mary Sheeran's Web Page. `http://www.cs.chalmers.se/~ms/`.

[4] Ian Page and Wayne Luk. Compiling occam into field-programmable gate arrays. In W. Moore and W. Luk (editors), **FPGAs, Oxford Workshop on Field Programmable Logic and Applications**, pages 271–283. Abingdon EE&CS Books, 15 Harcourt Way, Abingdon OX14 1NV, UK (1991).

[5] Russell O'Connor. Assembly: Circular Programming with Recursive do. `http://www.haskell.org/sitewiki/images/1/14/TMR-Issue6.pdf`.

[6] Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In **Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)**, volume 3725 of **Lecture Notes in Computer Science**. Springer Verlag (October 2005).

[7] Jan Tobias Muehlberg and Matthew Naylor. FUN with Lego Mindstorms. `http://www.cs.york.ac.uk/plasma/talkrelated/318.pdf`.

Caml Trading: Experiences in Functional Programming on Wall Street

by Yaron Minsky ⟨yminsky@janestcapital.com⟩

Statically-typed functional programming languages such as ML and Haskell have been very successful from a number points of view. We now have the choice of a number of powerful and well-designed languages with high-quality and efficient implementations, useful libraries, and strong and friendly communities. Indeed, the rest of the computer industry has noticed some of the lovely qualities of these languages, as evidenced by the fact that the features and design of ML and Haskell have influenced the development of mainstream languages like C# and Java, as can be seen from the addition of features like generics, delegates and inner classes.

For all the success of statically typed functional languages, they are still used quite rarely in commercial settings. There are a handful of small companies that use such languages as general purpose development languages, and there are a number of large companies, such as Microsoft and Intel, that use them for specialized tasks such as hardware and software verification. But by and large, industry avoids functional programming languages, and statically typed functional languages doubly so.

Nonetheless, there are some exceptions to the rule, and in the next few pages I'd like to talk about one of them. Jane Street Capital [1] is a successful proprietary trading company that has switched over to using OCaml, a member of the ML family, as its primary development language. After a quick overview of the company, I will discuss some of what we've learned about the advantages of using a language like OCaml, as well as some of the pitfalls and limitations.

About Jane Street

Jane Street Capital is a proprietary trading company, which is to say that the business of the company is to use its capital base to trade profitably in the financial markets. The company was founded in 2000 with just three employees. Since then, Jane Street has grown to around 120 employees, and has become a global organization, with offices in New York, Chicago, Japan and London.

Trading is an intensely technological business, and getting more so by the day. Jane Street puts a large amount of effort into developing custom systems for a variety of needs: management of historical data; quantitative research; live monitoring of positions and risk; trading systems; order management and transmission systems; and so on.

As Jane Street has grown, its technology has grown and changed along with it. I came to the company part-time in 2002, and joined full-time in 2003. At the time that I joined, most of our technology was based on Excel and VBA. Excel and VBA turn out to be excellent tools for building first versions of the kinds of systems we needed, but it was clear from the start that this was not a long-term strategy.

After some experiments with using C#, we decided in 2005 to switch to using OCaml as our primary development language. Up until then, OCaml had only been used for quantitative research. Today, OCaml is behind most of our critical systems. We have over a dozen developers and researchers that use OCaml, and we're continuing to hire and grow that group.

Why OCaml?

A trading operation has stringent requirements for its software. Performance is important, both for low-latency systems that must respond quickly to the market and for analytical applications that require large amounts of computing resources; reliability is essential, since mistakes in live trading can be extraordinarily costly; rapid development is key to allowing the company to adjust to changing market conditions and emerging opportunities; and the software must be manageable, so that the complexity inherent in a sophisticated trading operation can be handled successfully.

These requirements are quite difficult to achieve at the same time, but our experience has been that using OCaml makes that task easier. Below is an overview of some of what we have come to think of as the key properties of OCaml that make it so effective for our business.

Readability

One of the easiest ways that a trading company can put itself out of business is through faulty software. For this reason, Jane Street has for some years now insisted on complete and careful code reviews for our core trading systems. In particular, a number of the partners of the firm personally review almost every line of code that goes into these systems. Our code review practices were initiated when all of our systems were written in VBA and Excel, and continue to this day. Interestingly, the partners who do the code review have minimal training in programming and computer science. For one of them, VBA was the first programming language he used in any serious way, and OCaml was the second.

Given the role of code review to our company, readability was an important metric. We eventually came to the conclusion that OCaml was considerably easier to read and review than languages such as VB, C# and Java, for a number of reasons.

Terseness Other things being equal, expressing program logic more concisely makes that logic easier to read. There are obvious limits to this; readability is not generally improved by reducing all one's function names to single characters. But OCaml allows for a pleasantly terse coding style while giving sufficient context to the reader to keep the code comprehensible.

One part of being terse is avoiding duplicated code. We try very hard to avoid saying the same thing over and over in our code, not just because it makes the code longer, but because it has been our experience that it is hard for human beings to read boilerplate as carefully as it deserves. There's a tendency to gloss over the repeats, which often leads to missed bugs. Moreover, as the code evolves, it's difficult to ensure that when one instance is updated, its siblings are updated as well. Higher-order functions and functors are powerful tools for factoring out common logic.

Immutability OCaml is not a pure language, but the default in OCaml is for immutability. Imperative code is an important part of the language, and it's a feature that we use quite a bit. But it's much easier to think about a codebase where mutability is the exception rather than the rule.

Pattern-Matching A great deal of what happens in many programs is case analysis. One of the best features of ML and similar languages is pattern-matching, which provides two things: a convenient syntax for data-directed case analysis; and a proof guaranteed by the compiler that the case analysis is exhaustive. This is useful both when writing the case analysis code in the first place, and also in alerting the programmer as to when a given case analysis needs to be updated due to the addition of some new cases.

Types The main point of code review is for the reader to put together an informal proof that the code they are reading does the right thing. Constructing such a proof is of course difficult, and we try to write our code to pack as much of the proof into the type system as possible.

There are a number of different features of the type system that are useful here. Algebraic datatypes are a powerful way of encoding basic invariants about types. One of our programming maxims is "make illegal states unrepresentable", by which we mean that if a given collection of values constitute an error, then it is better to arrange for that collection of values to be impossible to represent within the constraints of the type system. This is of course not always achievable, but algebraic datatypes (in particular variants) make it possible in many important cases.

Data-hiding using signatures can be used to encode tighter invariants than are possible with algebraic datatypes alone. The importance of abstraction in ML and Haskell is well understood, but OCaml has a nice extra feature which is the ability to declare types as **private**. As with abstract types, values of a private type can only be constructed using functions provided in the module where the type was defined. Unlike abstract types, however, the values can still be accessed and read directly, and in particular can be used in pattern-matches.

Another useful trick is phantom types. Phantom types can be used to do things like implement capability-style access control for data structures, or keep track of what kind of validation or sanity checking has been done to a given piece of data. All of this can be done in a way that the compiler proves ahead of time that the proper book-keeping is done, thus avoiding run-time errors.

No inheritance When we first tried switching over from VB to C#, one of the most disturbing features of the language to the partners who read the code was inheritance. They found it difficult to figure out which implementation of a given method was being invoked from a given call point, and therefore difficult to reason about the code.

It is worth mentioning that OCaml actually **does** support inheritance as part of its object system. That said, objects are a pretty obscure part of the language that can be ignored without much loss. Moreover, OCaml has other less confusing techniques (notably parametric polymorphism, functors and closures) for achieving what would be achieved using inheritance in an object-oriented language.

