The Monad.Reader Issue 6

by Bernie Pope ⟨bjpop@csse.unimelb.edu.au⟩
and Dan Piponi ⟨dpiponi@gmail.com⟩
and Russell O'Connor ⟨roconnor@alumni.uwaterloo.ca⟩

Wouter Swierstra, editor.

Contents

Editorial

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

It has been many months since the last issue of The Monad.Reader. Quite a few things have changed since Issue Five. For better or for worse, we have moved from wikipublishing to LaTeX. I, for one, am pleased with the result.

This issue consists of three top-notch articles on a variety of subjects: Bernie Pope explores just how expressive `foldr` is; Dan Piponi shows how to compile proofs in classical logic to Haskell programs; Russell O'Connor has written an embedded assembly language in Haskell.

Besides the authors, I would like to acknowledge several other people for their contributions to this issue. Andres Löh provided a tremendous amount of TeXnical support and wrote the class files. Peter Morris helped design the logo. Finally, I'd like to thank Shae Erisson for starting up The Monad.Reader – without his limitless enthusiasm for Haskell this magazine would never even have gotten off the ground.

Getting a Fix from the Right Fold

by Bernie Pope ⟨bjpop@csse.unimelb.edu.au⟩

What can you do with `foldr`*? This is a seemingly innocent question that will confront most functional programmers at some point in their life. I was recently posed a "folding challenge" by a teaching colleague. The challenge was to write* `dropWhile` *using* `foldr`*. We gave the challenge to our first-year students, and awarded a small prize to the author of the first working solution.*

I have since passed the challenge on to other "functional" friends, and the results have been illuminating. That prompted me to write this article.

Introduction

We can compute the sum of a list of numbers with the following recursive function:

```
sum :: Num a => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs
```

With only a couple of small changes, we can transform it into a function that computes the product of a list of numbers:

```
product :: Num a => [a] -> a
product []     = 1
product (x:xs) = x * product xs
```

There are two key differences between `sum` and `product`, apart from their names. The first difference is the value which is returned in the base case. The second difference is how the next value from the list is combined with the result from the recursive call.

Rather than repeat the same recursive pattern over-and-over again, we can distill its essence into a new, more general function. Thus we arrive at the definition of `foldr`:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr combine base []     = base
foldr combine base (x:xs) = combine x (foldr combine base xs)
```

Here we have the same pattern of recursion as found in `sum` and `product`, but the value of the base case and the combining function of the recursive case are taken as parameters, instead of being hard-coded. Effectively, `foldr` transforms a list into some other expression, by replacing all occurrences of the list constructor (`:`) with `combine`, and by replacing the empty list (`[]`) with `base` (if the end of the list is ever reached).

Now we can define `sum` and `product` by instantiating `foldr`'s first two parameters with appropriate values, like so:

```
sum     = foldr (+) 0
product = foldr (*) 1
```

Lots of useful functions over lists follow this pattern of recursion, which makes `foldr` widely applicable.

The challenge

As I said in the introduction, the challenge that was posed to me, and that I now pose to you, is to write `dropWhile` using `foldr`. The same challenge has been considered by other people in the literature, for example Richard Bird [1] sets it as exercise 4.5.2 in his introductory textbook on functional programming, and Graham Hutton [2] presents a solution (which we will see shortly) in his tutorial on the fold functions.

For reference, here is a straightforward recursive definition of `dropWhile`:

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile predicate [] = []
dropWhile predicate list@(x:xs)
   | predicate x = dropWhile predicate xs
   | otherwise   = list
```

It takes two arguments: a predicate and a list. It drops items from the front of the list until it finds one which falsifies the predicate, after which it returns the remainder of the list unchanged:

```
Hugs> dropWhile (<5) [1..10]
[5,6,7,8,9,10]
```

I encourage you to try your hand at a solution. But before you do, some ground rules are in order. Obviously it is not clear what it means to "use `foldr`". I will leave it to your better judgement as to what that entails, but trivial uses of `foldr` will not do. For instance, the following is **not** a solution:

```
dwCheating = const dropWhile foldr
```

It is also highly desirable that the non-strictness of `dropWhile` is preserved. For instance, it should still be able to return results on infinite lists:

```
Hugs> take 3 (dropWhile (<5) [1..])
[5,6,7]
```

A non-solution

One of the advantages of `foldr` is that we can get a long way into writing a recursive list-processing function without having to think at all.[1] We simply start out with the following skeleton, and then fill in the blanks:

```
foo = foldr combine base
    where
    base             = ...
    combine next rec = ...
```

Naturally, we would like to apply this "join-the-dots" style of programming to our challenge. A typical first approach looks something like this:

```
dwNoThinking predicate
   = foldr combine base
   where
   base = []
   combine next rec
      | predicate next = rec
      | otherwise      = next : rec
```

Alas, it doesn't work – though it is tantalisingly close. This function is equivalent to `filter (not . predicate)`, which is not the same as `dropWhile predicate`, for example:

[1] That frees up our brains for more important tasks, like remembering the commands in VI or Emacs.

```
Hugs> dropWhile even [2,3,4]
[3,4]
Hugs> dwNoThinking even [2,3,4]
[3]
```

The problem is in the second guarded equation of `combine`. Reasoning from the original definition of `dropWhile`, we want the body of this clause to return the remainder of the list unchanged. Unfortunately `foldr` does not make the value of that list available directly. All we have is the head of the list (`next`) and the result of the recursive call (`rec`), but `rec` is missing all the items of the tail of the list which satisfy the predicate. So it is not always possible to reconstruct the remainder of the list from `next` and `rec`.

Clearly `dropWhile` does not fit into the classic `foldr` pattern, which is why the challenge is so interesting. We are going to have to work a bit harder than usual.

Solution one – working backwards with tuples

The first solution we will consider appears in Hutton's fold tutorial.[2] The non-solution above shows that it is difficult to get `foldr` to compute the answer we want directly. One way around this is to get `foldr` to compute something which is close to the solution we want, and then, as a last step, post-process that into the desired value:

```
dwBackwards predicate = fst . dwPairs predicate

dwPairs :: (a -> Bool) -> [a] -> ([a], [a])
dwPairs predicate
   = foldr combine base
   where
   combine next (ys, xs)
      | predicate next = (ys,      next:xs)
      | otherwise      = (next:xs, next:xs)
   base = ([], [])
```

The idea is to work backwards through the list. That is, `dwPairs` winds its way down to the end of the list, and then builds up its answer – a pair of lists – from right to left.