Performance

There are a lot of good things to say about the performance of the OCaml compiler: the garbage collector and allocator are extraordinarily fast; the code generation is very good; there is support for cross-module inlining; and OCaml provides a high quality foreign function interface which allows for the writing of very efficient bindings to C and C++ libraries.

The need to write our own bindings to external libraries is not just an obligation, it's also in a surprising way an advantage. There are many libraries that have only mediocre bindings to C# and Java, which means that the only way to use them really efficiently is to write in C or C++. By writing our own bindings directly to the C or C++ libraries, we are generally able to get much better performance than can be found in other managed languages.

Another important aspect of OCaml's performance is its predictability. OCaml has a reasonably simple execution model, making it easy to look at a piece of OCaml code and understand roughly how much space it is going to use and how fast it is going to run. This is particularly important in building systems that react to real-time data, where responsiveness and scalability really matter.

Modularity

Our initial systems based on Excel and VBA involved an enormous amount of cut-and-pasted code, both within a system and between different variants of the same system. When changes needed to be made, they needed to be done in one place and then manually copied to others. This is obviously a difficult and error-prone process, and something that could be improved considerably by moving to almost any language with decent support for modularity.

But our sense is that the mechanisms for providing modularity in OCaml are significantly better suited to the task than the mechanisms found in Java-like languages. As noted earlier, higher-order functions and functors are very effective tools for factoring out common logic. We feel like we have been able to separate out the different bits of logic in our systems in a way that is cleaner and easier to understand than would have been possible using a more mainstream language.

Macros

OCaml, like any language, has its limitations (which we will discuss in more detail later on). One way of mitigating the limitations of a language, as the Lisp community has long known, is to modify the language at a syntactic level. OCaml has an excellent tool for making such modifications called **camlp4** [2]. The **camlp4** tool provides a macro system which understands the OCaml AST and can be used

to add new syntax to the system or change the meaning of existing syntax. It has also been used to design domain-specific languages that translate into OCaml code.

Probably the best thing that we've done with the macro system to date is our addition of a set of macros for converting OCaml values back and forth to s-expressions. If you write the following declaration while using the s-expression macros,

```
module M = struct
 type dir = Buy | Sell with sexp

 type order =
    { sym: string;
      price: float;
      qty: int;
      dir: dir; }
 with sexp
end
```

you will end up with a module with the following signature:

```
module type M = sig
 type dir = Buy | Sell
 type order =
    { sym: string;
      price: float;
      qty: int;
      dir: dir; }

 val sexp_of_dir : dir -> Sexp.t
 val dir_of_sexp : Sexp.t -> dir
 val sexp_of_order : order -> Sexp.t
 val order_of_sexp : Sexp.t -> order
end
```

The s-expression conversion functions were written by the macros, which are triggered by the `with sexp` at the end of the type declaration. Note that a compile-time error would be triggered were the `with sexp` declaration not also appended to the `dir` type, since the functions for converting `order`s to and from s-expressions rely on the corresponding functions for the `dir` type.

The s-expression conversion functions allow you to do simple conversions back and forth between OCaml values and s-expressions, as shown below.

```
# sexp_of_order { sym = "IBM";
                  price = 38.59;
                  qty = 1200;
                  dir = Buy };;
- : Sexp.t = ((sym IBM) (price 38.59) (qty 1200) (dir Buy))
# order_of_sexp (Sexp.of_string
  "((sym IBM) (price 38.59) (qty 1200) (dir Buy))");;
- : order = { sym: "IBM"; price = 38.59; qty = 1200;
              dir = Buy; }
```

This fills an important gap in OCaml, which is the lack of generic printers and safe ways of marshaling and unmarshaling data (OCaml does have a marshaling facility, but it can lead to a segfault if a programmer guesses the type of some marshaled-in value wrong). Macros can serve this and many other roles, making it possible to extend the language without digging into the guts of the implementation.

For those who are interested, our s-expression library has been released under an open-source license, and is available from our website [3].

Hiring

One of the things we noticed very quickly when we started hiring people to program in OCaml was that the average quality of applicants we saw was much higher than what we saw when trying to hire, say, Java programmers. It's not that there aren't really talented Java programmers out there; there are. It's just that for us, finding them was much harder. The density of bright people in the OCaml community is impressive, and it shows up in hiring and when reading the OCaml mailing list and when reading the software written by people in the community. That pool of talent is probably the single best thing about OCaml from our point of view.

OCaml Pitfalls

For all of OCaml's virtues, it is hardly a perfect tool. In the following, I'd like to talk about some of the biggest issues that we've run up against while using OCaml.

Parallelism

OCaml doesn't have a concurrent garbage collector, and as a result, OCaml doesn't support truly parallel threads. Threads in OCaml are useful for a variety of purposes: overlaying computation and I/O; interfacing with foreign libraries that

require their own threads; writing responsive applications in the presence of long-running computations; and so on. But threads can not be used to take advantage of physical paralellism. This is becoming an increasingly serious limitation with the proliferation of multi-core machines.

It's not clear what the right solution is. The Caml team has been very clear that they are not interested in the complexity and performance compromises involved in building a truly concurrent garbage collector. And there are good arguments to be made that the right way of handling physical concurrency is to use multiple communicating processes. The main problem here is the lack of convenient abstractions for building such applications in OCaml. The key question is whether good enough abstractions can be provided by libraries, or whether language extensions are needed.

Generic Operations

One of the biggest annoyances of working in OCaml is the lack of generic printers, i.e. a simple general purpose mechanism for printing out human-readable representations of OCaml values. Interestingly, object-oriented languages are much better at providing this kind of feature. Generic human-readable printers are really just one class of generic operations, of which there are others, such as generic comparators and generic binary serialization algorithms. The only way of writing such generic algorithms in OCaml is through use of the macro system, as we have done with the s-expression library. While that is an adequate solution for us, OCaml could be improved for many users if some generic operations were available by default.

Objects

It has been my experience and the experience of most of the OCaml programmers I've known that the object system in OCaml is basically a mistake. The presence of objects in OCaml is perhaps best thought of as an attractive nuisance. Objects in ML should be at best a last resort. Things that other languages do with objects are in ML better achieved using features like parametric polymorphism, union types and functors. Unfortunately, programmers coming in from other languages where objects are the norm tend to use OCaml's objects as a matter of course, to their detriment. In the hundreds of thousands of lines of OCaml at Jane Street, there are only a handful of uses of objects, and most of those could be eliminated without much pain.

Programming in the Large

One of the best features a language can provide is a large ecosystem of libraries and components that make it possible to build applications quickly and easily. Languages such as Perl and Java have done an excellent job of cultivating such ecosystems, and one of the keys to doing so successfully is providing good language and tool support for programming in the large. Things that can help include: facilities for managing namespaces; tools for tracking dependencies and handling upgrades between packages; systems for searching for and fetching packages; and so on.

The most notable work in this direction in the OCaml world is **findlib** [4], an excellent package system that makes it easy to invoke installed OCaml packages; and **GODI** [5], a system for managing, downloading, upgrading and rebuilding OCaml packages. Both GODI and findlib where written by Gerd Stolpmann. GODI is still a bit rough around the edges, with an idiosyncratic user interface and sometimes temperamental behavior. One thing that would greatly help GODI or a system like it to take off would be if it was included in the standard distribution.

Compiler optimization

OCaml's compiler uses a straightforward approach to data representation and code optimization. This makes it easier to understand the time performance and space usage of a program. It is also easier to reason about performance tradeoffs when deciding between different ways of writing a piece of code, or when changing code to improve the performance of a bottleneck. Finally, the compiler itself is simpler, which contributes greatly to assurance in the correctness of generated code.

OCaml's approach to compilation has a cost, though. Programmers learn to write in a style that pleases the compiler, rather than using more readable or more clearly correct code. To get better performance they may duplicate code, expose type information, manually pack data structures, or avoid the use of higher-order functions, polymorphic types, or functors. In short, programmers may sometimes avoid the same features that make OCaml such a pleasant and safe language to program with. It is possible to adress this problem by using more aggressive optimization techniques, in particular whole-program optimization such as is used in the MLton Standard ML compiler [6].

The Cathedral and the Bazaar

The team that developed the OCaml compiler has historically adopted a cathedral-style development model for the compiler and the core libraries. Contributions are for the most part only accepted from members of the team, which largely consists

of researchers at INRIA. While the OCaml developers do an outstanding job, it's hard not to think that the system could be improved by leveraging the talents of the larger OCaml community.

One example of how OCaml could be improved by a more open development process is the standard library. The current standard library is implemented well and provides reasonable coverage, but it is missing a lot of useful functionality and has a number of well-known pitfalls (perhaps the most commented-upon is the fact that a number of the functions in the `List` module are not tail-recursive).

As a result, many people have implemented their own extensions to the standard library. There have even been projects, like ExtLib [7] that have tried to gain acceptance as "standard" extensions to the standard library. It is, however, hard to get the community to coalesce around a single such library without getting it absorbed into the standard distribution.

But the standard library is just one place where a more open development process could improve things. There is a lot of energy and talent swirling around the OCaml community, and it would be great if a way could be found to tap into that for improving the language itself.

Building a Better Caml

Our experiences with using OCaml in a commercial environment have strengthened our belief in the value that powerful programming languages can can provide to an organization nimble enough to take advantage of them. OCaml succeeds on many fronts at once: it is a beautiful and expressive language; it has an excellent implementation; and its community is second to none.

But we believe that OCaml can become better yet. It is for that reason that we founded the OCaml Summer Project [8]. Inspired by Google's Summer of Code, the OSP will fund students working on open-source OCaml projects over the summer. Our goal is to get more students excited about OCaml and functional programming in general, and also to strengthen OCaml's base of libraries and tools.

Another way we hope to encourage the growth of OCaml is by showing students and universities that statically typed functional languages are practical, real-world tools, and that there really are jobs where students can make use of these languages. Hopefully such examples will encourage the teaching, study and development of these languages.

About the Author

Yaron Minsky is a Managing Director at Jane Street Capital.

References

[1] Jane Street Capital. http://www.janestcapital.com/.

[2] Camlp4 – Reference Manual. http://caml.inria.fr/pub/docs/manual-camlp4/index.html.

[3] Sexplib. Available from:. http://www.janestcapital.com/ocaml/index.html.

[4] Findlib. http://www.ocaml-programming.de/programming/findlib.html.

[5] GODI homepage. http://godi.ocaml-programming.de/.

[6] MLton Standard ML Compiler. http://mlton.org/.

[7] OCaml ExtLib Homepage. http://ocaml-lib.sourceforge.net/.

[8] The OCaml Summer Project. http://ocamlsummerproject.com/.

Book Review: "Programming in Haskell" by Graham Hutton

by Duncan Coutts ⟨duncan.coutts@comlab.ox.ac.uk⟩

Do we need another introductory Haskell book? Is there anything new to be said or a better approach to take? Graham Hutton thinks there is. I think he is right.

Details

Title	Programming in Haskell
Author	Graham Hutton
Pages	171
Published	2007
List price	£23.99, $45.00, AUD$79.95 (paperback)
Street price	£22.79, €35,55, 380kr (paperback)
Publisher	Cambridge University Press
ISBN	978-0-521-87172-3 (paperback), 978-0-521-69269-4 (hardback)

About the Author

Dr Graham Hutton is a Reader in Computer Science at The University of Nottingham where he taught the first year functional programming course for several years. He has an impressive publishing record going back to 1990.

Scope

The book covers the Haskell language, programming techniques and the core library functions. It is a tutorial, not a reference manual. It is a pleasantly short book that covers all the essentials and nothing more.

If you don't know Haskell yet, reading this book is probably the quickest and most painless way of getting up to speed.

Audience

The book does not assume any advanced mathematics or much previous programming experience. For those with some background in imperative programming it briefly explains the essential difference in approach. It is perfectly suitable for people who have done a bit of programming and want to know what all this fuss over Haskell is about – you don't need to be a mathematician.

On the other hand it is probably not suitable as a very first introduction to computing. It is not about the very basics of solving problems using algorithms.

Style and approach

The writing style is very clear and straightforward. Examples are used consistently throughout when introducing new ideas. There are no long expanses of exposition. The pace is quite brisk but the explanations are clear and the examples keep the ideas concrete and help build an intuition.

There are exercises at the end of each chapter to help get to grips with the new ideas in the chapter. None of the exercises are essential for understanding the following chapters, so it would be quite possible to skip them on a first reading and come back to them later. There are notes at the end of each chapter giving a bit of context and references to further reading.

It does not deviate into large case studies in the early chapters, instead it uses lots of simple examples at each stage. There is one bigger example at the juncture just before introducing recursion, which is probably appropriate. At each stage we can write and run programs. Each new idea allows us to write more or better code.

The approach of this book is to try to have the reader learn the central ideas of the language as quickly as possible. Remarkably, the core ideas are all covered in the first 70 pages. While it is concise I don't think any essential explanation has been sacrificed, it is just well chosen. The learning curve is not too steep, especially if the reader is prepared to try some of the exercises at the end of each chapter. In the university lecture course upon which this book was based, the students cover roughly the same material in about four months, though it is not the very first course they take.

Comments

The distinguishing feature of this book, in my opinion, is the order and the way in which each topic is introduced. It is clear that Graham has observed how students learn this topic and from that he has carefully considered how to present the material so that it is most effectively learnt. To put it simply, it has been optimised for learning. This is a slightly subtle point perhaps, but an important one. This approach is in contrast to merely presenting the material in a logical order or in a way that reflects the author's philosophy of functional programming.

For example, I think he is right to introduce types and particularly overloading so early. Similarly, I would say he has made the right decision to defer treatment of lazy evaluation and equational reasoning to the final chapters. IO is presented about halfway through the book, just after the core pure subset of the language.

The presentation can mostly be characterised as a top-down but Graham does explain at various points how newly introduced constructs can be used to explain or give an alternative definition for things that were introduced previously. This should satisfy those readers who understand best when they can see how everything is built from a small set of primitives, but for those who don't learn that way it does not force that approach upon them. Graham's approach means that with each new idea we are only taking small steps away from the familiar and the examples allows us to keep an intuition. This contrasts with a bottom-up approach of presenting ideas abstractly and then showing how that allows one to build the familiar.

Calculation

Some presentations of functional programming heavily emphasise the notion that programs are mathematical objects and can be manipulated, optimised and reasoned about using equational reasoning. While this is certainly something we want to tell people, I usually find that the majority of the audience do not naturally warm to the mathematical approach to learning. To many people it seems hard and off-putting.

I believe it is easier for most readers to grasp these ideas after they have got the basics of the language. So it is not that calculational methods are not important, it is just that they are not necessarily the easiest way to learn. For most students, the intuition drives the calculation, not the other way around. It is not obvious in the earlier stages of learning why this “calculation stuff” might ever be useful. There's the danger that readers and students assume it's just something academics do so that they have some hard questions to put in exams. I am convinced it is better not to clutter the critical early phases of learning this new language with too much formalism but to explain that later at a more relaxed pace.

Laziness

Similarly, I think it is perfectly OK to defer explaining lazy evaluation. While it is a distinguishing feature of Haskell, you can get quite a long way in learning functional programming before strict or lazy evaluation matters at all. From a practical point of view, the advantage of laziness is mainly that it enable us write programs in a nicer style, which is something that is easier to appreciate once we can read on understand any Haskell programs at all!