To understand the need for two lists, it is useful to consider an arbitrary list element in its context:

[2] This was a popular approach amongst the people who tackled the challenge. My teaching colleague had this solution in mind, as did the student who won the challenge in our class, though neither had read Hutton's paper.

```
<prefix> item <suffix>
```

If `item` falsifies the predicate then it will always be included in the answer. However, if `item` satisfies the predicate, it will be kept in the answer only if there is at least one item appearing in `<prefix>` which falsifies the predicate, otherwise it will be dropped. When `item` is processed we don't know which scenario will eventuate because we haven't looked at the items in `prefix` yet, so we keep track of two lists: one which excludes the item, and one which includes it. If we later discover an item in the prefix which falsifies the predicate, we throw away the list with the dropped items, and copy over the list with the kept items. At the end of the computation the first list in the pair contains the answer that we are looking for, namely `dropWhile predicate` over our input list. As a final step, `dwBackwards` selects that list from the pair.

Though this solution is rather clever, it does have a significant failing point: it is too strict. By working backwards through the list it requires that the list is finite, thus it fails to produce any output at all when the list is infinite:

```
Hugs> take 3 (dwBackwards (<5) [1..])
ERROR - Control stack overflow
```

The next couple of solutions avoid this unwanted strictness.

Solution two – taking the higher (order) road

The philosophy of this next solution is a play on the old meta-programming slogan:

> Why write a function to solve a problem, when you can write a function which returns a function to solve that problem?

One of the pitfalls of this challenge is the desire to have `foldr` return the answer directly. Prior experience with `foldr` probably conditions us to think in this way. But if we look at the type of `foldr` we see that there is no obligation for the result to be a list; the result type is a variable, and that variable could well be instantiated to a function type.

In this solution we get `foldr` to build up a function, which, when applied to the input list, will return the desired result:[3]

[3]This solution was proposed independently by two people who received the challenge from me. It must be pointed out that they both had considerable experience with Haskell, and thus were likely to be quite comfortable with the nuances of higher-order programming.

```
dwHo predicate list
  = (foldr combine base list) list     -- brackets for emphasis only
  where
  base = id
  combine next rec
     | predicate next = rec . tail
     | otherwise      = id
```

It is best to view this function in action. Suppose we want to compute the result of `dwHo (<5) [1..10]`. The call to `foldr` evaluates to the following function composition:

```
id . tail . tail . tail . tail
```

that function is applied to the input list:

```
(id . tail . tail . tail . tail) [1,2,3,4,5,6,7,8,9,10]
```

which reduces to the desired result:

```
[5,6,7,8,9,10]
```

Clearly, this solution is an improvement over `dwBackwards`, because it is not strict in the tail of the list:

```
Hugs> take 3 (dwHo (<5) [1..])
[5,6,7]
```

But it is not without fault. Notice that the size of the function returned by `foldr` grows proportionally to the number of elements which are dropped from the front of the list. As this function grows in size, it consumes heap space, and so we get a space leak. Consider the evaluation of this expression: `dwHo (<100000) [1..]`. The problem is that we build up a big function composition first, and then apply it to the list. It would be better to interleave the construction of that function and its application to the list. That is the approach of the next solution.

Solution three – pulling in our tail to save space

This solution is a refinement of the previous one. It is based on the same higher-order idea, but it avoids the space leak, by interleaving the construction and application of the function returned by `foldr`.

We can arrive at this solution by applying some equational reasoning. Recall the `combine` function from above. It returns a function as its result. We can make this more explicit by giving it an extra argument by eta-expansion:

```
combine next rec list
   | predicate next = (rec . tail) list
   | otherwise      = id list
```

Note: this step does not change the behaviour of `combine`, it simply makes its third argument explicit. The next step is to in-line, and thus eliminate, the calls to `(.)` and `id`, like so:

```
combine next rec list
   | predicate next = rec (tail list)
   | otherwise      = list
```

Now comes the trick where we pull in our tail. Remember that `rec` stands for the result of the recursive application of `foldr`. Notice that it receives the expression '`tail list`' as its argument. Effectively this delays the application of `tail` until too late. What we would like to do is evaluate the call to `tail` before the recursive call, thus consuming some of the list as the evaluation of `foldr` proceeds, rather than waiting until the end. First, we in-line the call to `tail`:

```
combine next rec list
   | predicate next = rec (case list of (_:xs) -> xs)
   | otherwise      = list
```

Then we push the call to `rec` inside the case expression:

```
combine next rec list
   | predicate next = case list of (_:xs) -> rec xs
   | otherwise      = list
```

Voila! Space leak solved. A minor aesthetic improvement is to use pattern matching sugar instead of a case statement, giving us our final version:

```
dwTailFree predicate list
   = foldr combine base list list
   where
   base = id
   combine next rec list@(_:xs)
      | predicate next = rec xs
      | otherwise      = list
```

As it happens, the `dwTailFree` solution came to me in one of those rare **eureka!** moments, at 3am in the morning as I lay in bed, worrying over the strictness of `dwBackwards`. It wasn't until some time later that I realised the connection with `dwHo`. At first, I thought `dwTailFree` was the last thing that could possibly be said about the challenge, but there was one final discovery waiting for me some weeks later; that brings us to the fourth and final solution that I would like to present to you.

Solution four – the fix that you've been waiting for

As I said, I didn't think there was any more to be done with the challenge after `dwTailFree` came along. But it seems that the functional programming co-processor in my head was quietly busying itself on the problem whenever any spare cycles came its way.

At first it occurred to me that there were two ways of writing `combine` from `dwTailFree`. The original way:

```
combine next rec list@(_:xs)
   | predicate next = rec xs
   | otherwise      = list
```

and an alternate way:

```
combine _ rec list@(next:xs)
   | predicate next = rec xs
   | otherwise      = list
```

The difference is where we get the `next` value from the input list. The alternate version of `combine` got me thinking: in `dwTailFree`, the call to `foldr` doesn't need to look at the values in the first copy of the list. It simply uses the list structure to provide some traction for the recursion. I soon realised that we don't actually need two copies of the same list, the first list can be any list at all, providing it is sufficiently long to provide enough recursive calls.