IO

IO is introduced with little fuss. Monadic IO may have been remarkable when it was first invented but now it is right to present it reasonably early and as fairly straightforward. Otherwise, the danger is that people are put off by the assumption that IO is hard. On the other hand it is not necessarily great to put monadic effects at the centre of the whole presentation. IO is just something we have to be able to do, it is not an area in which Haskell is particularly radical (except in the way that it cleanly separates it).

A clever aspect of the presentation of IO is that the previous chapter on parsers already introduces the notion of sequencing and the `do` notation (without ever using the term "monad"). So when these concepts get re-used for IO they are already familiar, which makes it clear that this funny `do` syntax is not peculiar to doing IO. It neatly avoids the trap of thinking that Haskell has a special imperative language or that the `do` notation is only for programming with side effects.

Recursion

Graham introduces us to lists, many standard functions on lists and list comprehensions before recursion. This allows the reader to gain confidence with some realistic code before tackling recursion. Then immediately after recursion, we move onto higher order functions. Hopefully this will help readers to not get "stuck" on doing everything with primitive recursion, which is something I see with some of my own students and a phenomenon that Simon Thompson tried to address in the second edition of his book [1].

Chapter summaries

1. The introduction starts with what a function is, what people understand by "functional programming" and how that contrasts with imperative programming styles. It gives a brief overview of the language features that the

remaining chapters cover, a brief historical context of the language and a quick taste of what Haskell code and evaluation look like.

2. After the introduction, the book starts in earnest in chapter 2 by showing some examples of simple expressions on numbers and lists and how to evaluate these using the Hugs interactive Haskell interpreter. It then covers the basic syntactical conventions, in particular the notation for function application which is something that can easily trip up people who are familiar with the convention of other programming languages.

3. The next topic is types: basic types, tuples and lists. It then covers the important topic of function types and currying. Overloading is introduced very early. It then goes on to present the important type classes: `Eq`, `Ord`, `Show`, `Read` and `Num`.

4. Having been armed with the basics of expressions and several useful functions on numbers and lists, the next chapter explains how to build more interesting expressions by defining functions, including conditionals, guards and patterns. After introducing lambda expressions the earlier concepts of function definition and currying are revisited in the light of this primitive.

5. Before getting on to the more tricky concept of recursion, list comprehensions are introduced. Along with the other ideas from the previous chapters we work through a non-trivial example problem: cracking a simple Caesar cipher.

6. The following two chapters cover recursion and higher order functions. Recursion is explained clearly before moving on to examples on lists and examples with multiple arguments, multiple recursive calls and mutual recursion. Since recursion can be a stumbling point for some, the chapter concludes with advice and a five point strategy for making recursive functions. To reinforce the strategy it is demonstrated by working through the process on several examples.

7. In my experience teaching Haskell I have noticed that having discovered and mastered recursion, some students seem to get stuck and write all their code in a primitive recursive style rather than moving on to take advantage of Haskell's ability to abstract. By presenting higher order functions immediately after recursion, readers of this book should be able to avoid a first order fate. It introduces the idea of functions as arguments and explains `map` and `filter`, as always, with plenty of examples. We are introduced next to `foldr` and `foldl` and shown how they can be used to capture the pattern of recursion used in many of the simple recursive functions defined previously.

8. The first major example after covering the core language is a parser for a simple expression language. It shows off the ability to create abstractions and, importantly, it introduces the `(>>=)` sequencing operator and the `do` notation.

9. Sequencing and the `do` notation is immediately re-used in the next chapter in explaining interactive programs and IO. It is illustrated with a console calculator and game of life program.

10. The first of the more advanced topics is on how to declare new types, first type aliases and then new structured data types. It covers recursive data types including trees and a bigger example using boolean expressions. Significantly, it covers how to make new types instances of standard type classes such as `Eq`. It gives examples of how to define instances using both `instance` declarations and the `deriving` mechanism. It also comes back to sequencing, showing how the previous examples of parsers and IO fit into the `Monad` class, which makes it clear when the `do` notation can be used.

11. The next chapter is dedicated to a case study working though Graham's famous "countdown" program. It covers how to model the problem, starts with a simple brute-force search algorithm and then refines it in several stages to make it much faster. The solution makes good use of the features introduced earlier like list comprehensions, standard list functions, recursion and tree data types.

12. The penultimate chapter covers lazy evaluation. It compares 'inner', 'outer' and lazy evaluation strategies on the grounds of termination and reduction count. It continues with examples of how it allows infinite structures and helps modular programming. It also covers strict application (mentioning `foldl'`) and when it is useful to control space use.

13. The final chapter is on reasoning about programs. It starts with equational reasoning and continues with inductive proofs on numbers and lists. In addition to proving properties about programs it shows how the equational approach can be used to optimise programs by deriving an efficient implementation from a simple initial implementation. It uses the nice example of eliminating expensive append operations by abstracting over the tail of the list, obtaining a rule and applying the rule to get a new efficient version. It's a convincing advertisement for the formal viewpoint: it gives a huge speedup and a final implementation that we would not necessarily have thought of directly.

Criticism

Apart from annoyances noted below, personally, I find little to complain about.

It is possible that instructors who consider using the book in a lecture course might worry that the examples are not sufficiently exciting or that there may be insufficient motivation for why we might want to do things like parsing. I think the examples are about right in size for illustrating the ideas, bigger more exciting programming tasks can always be set for practical labs.

One may complain that the book does not cover enough, that we might prefer more topics on more advanced material. The topics selected cover an essential minimum, which seems like an appropriate place to cut. Probably the topics that will be most missed are operational issues like space and time complexity. There is certainly a gap in the market for an intermediate Haskell book that starts where this one leaves off.

One might also complain that not every aspect of the language is covered. It uses the common declaration style with named functions and `where` clauses, it does not cover `let` expressions and `case` expressions are only briefly mentioned. There are various other lesser used corners of the language not mentioned. The book is very clearly presented as not being a language reference manual so it is hard to make this kind of criticism stick.

Annoyances

One minor thing to watch out for is that while the code examples are beautifully typeset using lhs2TEX[2] they cannot be directly typed into Hugs because many of the operators are typeset as symbols that do not correspond to the ASCII syntax that one has to use. The section introducing the Hugs system and the prelude references an ASCII translation table in appendix B. I would have preferred if this table were just included inline, like the table of hugs commands. I managed to miss the reference on a first reading.

Unfortunately, a few of the code examples rely on bugs in an old version of the Hugs system that have since been corrected. For example, Hugs previously exported several non-standard and `Char` module functions through the `Prelude`. After Hugs was fixed the three other books published around that time suffered from this problem. It is rather a shame that this new book suffers the same problem. While the solution in each case is very simple, e.g. just import the `Char` module, these kind of minor problems can be quite bewildering to new users. An errata with all the details is available online [3].

A slight quibble is that the introduction describes the Haskell98 standard as being “recently published”. It is true that the final version of the language report was only published in 2003, however an 18 year old university student is likely to

claim that anything that is apparently nearly 10 years old is archaic. For marketing purposes it is probably better to describe Haskell98 as the stable version of the language. There is an obvious tension in wanting to make it clear to potential readers that the material has not been superseded by changes in the language, but at the same time not give the impression that they would be learning something that is dead.

Comparison with other books

There are three main other Haskell books available in English. There are also recent books in Portuguese, Spanish, Romanian, Russian and Japanese [4].

It is an interesting sign of the times perhaps that the book is titled simply *Programming in Haskell* rather than the title having to relate Haskell to functional programming in general.