Then it dawned on me – again as I lay in bed at night – `dwTailFree` is really using `foldr` as a fixpoint combinator!

We can take a recursive function, such as `dropWhile`, and pull out its recursive call as a parameter, thus eliminating the recursion:

```
dwNonRec :: ((a -> Bool) -> [a] -> [a]) -> (a -> Bool) -> [a] -> [a]
dwNonRec rec predicate [] = []
dwNonRec rec predicate list@(next:xs)
   | predicate next = rec predicate xs
   | otherwise      = list
```

Note that the parameter `rec` now takes the place of the recursive call. We can re-introduce the recursion using an explicit fixpoint operator:

```
dwFix :: (a -> Bool) -> [a] -> [a]
dwFix = fix dwNonRec
```

where `fix` is a function that satisfies the equation '`fix f = f (fix f)`'. You may notice that there is quite a lot of similarity between `dwNonRec` and the alternate version of `combine`.[4]

So the question is, can we write `fix` using `foldr`? Yes, we can:

```
fix :: (a -> a) -> a
fix f = foldr (\_ -> f) undefined (repeat undefined)
```

The `undefined` is just a gap-filler. The expression '`repeat undefined`' generates a list of unbounded length. We don't care about its elements, so anything will do. The fact that the list is infinite means that we get as many applications of the `f` parameter as we need. Note that, because the list is infinite, `foldr` will never reach the empty list, so it does not matter what value we give for the base case argument. Nonetheless, the type-checker requires that the argument has a polymorphic type, so it is quite convenient to use `undefined` here also.

To show that we have indeed implemented a suitable fixpoint function we can apply a little equational reasoning on the body of `fix`. Here is a possible definition of `repeat`:

```
repeat :: a -> [a]
repeat x = x : repeat x
```

We can unfold the application of `repeat` by one step:

```
foldr (\_ -> f) undefined (undefined : repeat undefined)
```

Then we can unfold the application of `foldr`:

```
(\_ -> f) undefined (foldr (\_ -> f) undefined (repeat undefined))
```

The application of the lambda abstraction can be simplified by beta reduction:

```
f (foldr (\_ -> f) undefined (repeat undefined))
```

From the original definition of `fix` we can see that this expression is equivalent to:

```
f (fix f)
```

which means that '`fix f = f (fix f)`', as desired.

[4] Indeed, the `rec` parameter is a dead giveaway.

Conclusion

Now that we have `fix` defined in terms of `foldr`, a whole world of opportunities arise, especially for devious exam questions!

In the end, it is not too surprising that we can get `fix` from `foldr`. Though it certainly didn't occur to me when I first was given the challenge by my colleague.

The next obvious question is what can you do with `foldl`? In particular, can you get a fix from that? I'll leave that as a homework exercise.

Postscript

Naturally, us Haskellers like to work lazily, and I must admit to being a little lazy in my research on this topic. Having stumbled upon this `foldr` – `fix` connection, the thought did occur to me that I should look it up in the literature. But I did not take it any further. To be honest, I was a little bit worried that the notion was so obvious that no one had bothered to even mention it before. Fortunately, parallel evaluation saved the day. Wouter Swierstra passed a draft of my paper to Graham Hutton, who pointed out that, to the best of his knowledge, Peter Freyd [3] was the first to present the idea. He showed how to derive a fixpoint operation from `foldr` and `unfold`. The `repeat` function I used in my definition of `fix` can, of course, be written in terms of `unfold`. Interested readers might enjoy the discussion in Section 2.6 of Hutton and Meijer's paper [4], which is a bit more accessible to functional programmers than Freyd's original work.

Acknowledgments

I would like to thank the following people who have contributed to the challenge and this paper. Tony Wirth, who posed the challenge to me, during a very enjoyable semester of teaching. Harald Søndergaard, Ben Horsfall, Marco Lui, Bryn Humberstone and Kevin Glynn, for attempting the challenge, and providing interesting solutions. Wouter Swierstra, for editing the Monad Reader. Graham Hutton, for reading an earlier draft and pointing out the reference to Freyd.

About the author

Bernie Pope is a postgraduate student within the Department of Computer Science at the University of Melbourne. Amongst other things, he is the designer of Buddha, a declarative debugger for Haskell. At the moment, he is spending three

months as an intern at Microsoft Research in Cambridge where he is working on a procedural debugger to be integrated with GHC.

References

[1] Richard Bird. **Introduction to Functional Programming Using Haskell**. Prentice Hall Europe, second edition (1998).

[2] Graham Hutton. A tutorial on the universality and expressiveness of fold. **Journal of Functional Programming**, 9(4):pages 355–372 (July 1999).

[3] Peter Freyd. Algebraically complete categories. In **Proceedings of the 1990 Como Category Theory Conference**, volume 1488 of **Lecture Notes In Math**, pages 95–104. Springer-Verlag (1990).

[4] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to Exponential Types. In **Proceedings of the 7th SIGPLAN-SIGARCH-WG2.8 International Conference on Functional Programming and Computer Architecture**, pages 324–333. ACM Press (June 1995).

Adventures in Classical-Land

by Dan Piponi ⟨dpiponi@gmail.com⟩

Many Haskell programmers are familiar with how the Curry-Howard isomorphism shows that logical propositions correspond to types and their proofs correspond to programs. But this correspondence only holds for intuitionistic logic where laws such as double negation elimination fail to hold. There are many papers that discuss how proofs using classical logic, with double negative elimination, can also be interpreted as programs, but some of the papers unfortunately assume considerable prerequisites and are not easy to understand. This is a presentation of these ideas in a (hopefully) less intimidating manner. And because this is written as literate Haskell, along the way I build a simple compiler for a programming language that supports double negative elimination allowing anyone to get their hands directly on the objects described.

The Curry-Howard Lens

Suppose we know both that p is true and that p implies q. Then by the principle of logic, as old as late antiquity, and known as *modus ponens*, we can deduce q. Similarly, suppose we have a function `f` of type `p -> q`, and an object `x` of type `p`, then by applying `f` to `x` we construct an object `f x` of type `q`. This, in essence, is what the Curry-Howard isomorphism is about. Logical proofs translate into programs with the propositions being translated into types. In the example we have just given, a proof step using *modus ponens* translates into a program step involving a function application. The isomorphism works so well that we can use exactly the same notation to talk about proofs and programs. In particular, we write $u :: p$ to mean u is a proof of the proposition p, `u::p` to mean that `u` is an object of type `p`, $p \to q$ to mean p implies q, and `p -> q` to mean the type of functions mapping objects of type `p` to objects of type `q`. If we have $x :: p$ and $f :: p \to q$ then write $fx :: q$ to mean the proof of q that deduces it directly from f and x by *modus ponens*.