Introduction to Functional Programming using Haskell

The Introduction to Functional Programming using Haskell [5] is the text for the mathematically minded. It aims to teach the concepts of functional programming. To be concrete it uses Haskell. The emphasis is on constructing functions by calculation. It introduces equational reasoning very early where as *Programming in Haskell* delays that topic to the final chapter. Conversely, IO is deferred to the final chapter where as *Programming in Haskell* covers it much earlier.

It is a much more substantial book (448 pages) and covers topics like trees, abstract data types and efficiency/complexity.

The Haskell School of Expression: Learning Functional Programming through Multimedia

In a complete departure from the mathematical approach, *The Haskell School of Expression* [6] takes the somewhat radical approach of using multimedia as the main motivating examples throughout the book. This is an approach that appeals to some, but not to others.

It starts at a somewhat more advanced level than *Programming in Haskell* and goes on to cover modules, higher order and polymorphic functions, abstract data types, IO and various domain-specific embedded languages.

Again, it is considerably longer at 382 pages.

Haskell: The Craft of Functional Programming

Simon Thompson's *Haskell: The Craft of Functional Programming* [1] sits somewhere between the other two in terms of style. It aims to teach Haskell and the functional programming approach. As of the second edition it uses more examples and more of a top down style, where for example, functions are used before going into the details of their implementation.

It covers modules reasonably early where as *Programming in Haskell* does not cover them at all. It introduces proofs and equational reasoning much earlier. It is much less compact in its explanations than *Programming in Haskell* and is a much larger book (512 pages). It does cover several more advanced topics such as algebraic data types and time and space behaviour. It defers IO to a much later chapter than *Programming in Haskell*.

Conclusion

In my opinion, this book is the best introduction to Haskell available. There are many paths towards becoming comfortable and competent with the language but I think studying this book is the quickest path.

I urge readers of this magazine to recommend *Programming in Haskell* to anyone who has been thinking about learning the language.

The book is available now from online [7] and high street bookstores.

Acknowledgements

Thanks to The Monad.Reader editor Wouter Swierstra for arranging a review copy of the book for me. Thanks to Lennart Kolmodin and other denizens of `#haskell` for helpful comments on drafts of this article.

About the Reviewer

Duncan Coutts has a BA in Computation from the University of Oxford. He is still there trying to finish his PhD. He has seen Haskell teaching from both sides: as an undergraduate he attended Richard Bird's first year functional programming course; as a teaching assistant he now runs the practicals for the same course.

References

[1] Simon Thompson. **Haskell: The Craft of Functional Programming**. Addison Wesley, 2nd edition (1999).

[2] Andres Löh. lhs2TeX. `http://www.informatik.uni-bonn.de/~loeh/lhs2tex/`.

[3] Errata. `http://www.cs.nott.ac.uk/~gmh/errata.html`.

[4] `http://haskell.org/haskellwiki/Books_and_tutorials#Textbooks`.

[5] Richard Bird. **Introduction to Functional Programming using Haskell**. Prentice Hall Press, 2nd edition (1998).

[6] Paul Hudak. **The Haskell School of Expression: Learning Functional Programming through Multimedia**. Cambridge University Press (2000).

[7] Programming in Haskell website. `http://www.cs.nott.ac.uk/~gmh/book.html`.

Yhc.Core – from Haskell to Core

by Dimitry Golubovsky ⟨golubovsky@gmail.com⟩
and Neil Mitchell ⟨ndm@cs.york.ac.uk⟩
and Matthew Naylor ⟨mfn@cs.york.ac.uk⟩

The Yhc compiler is a hot-bed of new and interesting ideas. We present Yhc.Core – one of the most popular libraries from Yhc. We describe what we think makes Yhc.Core special, and how people have used it in various projects including an evaluator, and a Javascript code generator.

What is Yhc Core?

The York Haskell Compiler (Yhc) [1] is a fork of the nhc98 compiler [2], started by Tom Shackell. The initial goals included increased portability, a platform independent bytecode, integrated Hat [3] support and generally being a cleaner code base to work with. Yhc has been going for a number of years, and now compiles and runs almost all Haskell 98 programs and has basic FFI support – the main thing missing is the Haskell base library.

Yhc.Core is one of our most successful libraries to date. The original nhc compiler used an intermediate core language called PosLambda – a basic lambda calculus extended with positional information. The language was neither a subset nor a superset of Haskell. In particular there were unusual constructs and all names were stored in a symbol table. There was also no defined external representation.

When one of the authors required a core Haskell language, after evaluating GHC Core [4], it was decided that PosLambda was closest to what was desired but required substantial clean up. Rather than attempt to change the PosLambda language, a task that would have been decidedly painful, we chose instead to write a Core language from scratch. When designing our Core language, we took ideas from both PosLambda and GHC Core, aiming for something as simple as possible. Due to the similarities to PosLambda we have written a translator from our Core language to PosLambda, which is part of the Yhc compiler.

Our idealised Core language differs from GHC Core in a number of ways:

- Untyped – originally this was a restriction of PosLambda, but now we see this as a feature, although not everyone agrees.
- Syntactically a subset of Haskell.
- Minimal name mangling.

All these features combine to create a Core language which resembles Haskell much more than Core languages in other Haskell compilers. As a result, most Haskell programmers can feel at home with relatively little effort.

By keeping a much simpler Core language, it is less effort to learn, and the number of projects depending on it has grown rapidly. We have tried to add facilities to the libraries for common tasks, rather than duplicating them separately in projects. As a result the Core library now has facilities for dealing with primitives, removing recursive lets, reachability analysis, strictness analysis, simplification, inlining and more.

One of the first features we added to Core was *whole program linking* – any Haskell program, regardless of the number of modules, can be collapsed into one single Yhc.Core module. While this breaks separate compilation, it simplifies many types of analysis and transformation. If a such an analysis turns out to be successful then breaking the dependence on whole program compilation is a worthy goal – but this approach allows developers to pay that cost only when it is needed.

A Small Example

To give a flavour of what Core looks like, it is easiest to start with a small program:

```
head2 (x:xs) = x

map2 f [] = []
map2 f (x:xs) = f x : map2 f xs

test x = map2 head2 x
```

Compiling with `yhc -showcore Sample.hs` generates:

```
Sample.head2 v220 =
    case v220 of
        (:) v221 v222 -> v221
        _ -> Prelude.error Sample._LAMBDA228
```

```
Sample._LAMBDA228 =
    "Sample: Pattern match failure in function at 9:1-9:15."

Sample.map2 v223 v224 =
    case v224 of
        [] -> []
        (:) v225 v226 -> (:) (v223 v225) (Sample.map2 v223 v226)

Sample.test v227 = Sample.map2 Sample.head2 v227
```

The generated Core can be treated as a subset of Haskell, with many restrictions:

- Case statements only examine their outermost constructor
- No type classes
- No `where` statements
- Only top-level functions.
- All names are fully qualified
- All constructors and primitives are fully applied

Yhc.Core.Overlay

We provide many library functions to operate on Core, but one of our most unusual features is the *overlay* concept. Overlays specify modifications to be made to a piece of code – which functions should be replaced, which ones inserted, which data structures modified. By combining a Core file with an overlay, modifications can be made after translation from Haskell to Core. This idea originated in the Mozilla project [5], and is used successfully to enable extensions in Firefox, and elsewhere throughout their platform.

To take a simple example, in Haskell there are two common definitions for `reverse`:

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

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

The first definition uses an accumulator, and takes $O(n)$. The second definition requires $O(n^2)$, as the tail element is appended onto the whole list. Clearly a Haskell compiler should pick the first variant. However, a program analysis tool may wish to use the second variant as it may present fewer analysis challenges. The overlay mechanism allows this to be done easily.