The Curry-Howard isomorphism becomes a kind of lens through which we can look at statements of propositional calculus, and interpret them as statements about functional programming. With that in mind, it seemed appropriate to me to pick up a book on logic and start reading it through this lens. The book I chose was a textbook on logic from my undergraduate days many years ago: *Notes on Logic and Set Theory* by PT Johnstone [1]. Unfortunately for computer scientists, this book might not seem the best choice, as it is mainly aimed at mathematicians, but it's the only book on logic I have read, and more importantly, I wanted to approach this subject from a slightly different direction. In particular, picking up a book on logic for computer scientists and interpreting it as computer science doesn't seem like a great achievement. But taking a book on logic that is largely aimed at a different audience from computer scientists, and interpreting that as being about computer science – that seems to me much more like an interesting and non-trivial achievement. And when such a book on logic, viewed though the Curry-Howard lens, turns out to contain detailed instructions on how to write and optimise a compiler, then it gets really interesting, but now I'm getting ahead of myself.

Logic

Johnstone begins his section on propositional calculus by introducing three axioms:

$$\begin{array}{ll} \text{(a)} & (p \to (q \to p)) \\ \text{(b)} & ((p \to (q \to r)) \to ((p \to q) \to (p \to r))) \\ \text{(c)} & (\neg\neg p \to p) \end{array}$$

The symbol $\neg$ is simply shorthand: $\neg p$ really means $(p \to \bot)$ where $\bot$ is an unprovable proposition (though it requires some work to show that it is unprovable). In addition, Johnstone introduces just one derivation rule, *modus ponens*, which we've already discussed above. Note that the axioms above apply not just to the symbols p, q and r but to any propositions. For example $((p \to r) \to (q \to (p \to r)))$ is an instance of axiom (a) and hence we can introduce it in our proofs whenever we want.

Let's build Haskell datatypes to manipulate propositions:

```
> infixr 5 :->

> data Proposition = Proposition :-> Proposition
>                  | Symbol String
>                  | False deriving Eq
```

```
> not p = p :-> False

> instance Show Proposition where
>    show (a :-> b) = "(" ++ show a ++ " -> " ++ show b ++ ")"
>    show (Symbol s) = s
>    show False = "False"
```

The proposition $p \to q$ can now be built as the Haskell object

```
Symbol "p" :-> Symbol "q"
```

and we represent $\bot$ with `False`. Note how `show` has been defined so that these propositions are output in precisely the format recognised by Haskell. So a proposition may be output directly as a Haskell type simply by printing it.

We also need to represent proofs and we can do this as follows:

```
> data Proof = MP Proof Proof
>     | Axiom String Proposition deriving Eq

> instance Show Proof where
>     show (Axiom n t) = "(" ++ n ++ " :: " ++ show t ++ ")"
>     show (MP f x) = "(" ++ show f ++ " " ++ show x ++ ")"
```

A proof is either an application of *modus ponens* or an axiom. If it is an axiom then it contains a string which is its name and the proposition which it introduces. If it is an application of *modus ponens* then it's represented as `MP f x` where `f` is the condition and `x` is the antecedent. Note how `show` has been defined for this type so that when a proof is printed it is output exactly like a Haskell expression. In fact, if the names of all of the axioms correspond to names of Haskell functions then the output expression is perfectly evaluable Haskell.

We could also use some helpful utilities:

```
> source :: Proposition -> Proposition
> source (a :-> b) = a

> target :: Proposition -> Proposition
> target (a :-> b) = b

> consequence :: Proof -> Proposition
> consequence (Axiom _ p) = p
> consequence (MP f g) = target (consequence f)
```

```
> infixl 5 @@

> (@@) :: Proof -> Proof -> Proof
> f @@ g | source (consequence f) == consequence g = MP f g
>        | otherwise = error ("@@ error "++show f ++"   "++show g)
```

The function `consequence` takes a proof and returns the proposition that it proves. The binary operator `(@@)` is a safe version of `MP` that generates an error if *modus ponens* is applied when it's not appropriate. It's a lot like function application.

Now we need to consider our three axioms above. As it stands, the function `Axiom` allows you to introduce any proposition. We need to define an interface so that only axioms valid in our logic may be introduced. To this end, we define three functions `a`, `b` and `c`:

```
> a :: Proposition -> Proof
> a m@(p :-> (q :-> p'))
>  | p==p' = Axiom "a" m

> b :: Proposition -> Proof
> b m@((p :-> (q :-> r)) :-> ((p' :-> q') :-> (p'' :-> r')))
>  | p==p' && p==p'' && q==q' && r==r' = Axiom "b" m

> c :: Proposition -> Proof
> c m@(((p :-> False) :-> False) :-> p')
>  | p==p' = Axiom "c" m
```

Each of these functions constructs an `Axiom` value with a proposition of the correct form. We now have everything we need to start proving theorems. Consider Johnstone's first proof in this system, a proof that $p \to p$. It goes:

$$\begin{aligned}
(p \to ((p \to p) \to p)) &\quad \text{(instance of (a))} \\
((p \to ((p \to p) \to p)) \to ((p \to (p \to p)) \to (p \to p))) &\quad \text{(instance of (b))} \\
((p \to (p \to p)) \to (p \to p)) &\quad \text{(modus ponens)} \\
(p \to (p \to p)) &\quad \text{(instance of (a))} \\
(p \to p) &\quad \text{(modus ponens).}
\end{aligned}$$

We can write this proof using our Haskell combinators as follows:

```
> identity :: Proposition -> Proof
> identity p = b1 @@ a1 @@ a2 where
```

```
>    b1 = b ((p :-> (p :-> p) :-> p)
>                :-> (p :-> p :-> p)
>                :-> (p :-> p))
>    a1 = a (p :-> (p :-> p) :-> p)
>    a2 = a (p :-> p :-> p)
```

You should be able to see that `b1`, for example, corresponds to line 2 of the proof, and that the second `@@` corresponds to the final line of the proof.