The first step is to write an overlay file:

```
global_Prelude'_reverse []     = []
global_Prelude'_reverse (x:xs) = global_Prelude'_reverse xs ++ [x]
```

This Overlay file contains a list of functions whose definitions we would like to replace. Any function that previously called `Prelude.reverse` will now invoke this new copy. For a program to insert an overlay, both Haskell files need to be compiled to Core, then the `overlay` function is called.

But we need not stop at simply replacing the `reverse` function. Yhc defines an IO type as a function over the `World` type, but for some applications this may not be appropriate. We can redefine IO as:

```
data IO a = IO a

global_Monad'_IO'_return a = IO a
global_Monad'_IO'_'gt'gt (IO a) b = b
global_Monad'_IO'_'gt'gt'eq (IO a) f = f a
global_YHC'_Internal'_unsafePerformIO (IO a) = a
```

The Overlay mechanism supports escape characters – `'gt` is the `>` character – allowing us to replace the bind and return methods.

We have found that with Overlays a compiler can be customized for many different tasks, without causing conflicts. With one code base, we can allow different programs to modify the libraries to suit their needs. Taking the example of `Int` addition, there are at least three different implementations in use: Javascript native numbers, binary arithmetic on a Haskell data type and abstract interpretation.

Semantics of Yhc Core

In this section an evaluator for Yhc Core programs is presented in the form of a literate Haskell program. The aim is to define the informal semantics of Core programs while demonstrating a full, albeit simple, application of the `Yhc.Core` library.

```
module Main where

import Yhc.Core
import System
import Monad
```

Our evaluator is based around the function `whnf` that takes a Core program (of type `Core`) along with a Core expression (of type `CoreExpr`) and reduces that expression until it has the form of:

- a data constructor with unevaluated arguments, or
- an unapplied lambda expression.

In general, data values in Haskell are tree-shaped. The function `whnf` is often said to "reduce an expression to head normal form" because it reveals the head (or root) of a value's tree and no more. Stricly speaking, when the result of reduction could be a functional value (i.e. a lambda expression), and the body of that lambda is left unevaluated, then the result is said to be in "weak head normal form" – this explains the strange acronym.

The type of `whnf` is:

```
whnf :: Core -> CoreExpr -> CoreExpr
```

Defining it is a process of taking each kind of Core expression in turn, and asking "how do I reduce this to weak head normal form?" As usual, it makes sense to define the base cases first, namely constructors and lambda expressions:

```
whnf p (CoreCon c)              =  CoreCon c
whnf p (CoreApp (CoreCon c) as) =  CoreApp (CoreCon c) as
whnf p (CoreLam (v:vs) e)       =  CoreLam (v:vs) e
```

Notice that a constructor may take one of two forms: stand-alone with no arguments, or as function application to a list of arguments. Also, because of the way our evaluator is designed, we may encounter lambda expressions with no arguments. Hence, only lambdas with arguments represent a base-case. For the no-arguments case, we just shift the focus of reduction to the body:

```
whnf p (CoreLam [] e) =  whnf p e
```

Currently, lambda expressions do not occur in the Core output of Yhc. They are part of the Core syntax because they are useful conceptually, particularly when maniplating (and evaluating) higher-order functions.

Moving on to case-expressions, we first reduce the case subject, then match it against each pattern in turn, and finally reduce the body of the chosen alternative. In Core, we can safely assume that patterns are at most one constructor deep, so reduction of the subject to WHNF is sufficient.

```
whnf p (CoreCase e as) =  whnf p (match (whnf p e) as)
```

We defer the definition of `match` for the moment.

To reduce a let-expression, we substitute the let-bindings in the body of the let. This is easily done using the Core function `replaceFreeVars`. Like in Haskell, let-expressions in Core are recursive, but before evaluating a Core program we transform them all to non-recursive lets (see below). Notice that we are in no way trying to preserve the sharing implied by let-expressions, although we have done so in more complex variants of the evaluator. Strictly speaking, Haskell evaluators are not obliged to implement sharing – this is why it is more correct to term Haskell non-strict than lazy.

```
whnf p (CoreLet bs e) =  whnf p (replaceFreeVars bs e)
```

When we ecounter an unapplied function we call `coreFunc` to lookup its definition (i.e. its arguments and its right-hand-side), and construct a corresponding lambda expression:

```
whnf p (CoreFun f)     =  whnf p (CoreLam bs body)
  where
    CoreFunc _ bs body =  coreFunc p f
```

This means that when reducing function applications, we know that reduction of the function part will yield a lambda:

```
whnf p (CoreApp f [])     = whnf p f
whnf p (CoreApp f (a:as)) = whnf p (CoreLet [(b,a)]
                                    (CoreApp (CoreLam bs e) as))
  where
    CoreLam (b:bs) e      = whnf p f
```

Core programs may contain information about where definitions originally occurred in the Haskell source. We just ignore these:

```
whnf p (CorePos _ e) =  whnf p e
```

And the final, fall-through case covers primitive literals and functions which we are not concerned with here:

```
whnf p e =  e
```

Now, for the sake of completeness, we return to our `match` function. It takes the evaluated case subject and tries to match it against each case-alternative (a pattern-expression pair) in order of appearance. We use the "failure by a list of successes" technique [6] to model the fact that matching may fail.

```
type Alt    =  (CoreExpr, CoreExpr)

match       :: CoreExpr -> [Alt] -> CoreExpr
match e as  =  head (concatMap (try e) as)
```

Before defining `try`, it is useful to have a function that turns the two possible constructor forms into a single normal form. This greatly reduces the number of cases we need to consider in the definition of `try`.

```
norm             :: CoreExpr -> CoreExpr
norm (CoreCon c) =  CoreApp (CoreCon c) []
norm x           =  x
```

Hopefully, by now the definition of `try` will be self-explanatory:

```
try                             :: CoreExpr -> Alt -> [CoreExpr]
try e (pat, rhs)                =
  case (norm pat, norm e) of
    (CoreApp (CoreCon f) as, CoreApp (CoreCon g) bs)
       | f == g                 -> [CoreLet (zip (vars as) bs) rhs]
    (CoreVar v, e)              -> [CoreLet [(v, e)] rhs]
    _                           -> []
  where
    vars                        =  map fromCoreVar
```

This completes the definition of `whnf`. However, we would like to be able to fully evaluate expressions – to what we simply call "normal form" – so that the resulting value's tree is computed in its entirety. Our `nf` function repeatedly applies `whnf` at progressively deeper nodes in the growing tree:

```
nf                          :: Core -> CoreExpr -> CoreExpr
nf p e                      =
  case whnf p e of
    CoreCon c               -> CoreCon c
    CoreApp (CoreCon c) es -> CoreApp (CoreCon c) (map (nf p) es)
    e                       -> e
```

All that remains is to turn our evaluator into a program by giving it a sensible `main` function. We first load the Core file using `loadCore` and then apply `removeRecursiveLet`, as discussed ealier, before evaluating the expression `CoreFun "main"` to normal form and printing it.

```
main :: IO ()
main =   liftM head getArgs
     >>= liftM removeRecursiveLet . loadCore
     >>= print . flip nf (CoreFun "main")
```

In future we hope to use a variant of this evaluator (with sharing) in a property-based testing framework. This will let us check that various program analyses and transformations that we have developed are semantics-preserving. As part of another project, we have sucessfully extended the evaluator to support various functional-logic evaluation strategies.

Javascript backend

The Javascript backend is a unique feature of Yhc. The idea to write a converter from Haskell to Javascript, enabling the execution of Haskell programs in a web browser, has been floating around for some time [7, 8, 9]. Many people expressed interest in such feature, but no practical implementation has emerged.

Initial goals of this subproject were:

- To develop a program that converts the Yhc Core to Javascript, thus making it possible to execute arbitrary Haskell code within a web browser.
- To develop an unsafe interface layer for quick access to Javascript objects with ability to wrap arbitrary Javascript code into a Haskell-callable function.
- To develop a typesafe interface layer on top of the unsafe interface layer for access to the Document Object Model (DOM) available to Javascript executed in a web browser.
- To develop or adopt an existing GUI library or toolkit working on top of the typesafe DOM layer for actual development of client-side Web applications.

General concepts

The Javascript backend converts a linked and optimized Yhc Core file into a piece of Javascript code to be embedded in a XHTML document. The Javascript code generator tries to translate Core expressions to Javascript expressions one-to-one with minor optimizations of its own, taking advantage of the Javascript capability to pass functions around as values.

Three kinds of functions are present in the Javascript backend:

- Unsafe functions that embed pieces of Javascript directly into the generated code: these functions pay no respect to types of arguments passed, and may force evaluation of their arguments if needed.
- Typesafe wrappers that provide type signatures for unsafe functions. Such wrappers are either handwritten, or automatically generated from external interface specifications (such as the DOM interface).
- Regular library functions. These either come unmodified from the standard Yhc packages, or are substituted by the Javascript backend using the Core overlay technique. An example of such a function is the `toUpper` function which is hooked up to the Javascript implementation supporting Unicode (the original library function currently works correctly only for the Latin1 range of characters).

Unsafe interfaces

The core part of unsafe interface to Javascript (or, in other words, Javascript FFI) is a pseudo-function `unsafeJS`. The function has a type signature:

```
foreign import primitive unsafeJS :: String -> a
```

The input is a `String`, but the type of the return value does not matter: the function itself is never executed. Its applications are detected by the Yhc Core to Javascript conversion program and dealt with at the time of Javascript generation.

The `unsafeJS` function should be called with a string literal. Both explicitly coded (with (:)) lists of characters and the concatenation of two or more strings will cause the converter to report an error.

A valid example of using `unsafeJS` is shown below:

```
global_YHC'_Primitive'_primIntSignum :: Int -> Int
global_YHC'_Primitive'_primIntSignum a = unsafeJS
  "var ea = exprEval(a); if (ea>0) return 1; else if (ea<0)
   return -1; else return 0;"
```

This is a Javascript overlay (in the sense that it overlays the default Prelude definition of the `signum` function) of a function that returns sign of an `Int` value. The string literal given to `unsafeJS` is the Javascript code to be wrapped. Below is the Javascript representation of this function found in generated code.

```
strIdx["F_hy"] = "YHC.Primitive.primIntSignum";
...
var F_hy=new HSFun("F_hy", 1, function(a){
    var ea = exprEval(a); if (ea>0) return 1;
    else if (ea<0) return -1; else return 0;});
```

Typesafe wrappers

These functions add type safety on top of unsafe interface to Javascript. Sometimes they are defined within the same module as unsafe interfaces themselves, thus avoiding the exposure of unsafe interfaces to programmers.

An example of a handwritten wrapper is a function to create a new `JSRef`: a mechanism similar to Haskell's `IORef`, but specific to Javascript.

```
data JSRef a

newJSRef :: a -> CPS b (JSRef a)

newJSRef a = toCPE (newJSRef' a)
newJSRef' a = unsafeJS "return {_val:a};"
```

Technically, a `JSRef` is a Javascript object with a property named *_val* that holds a persistent reference to some value. On the unsafe side, invoking a constructor for such an object would be sufficient. It is however desired that:

- calls to functions creating such persistent references are properly sequenced with calls to functions using these references, and
- the type of values referred to are known to the Haskell compiler.

The unsafe part is implemented by the function `newJSRef'` which merely calls `unsafeJS` with a proper Javascript constructor. The wrapper part `newJSRef` wraps the unsafe function into a CPS-style function, and is given a proper type signature, so more errors can be caught at compile time.

In some cases, such typesafe wrappers may be generated automatically, using some external interface specifications provided by third parties for their APIs. The W3C DOM interface is one such API. For instance, this piece of OMG IDL:

```
interface Text : CharacterData {
  Text               splitText(in unsigned long offset)
                                      raises(DOMException);
};
```

is converted into:

```
data TText = TText
...
instance CText TText
instance CCharacterData TText
instance CNode TText
...
splitText :: (CText this, CText zz) => this -> Int -> CPS c zz
splitText a b = toCPE (splitText' a b)
splitText' a b
  = unsafeJS "return((exprEval(a)).splitText(exprEval(b)));"
```

These instances and signatures give the Haskell compiler better control over this function's (initially type-agnostic) arguments.

Usage of Continuation Passing Style

Initially we attempted to build a monadic framework. The `JS` monad was designed to play the same role as the `IO` monad plays in "regular" Haskell programming. There were, however, arguments in favor of using Continuation Passing Style (CPS) [10]:

- CPS involves less overhead as each expression passes its continuation itself, instead of `bind` which takes the expression and invokes the continuation
- CPS results in Javascript patterns that are easy to detect and optimize, although this is a not implemented yet.
- The Fudgets [11] GUI library internals are written in CPS, so taking CPS as general approach to programming is believed to make adoption of Fudgets easier.

Integration with DOM

The Web Consortium [12] provides OMG IDL [13] files to describe the API to use with the Document Object Model (DOM) [14]. A utility was designed, based on

HaskellDirect [15], to parse these files and convert them to set of Haskell modules. The way interface inheritance is reflected differs from HaskellDirect: in HaskellDirect this was achieved by declaration of "nested" algebraic data types. The Javascript backend takes advantage of Haskell typeclasses – representing DOM types with phantom types, and declaring them instances of appropriate classes.

Unicode support

Despite the fact that all modern Web browsers support Unicode, this is not the case with Javascript: no access to Unicode characters' properties is provided. At the same time it is desirable for a Haskell application running in a browser to have access to such information. The approach used is the same as in Hugs [16] and GHC [17]: the Unicode characters database file from the Unicode Consortium [18] was converted into a set of Javascript arrays, each array entry represents a range of character code values, or a case conversion rule for a range. For this implementation, Unicode support is limited to the character category, and simple case conversions. First, a range is found by looking up the character code; then the character category and case conversion distances, i.e. values to add to character code to convert between upper and lower cases, are retrieved from the range entry. The whole set of arrays adds about 70 kilobytes to the web page size, if embedded inside a `<script>` tag.

Using the Core overlay technique, Haskell character functions (like `toUpper`, `isAlpha`, etc.) were hooked up to the Javascript implementations supporting Unicode. This did not result in noticeable slowdowns; some browsers even showed a minor speedup in functions like `read::String -> Int` that perform large amounts of string manipulations.

Examples of code generation

The two examples below show conversion of real-life functions from Haskell, via Core, to Javascript. It is important to mention that as the Javascript code generator evolves, the resultant code may do so too.