I have used a particular style in writing my proof. The proof is essentially one line `b1 @@ a1 @@ a2` with the remaining lines saying which instances of the axioms are required. I will use this style throughout. In a more sophisticated implementation we'd have an automatic 'type checker' (or maybe I should say 'proposition checker') that would save us having to specify exactly which instance we need and our proofs really would be one-liners. Alternatively we can write this proof using the notation of λ-calculus as baa.

Curry-Howard revisited

At this point we have a complete working proof checking system for classical logic. If your attempt to construct a proof (i.e. an object of type `Proof`) using `a`, `b`, `c` and `(@@)` results in a Haskell runtime error then your proof wasn't valid. You can use `consequence` to find out what your proof has proved.

But so far we have made no use of the Curry-Howard isomorphism. So now it's time to start trying to interpret our proofs and propositions as types and programs. If we can write functions `a`, `b`, and `c` with type signatures corresponding to the relevant axioms, then we will immediately have a translation of any proof into a working program. Note that when I say `a`, `b` and `c` now I mean functions in the target, i.e. strings that are output by `show`. To find suitable functions there's a nice trick: we can use Lennart Augustsson's theorem prover Djinn [2]. In fact, here's part of a Djinn session:

```
Djinn> a ? p -> (q -> p)
a :: p -> q -> p
a x1 _ = x1
Djinn> b ? (p -> (q -> r)) -> ((p -> q) -> (p -> r))
b :: (p -> q -> r) -> (p -> q) -> p -> r
b x1 x2 x3 = x1 x3 (x2 x3)
```

These definitions appear in the 'preamble' later. It's looking good so far. We just need a function for axiom (c):

```
Djinn> c ? (not (not p)) -> p
-- c cannot be realized.
```

And now we hit a major snag. There is no function `c` of this type and we seem doomed. What went wrong?

The Curry-Howard isomorphism applies only to intuitionistic propositional calculus but double negative elimination is an axiom of classical logic. Intuitionistic logic is a form of logic that rejects some of the principles of classical logic, some of which might seem to be so obviously true that nobody could possibly reject them. Nonetheless, intuitionistic logic sometimes meshes better with with constructive nature of much of computer science [3]. My textbook is aimed at ordinary mathematicians who typically spend their days working with classical logic. But this doesn't mean our project has ended.

There is a 'translation' of statements of classical logic into statements of intuitionistic logic such that a statement provable in classical logic is turned into a statement provable in intuitionistic logic. This translation is known as the Gödel-Gentzen translation. We denote this translation by using $\perp$ as a superscript and it is defined by:

$$\begin{aligned} \perp^{\perp} &= \perp \\ p^{\perp} &= \neg\neg p = ((p \rightarrow \perp) \rightarrow \perp) \text{ where } p \text{ is any symbol} \\ (p \rightarrow q)^{\perp} &= (p^{\perp} \rightarrow q^{\perp}). \end{aligned}$$

There is also a corresponding transformation for the proofs although this is more complex to describe. However, notice how the translated propositions contain many occurrences of $\perp$. This is an unprovable proposition and so under the Curry-Howard isomorphism corresponds to a type without inhabitants. This means we have a perfectly good translation, but we end up translating propositions into ones that make frequent use of an uninhabited type. So it's actually a bit of a boring translation. Instead we modify the Gödel-Gentzen translation slightly to the following [4]:

$$\begin{aligned} \perp^{k} &= k \\ p^{k} &= \neg\neg p = ((p \rightarrow k) \rightarrow k) \text{ where } p \text{ is any symbol} \\ (p \rightarrow q)^{k} &= (p^{k} \rightarrow q^{k}). \end{aligned}$$

where k is any symbol we choose. This is still a perfectly well-behaved translation and we can implement it in Haskell as follows:

```
> class Translatable a where
>     (^) :: a -> Proposition -> a

> instance Translatable Proposition where
>    False^k = k
>    (p :-> q)^k = (p^k) :-> (q^k)
>    p^k = (p :-> k) :-> k
```

There is also a corresponding definition of `(^)` that translates proofs but as it is more complex, and the details aren't relevant to most of what I say, it's postponed until the appendix. The important thing that needs to be borne in mind is that `consequence (x^k) == (consequence x)^k`. In other words, the operator `(^)` translates propositions and proofs together so that translated proofs become proofs of translated propositions.

Suppose we write an expression of type a. Then after the Gödel-Gentzen translation we have something of type a^k. Suppose a corresponds to the type `Integer`, for example. Then we end up with an expression of type `(Integer -> k) -> k` after translation. It's not clear how we could display such a result. However, if we know that we are ultimately evaluating a classical expression of type `Integer`, then by choosing `k = Integer` we end up with something of type

```
(Integer -> Integer) -> Integer
```

We can easily convert one of these back into an integer by applying it to the identity function. So we now have a canonical choice of `k` in the Gödel-Gentzen translation. We simply choose it to be the type of the final expression we are evaluating.

In theory we now have the parts to start converting classical proofs into complete runnable programs. We apply `^k` to the proof and then use `show` to output the translated proof in Haskell. Here's a complete driver routine that performs this translation, outputs the result to a file with a 'preamble' (containing the definitions of the combinators) prepended to it, and then runs it. As I use the `system` command you may need to modify this code slightly to make it run with your operating system. The preamble is contained in the appendix.

```
> compile start = do
>     readFile "preamble.hs" >>= writeFile "out.hs"
>     appendFile "out.hs" "\nstart = "
>     appendFile "out.hs" (show $ start^(Symbol "K"))
>     appendFile "out.hs" "\n"
>     system "runhugs out.hs"
```

We can now run classical proofs using the function `compile`. The standard way to build integers in a small functional language like this is to use Church

numerals [5]. But such numbers are tricky to display and are hard to use. So we'll cheat a little. The functions `Symbol` and `Axiom` can serve as a small 'foreign function interface' that allows us to embed calls to Haskell functions within our language to extend it beyond the three combinators `a`, `b` and `c`. This will allow us to use familiar types such as `Integer`. But a type like `Integer` in our language will be translated into `(Integer -> k) -> k` so if we want to use integers, say, they'll have to be translated too. We need a way to lift Haskell objects of type `a` to `(a -> k) -> k`. We can cheat. We simply ask Djinn to conjure up a suitable function of type `a -> ((a -> k) -> k)` and place this in our preamble along with some similar functions for lifting other kinds of object:

```
> integer = Symbol "Integer"
> int n = Axiom ("lift0 " ++ show n) integer
> plus = Axiom "lift2 (+)" (integer :-> integer :-> integer)
```

Let's start with a simple example:

```
> ex1 = plus @@ int 1 @@ int 2
```

We can run this with `compile ex1`. Running the generated Haskell program returns `3`, as you would expect.

Unfortunately, it's not hard to see that writing proofs this way is rapidly going to become unwieldy when we have only three combinators `a`, `b` and `c` to work with (compare with Unlambda [6]). We need a way to simplify our proofs. One of the first theorems Johnstone proves about propositional calculus is precisely what we need, the Deduction Theorem. If S is a set of propositions and t is another proposition, then $S \vdash t$ is the statement that we can deduce t if we assume the propositions in S. If we are making assumptions, however, then we are no longer deriving theorems just from our original axioms (a), (b) and (c). Fortunately the Deduction Theorem gives a way to turn such 'proofs' back into proper proofs without any additional assumptions. The statement is:

Theorem 1 (Deduction Theorem)**.**

$$S \vdash (s \to t) \textit{ iff } S \cup s \vdash t$$

In other words, we can remove an assumption s from a proof of t and convert it into a proof of $s \to t$ that doesn't assume s. In our code we represent an assumption simply as a named axiom. We use the function `elim` to eliminate these assumptions and when they are all eliminated we have a theorem written in terms of just the axioms. The function `elim` is almost a literal implementation of Johnstone's proof except that I have added, as an optimisation, a clause to deal with proofs already free of the assumption being eliminated.

```
> free :: String -> Proof -> Bool
> free n (Axiom n' p) = n /= n'
> free n (MP a b)     = free n a && free n b

> elim :: Proof -> Proof -> Proof
> elim (Axiom n p) (Axiom n' q) =
>    if n==n'
>      then identity p
>      else a (q :-> (p :-> q)) @@ (Axiom n' q)
> elim (Axiom n p) m@(MP q r) =
>   if free n m
>     then
>       -- optimisation
>       let s = consequence m
>       in a (s :-> p :-> s) @@ m
>     else
>       let q' = elim (Axiom n p) q
>           r' = elim (Axiom n p) r
>           tj = consequence r
>           ti = target (consequence q)
>       in b ((p :-> (tj :-> ti)) :->
>            ((p :-> tj) :-> (p :-> ti))) @@ q' @@ r'
```

As our first proof in the new style consider the proof of $\bot \rightarrow q$. It's called `magic` because it apparently constructs anything you want from something of type `False`. The catch, of course, is that there are no objects of type `False`, so you can never use it to actually make something from nothing.

```
> magic :: Proposition -> Proof
> magic (False :-> q) = elim u1 $ c1 @@ (a1 @@ u1) where
>     a1 = a (False :-> ((q :-> False) :-> False))
>     c1 = c (not (not q) :-> q)
>     u1 = Axiom "u" False
```

Maybe you've noticed what's going on here. The `elim` function is exactly like lambda abstraction. In lambda abstraction we may use a variable inside an expression and then convert the expression into a function in the same way that we've just shown how to introduce a proposition and then convert it into a conditional. So we can write the above proof as $\lambda u \rightarrow c(au) :: \bot \rightarrow q$. In fact, we can define:

```
> lambda = elim
```

But what does it mean?

We have implemented a system with a 'function' `c`, but what does it mean? We know that it can't actually be a function. In some ways, meeting `c` is a little like meeting the complex number i for the first time. Syntactically it looks just like any other constant, and yet semantically it's not a number in any ordinary sense of the word. Consider the following example with suggestive variable names:

```
> catch = c
> ex2 = plus @@ int 1 @@ e where
>    throw1 = Axiom "throw" (not integer)
>    catch1 = catch (not (not integer) :-> integer)
>    e    = catch1 @@ (lambda throw1 $ throw1 @@ int 1)
```

The function `throw1` causes the computation of `e` to abort and the argument to `throw1` is passed out as the return value of `catch1`. Note how the type of `catch1` works. The function `throw1` takes an `integer` value and returns `False`. So the argument to `catch1` is of type `not (not integer)`. But ultimately, `catch1` returns an integer and so `catch1` turns a `not (not integer)` into an `integer`. This example isn't interesting but it becomes more interesting when we nest `catch`es.

This is nothing like an evaluation in a pure functional language. The `catch` function has a bizarre non-local effect which becomes more apparent in the following examples:

Compare

```
> ex3 = plus @@ int 1 @@ e1 where
>   throw1 = Axiom "f" (not integer)
>   throw2 = Axiom "g" (not integer)
>   c1 = c (not (not integer) :-> integer)
>   c2 = c (not (not integer) :-> integer)
>   e1 = c1 @@ (lambda throw1 (throw1 @@ (plus @@ int 10 @@ e2)))
>   e2 = c2 @@ (lambda throw2 (throw1 @@ int 100))
```

and

```
> ex4 = plus @@ int 1 @@ e1 where
>   throw1 = Axiom "f" (not integer)
>   throw2 = Axiom "g" (not integer)
>   c1 = c (not (not integer) :-> integer)
>   c2 = c (not (not integer) :-> integer)
>   e1 = c1 @@ (lambda throw1 (throw1 @@ (plus @@ int 10 @@ e2)))
>   e2 = c2 @@ (lambda throw2 (throw2 @@ int 100))
```

In the former case we really are throwing the value 100 past the addition of 10 whereas in the latter the throw is to the inner `catch` and hence the addition of 10 does take place.