Example 1. Taking a function from a Roman Numeral package:

```
fromRoman = foldr fromNumeral 0 . maxmunch . map toUpper
```

When converted to Yhc Core this becomes:

```
Roman.fromRoman =
    Prelude._LAMBDA27191
      (Prelude._LAMBDA27191
```

```
         (Prelude.map Data.Char.toUpper)
         (Roman.maxmunch Prelude.Prelude.Num.Prelude.Int))
      (Prelude.foldr
         (Roman.fromNumeral
            Prelude.Prelude.Num.Prelude.Int
            Prelude.Prelude.Ord.Prelude.Int)
         0)

Prelude._LAMBDA27191 v22167 v22166 v2007 = v22166 (v22167 v2007)
```

The introduced `LAMBDA` is similar to the composition function `(.)`, only with inverted order of application: `_LAMBDA27191 f g x = g (f x)`

When convert to Javascript we get:

```
/* fromRoman, code was formatted manually */

var F_g8=new HSFun("F_g8", 0, function(){
  return (F_e9)._ap([(F_e9)._ap([new HSFun("F_gz", 0,
    function(){
      return (F_gz)._ap([F_Z]);
    }), new HSFun("F_g9", 0,
    function(){
      return (F_g9)._ap([F_dC]);
    })]), new HSFun("F_gp", 0,
    function(){
      return (F_gp)._ap([new HSFun("F_g7", 0,
        function(){
          return (F_g7)._ap([F_dC, F_d1]);
        }), 0]);
    })]);
});

/* _LAMBDA27191 */

var F_e9=new HSFun("F_e9", 3, function(_b3, _b2, _b0)
         {return (_b2)._ap([(_b3)._ap([_b0])]);});
```

During the conversion to Javascript, all identifiers found in Yhc Core are renamed to much shorter ones consisting only of alphanumeric characters and thus surely valid for Javascript (identifiers in Yhc Core often are very long, or contain special characters, etc.)

While it is hard to understand anything from the Javascript for the `fromRoman` function (other than that the Javascript backend already makes a good obfuscator), something may be seen in the Javascript for the composition function. It builds an application of its first argument to the third, and then the application of the second to the previous application, and returns the latter.

Example 2. An example of a function whose implementation was replaced via the Overlay technique is the `isSpace` function:

```
global_Data'_Char'_isSpace = f . ord
  where f a = unsafeJS "return uIsSpace(exprEval(a));"
```

Translated to Core:

```
Data.Char.isSpace =
    Prelude._LAMBDA27191
      Data._CharNumeric.ord
      StdOverlay.StdOverlay.Prelude.287.f
```

Translated to Javascript:

```
var F_W=new HSFun("F_W", 0, function(){
    return (F_e9)._ap([F_bh, F_hk]);});
```

In the Haskell code, the `global_Data'_Char'_isSpace` identifier tells the Core Overlay engine that the function with qualified name `Data.Char.isSpace` is to be replaced with a new implementation. In Yhc Core, the previously reviewed reversed composition function can be seen which composes the `ord` function, and an inner function that actually invokes the Javascript function which in turn performs the Unicode properties lookup for a given character numeric code.

Browser compatibility

Our implementation is compatibile with major web browsers such as Netscape, Mozilla, Firefox, Microsoft Internet Explorer, and Opera. Simple programs such as echoing an input string, Roman to Decimal number conversion, a simple counter with increment and decrement buttons, etc. were used to test browser compatibility. Some of these programs accessed the DOM directly, other used a subset of the Fudgets API.

Opera generally showed the fastest execution of Javascript, but no representative data sample has been collected yet. Microsoft Internet Explorer showed memory leaks when closures over DOM elements were involved in versions up to 6, but

one source reported that in version 7 memory leaks were greatly reduced, and the speed of Javascript execution increased. The XML HTTP request technique was positively tested on Netscape and Microsoft Internet Explorer. Konqueror has never been tested. Compatibility with Safari has not been achieved so far: attempts to execute any of the test programs mentioned above resulted in obscure error messages delivering no information about the nature of incompatibility.

Future plan: Fudgets

We plan to port some portion of Fudgets, so it becomes possible to write Web applications using this library. Several experiments showed that the Stream Processors (SP), and some parts of Fudget Kernel layers work within a Javascript application. More problems are expected with porting the toplevel widgets due to differences in many concepts between a Web browser and X Windows, for which the Fudgets library was originally developed.

Conclusion and Future Goals

Yhc.Core is a library which has been used by quite a few people, it is still evolving – moving useful operations from the individual programs into the common library. There are at least five additional projects we are aware of that make use of this library including: static checkers, program validation, Java generation, optimisation and user hinting. We are always looking for more users – we hope that by providing the dull stuff (interfacing to Haskell), others will provide the cool applications. If you are tempted to use Core, we are most interested to know: `yhc@haskell.org`.

The original structure of nhc was as one big set of modules – some were broadly divided into type checking/parsing etc, but the overall structure and grouping was weaker than in other compilers. One of our first actions was to split up the code into hierarchical modules, introducing Type.*, Parse.* etc to better divide the components. We hope that some of these other sections can be repositioned as libraries, allowing others to make use of them. This approach attracts us, and we see this as the future direction of our compiler.

Acknowledgements

We would like to thank Mike Dodds and Tom Shackell for making comments on earlier drafts of this article.

Thanks also to everyone who has submitted a patch, become a buildbot, reported bugs or done anything else to benefit the Yhc project. We've put together a list

of most of the people (if we've missed you, we apologise, but definitely value your contribution!)

Andrew Wilkinson, Bernie Pope, Bob Davie, Brian Alliet, Christopher Lane Hinson, Dimitry Golubovsky, Gabor Greif, Goetz Isenmann, Ian Lynagh, Isaac Dupree, Kartik Vaddadi, Krasimir Angelov, Malcolm Wallace, Michal Palka, Mike Dodds, Neil Mitchell, Robert Dockins, Samuel Bronson, Simon Marlow, Stefan O'Rear, Thorkil Naur, Tom Shackell, Twan van Laarhoven

About the Authors

Dimitry Golubovsky is a software engineer, originally from St-Petersburg, Russia. He is currently working in the United States as a SAS consultant. He received MS in Electronics Engineering from St-Petersburg State Electrical Engineering University, formerly LEEI, in 1989. He only practices functional programming in his spare time, but it helps him a lot during his day job.

Neil Mitchell has an MEng in Computer Science and Software Engineering from the University of York. He is still there, working towards a PhD, under the supervision of Colin Runciman.

Matthew Naylor is a member of the programming languages and systems group at the University of York.

References

[1] The Yhc Team. The York Haskell Compiler - user's guide. `http://www.haskell.org/haskellwiki/Yhc`.

[2] Niklas Röjemo. Highlights from nhc - a space-efficient Haskell compiler. In **Proc. FPCA '95**, pages 282–292. ACM Press (1995).

[3] Hat – the Haskell Tracer. `http://www.haskell.org/hat`.

[4] Andrew Tolmach. An External Representation for the GHC Core Language (September 2001). `http://www.haskell.org/ghc/docs/papers/core.ps.gz`.

[5] XUL Overlays. `http://developer.mozilla.org/en/docs/XUL_Overlays`.

[6] Philip Wadler. How to replace failure by a list of successes. In **Proc. of a conference on Functional programming languages and computer architecture** (1985).

[7] AJAX applications in Haskell. `http://www.haskell.org//pipermail/haskell-cafe/2006-August/017286.html`.

[8] Re: AJAX applications in Haskell. `http://www.haskell.org//pipermail/haskell-cafe/2006-August/017287.html`.

[9] Hajax. http://www.haskell.org/haskellwiki/Hajax.

[10] Continuations. http://haskell.org/haskellwiki/Continuation.

[11] Fudgets Home Page. http://www.md.chalmers.se/Cs/Research/Functional/Fudgets/.

[12] World Wide Web Consortium. http://www.w3.org.

[13] Object Management Group. http://www.omg.org/gettingstarted/omg_idl.htm.

[14] W3C Document Object Model. http://www.w3.org/DOM/.

[15] HaskellDirect. http://www.haskell.org/hdirect/.

[16] Hugs 98. http://www.haskell.org/hugs.

[17] The Glasgow Haskell Compiler. http://www.haskell.org/ghc.

[18] Unicode Home Page. http://www.unicode.org.