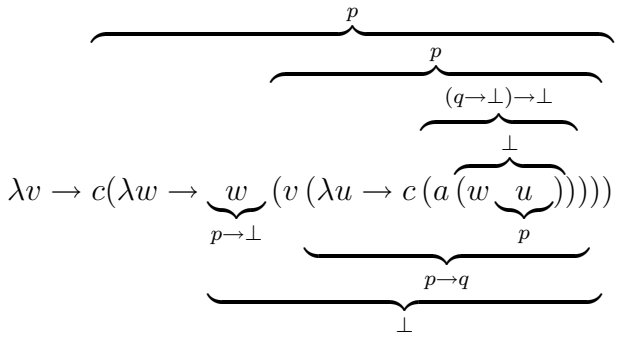
Using `catch` is a little awkward. It would be better if we could throw values out from any subexpression. We would also like `catch` to simply return the value of the expression inside it, when this expression does not use a `throw`. Instead of having the type

```
(((p -> False) -> False) -> p)
```

`catch` should have a type of the form `(((? -> ?) -> ?) -> ?` for some substitution of types for the `?`s. We'd like that the value of the `catch` is the same as the value thrown so that narrows things down to `(((p -> ?) -> ?) -> p`. But we'd also like to be able to throw from a context of any type, not just one of type `False`. So we must have type `(((p -> q) -> ?) -> p`. And if we'd like to have the option to not throw then the value that `throw` maps to must match `p` meaning the new improved `catch` should have type `(((p -> q) -> p) -> p`. Amazingly this type corresponds to a well known theorem of classical logic known as Peirce's law. ('Peirce', by the way, is pronounced like 'purse'.) We can prove it via:

$((p \to q) \to p)$	(hypothesis v)
$(p \to \bot)$	(hypothesis w)
p	(hypothesis u)
$\bot$	(modus ponens)
$(\bot \to (q \to \bot) \to \bot)$	(instance of (a))
$((q \to \bot) \to \bot)$	(modus ponens)
q	(instance of (c))
$(p \to q)$	(deduction theorem from (u))
p	(modus ponens)
$\bot$	(modus ponens)
$((p \to \bot) \to \bot)$	(deduction theorem from (w))
p	(instance of (c))
$(((p \to q) \to p) \to p)$	(deduction theorem from (v))

We can rewrite this as a partly annotated λ-expression:



And that in turn can be translated into the following code:

```
> peirce (((p :-> q) :-> p') :-> p'')
>   | (p',p'')==(p,p) =
>     lambda v1 $ c1 @@ (lambda w1 $ w1 @@ (v1 @@
>       (lambda u1 $ c2 @@ (a1 @@ (w1 @@ u1)))))
>       where
>          v1 = Axiom "v" ((p :-> q) :-> p)
>          w1 = Axiom "w" (not p)
>          u1 = Axiom "u" p
>          a1 = a (False :-> not q :-> False)
>          c1 = c (not (not p) :-> p)
>          c2 = c (not (not q) :-> q)
```

We can now throw subexpressions out from anywhere within a `peirce` and if we fail to throw anything the value of the function within `peirce` is simply returned.

```
> ex5 = plus @@ int 1 @@ (peirce1 @@
>   (lambda throw1 $ plus @@ int 2 @@ (throw1 @@ int 77)))
>     where
>     peirce1 = peirce i
>     i = ((integer :-> integer) :-> integer) :-> integer
>     throw1 = Axiom "throw" (integer :-> integer)
```

Try `compile ex5` to test it out!
The `peirce` function is better known by the name `callcc`

```
> callcc = peirce
```

from Scheme's 'call with current continuation'.

And that was the goal I was trying to achieve all along. I knew that call with current continuation was supposed to have a type corresponding to Peirce's law but I didn't understand how this could be, what it meant, and it all seemed rather *ad hoc*. But by starting from some simple axioms, this operation and its implementation has arisen in a completely natural way. I didn't even have to know what a 'continuation' was to implement it.

How Does it Work?

Consider a Haskell expression such as `f (g 3)` where `f` and `g` are defined as

```
f x = 2*x
g x = x+1
result = f (g 3)
```

We can approximately model how this is evaluated as follows: the function `f` needs its argument to be evaluated so `g` is first called with argument `3`. Then `g` returns some value which is in turn handed into `f`.

Now imagine a slightly different way of coding where instead of just accepting what a function evaluates and then doing the next thing, you explicitly tell each function you call what to do next. Let me rewrite the above example in this style:

```
f x c = c (2*x)
g x c = c (x+1)
result c = g 3 (\x -> f x c)
```

Even the final result has been rewritten so that it is, in some sense, 'awaiting orders' for what to do with its result. To actually get a numerical answer we need to tell it to do nothing more by evaluating `result id`. But here's the curious thing: we can rewrite `g` to be:

```
g x c = x+1
```

In other words, `g` is at liberty to ignore `c` and simply throw out its own result immediately. So by writing in this 'continuation passing style' (CPS) we are able to write functions that can control the future execution of programs just like `catch` and `throw` above. In fact, the Gödel-Gentzen translation turns out to be one way (among many) to translate code into continuation passing style. As I didn't need to know this to write the above code I'm not going to say too much about it here, but ultimately to understand what's really going on requires an understanding of continuations [7]. For now, I'll just mention that the argument `c` in the above examples is known as a *continuation* – it specifies how the program should 'continue'. In Haskell the `Cont` monad is defined by:

```
newtype Cont r a = Cont { runCont :: (a -> r) -> r }
```

which looks suspiciously like a^r.

Conclusions

It's time now to summarise what has been achieved. We have shown how to translate theorems of classical propositional calculus into intuitionistic propositional calculus and thence into Haskell. So we have a compiler for a programming language whose syntax is identical to classical propositional calculus. We have done so without making any reference to compiler writing texts, even as far as implementing lambda abstraction and continuations. We have seen how this programming language is essentially typed lambda calculus with support for continuations, but we didn't have to develop a new kind of abstract machine, or even know what a continuation was, to do this. We have also shown how properties of classical logic can be used to write code for our language. All of this comes more or less 'for free' from using the (modified) Gödel-Gentzen translation combined with the Curry-Howard isomorphism.

None of these are novel results. Continuations have a long history [8] with many papers describing the connection with classical logic [9]. But I hope I've given a less daunting path into some of the publications. Along the way we've also shown Haskell can play a useful role as a test bed for experimenting with logic. We also now have a typed combinator language to play with, a little like Unlambda [6]. And most curiously of all - we see that within an ordinary textbook on logic is a secret hidden text on compiler writing and we only need to look through the Curry-Howard lens to reveal it!

Acknowledgements

Thanks to Dirk Thierbach for a comment on comp.lang.functional that was cryptic enough not to give the whole game away but helpful enough to encourage me to figure out what was going on for myself. And thanks to the guys on `#haskell` for making suggestions and fixing typos.

About the author

Dan Piponi did his PhD in Mathematics at King's College, London. Since then, he has worked in the graphics and visual effects industry for 12 years – with movie credits including all three Matrix movies. He is probably the only Haskell hacker with an Academy Award to his name.

Exercises

1. We can define logical disjunction via $a \vee b = \neg a \to b$. Use this to define versions of Haskell's type constructor `Either`, the constructor `Left`, the constructor `Right` and `either`. For example `Left` should correspond to a map $a \to (\neg a \to b)$.
2. We can define conjunction, $a \wedge b$ similarly. Give a suitable definition and use your definition to define the type constructor `(,)`, the data constructor `(,)`, and the functions `fst` and `snd`.
3. Given any 'function' in our programming language, $a \to b$, we can define a dual $\neg b \to \neg a$. Use this and De Morgan's Law to find a way to automatically get a solution to problem (3) in terms of problem (2).
4. How does `either` work in terms of the semantics of `catch`? Where does the story of the Devil recounted by Wadler [10] fit in?
5. Describe what happens in the following code fragment:

```
callcc (\throw -> plus @@ throw (int 1) @@ throw (int 2))
```

6. Write a theorem prover for classical logic and test whether it correctly produces an implementation of `peirce`. (Theorem provers for classical propositional calculus are much easier to write than theorem provers for intuitionistic propositional calculus.)
7. Prove that `c` does what I claim.
8. The compiler we have written is really a meta program that generates another Haskell program. Rewrite it using Template Haskell. (Suggested by Wouter Swierstra.)
9. The function `magic` doesn't actually create something from nothing, and yet it is a useful component in some proofs. What purpose does it serve?

References

[1] P. T. Johnstone. **Notes on logic and set theory**. Cambridge University Press, New York, NY, USA (1987).

[2] `http://www.augustsson.net/Darcs/Djinn/`.

[3] `http://plato.stanford.edu/entries/logic-intuitionistic/`.

[4] Hajime Ishihara. A Note On The Gödel-Gentzen Translation. `http://citeseer.ist.psu.edu/ishihara98note.html`.

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

[6] `http://www.madore.org/~david/programs/unlambda/`.

[7] http://en.wikipedia.org/wiki/Continuation.

[8] John C. Reynolds. The Discoveries of Continuations. **LISP and Symbolic Computation**, 6(3–4):pages 233–247 (1993). http://citeseer.ist.psu.edu/reynolds93discoveries.html.

[9] Timothy G. Griffin. The Formulae-as-Types Notion of Control. In **Conf. Record 17th Annual ACM Symp. on Principles of Programming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan 1990**, pages 47–57. ACM Press, New York (1990). http://citeseer.ist.psu.edu/griffin90formulaeastypes.html.

[10] Philip Wadler. Call-by-value is dual to call-by-name. http://homepages.inf.ed.ac.uk/wadler/papers/dual/dual.pdf.

[11] A. S. Troelstra and D. van Dalen. **Constructivism in Mathematics: an Introduction**. Studies in Logic and the Foundations of Mathematics, North-Holland (1988).

Appendix: Gödel-Gentzen translation

To make the translation easier I, introduce new combinators `d`, `e`, `f` and `g`. These could all be written in terms of `b` and `c`, but it's easier to use Djinn to find suitable definitions.

This algorithm is more or less a literal interpretation of theorem 2.3.5 in Troelstra and van Dalen [11]. Translation of proof steps using double negation elimination are built by induction on the depth of the rightmost leaf of the proposition.

```
> pushdown q proof = g (
>             consequence proof
>             :-> (q :-> source (consequence proof))
>             :-> (q :-> target (consequence proof))
>             ) @@ proof

> g' p q r = g $ ((q :-> r) :-> (p :-> q) :-> (p :-> r))
> g2 a b | source (consequence a) == target (consequence b) =
>     ((g' (source (consequence b))
>         (target (consequence b)) (target (consequence a))) @@ a) @@ b

> not' k p = p :-> k

> not'2 k = not' k . not' k
> not'4 k = not'2 k . not'2 k
```

```
> instance Translatable Proof where
>    m@(Axiom "a" _)^k = a ((consequence m)^k)
>    m@(Axiom "b" _)^k = b ((consequence m)^k)
>    m@(MP f g)^k = (f^k) @@ (g^k)
>    m@(Axiom "c" (((p :-> False) :-> False) :-> p'))^k =
>            foldl1 g2 (reverse $ steps p) where
>            steps False = [f (((k :-> k) :-> k) :-> k)]
>            steps p@(Symbol _) = [d (not'4 k p :-> not'2 k p)]
>            steps (p :-> q) =
>                let pushed = map (pushdown (p^k)) (steps q)
>                in e (not'2 k ((p^k) :-> (q^k)) :->
>                  source (consequence (head pushed))) : pushed
>    m@(Axiom a p)^k = Axiom a (p^k)
>    x^k = error ("Can't gg_proof k " ++ show x)

> d m@((((p :-> k) :-> k1) :-> k2) :-> (p' :-> k3))
>    | (p,k,k,k)==(p',k1,k2,k3) = Axiom "d" m

> e m@((((p :-> q) :-> end1) :-> end2) :->
>     (p' :-> ((q' :-> end3) :-> end4))) |
>     (p,q,end1,end1,(end1,end1))==(p',q',end1,end2,(end3,end4)) =
>     Axiom "e" m

> f m@(((result :-> end1) :-> end2) :-> end3) |
>    (result,result,result)==(end1,end2,end3) = Axiom "f" m

> g m@((q :-> r) :-> (p :-> q') :-> (p' :-> r'))
>    = if (p,q,r)==(p',q',r') then Axiom "(.)" m else error "g"
```

Appendix: Preamble

The following code forms the preamble and should be placed in preamble.hs:

```
a :: p -> q -> p
a x1 _ = x1

b :: (p -> q -> r) -> (p -> q) -> p -> r
b x1 x2 x3 = x1 x3 (x2 x3)

d :: (((a -> r) -> r) -> r) -> a -> r
d x1 x2 = x1 (\ c8 -> c8 x2)

e :: (((p -> q) -> c) -> c) -> p -> (q -> c) -> c
e x1 x2 x3 = x1 (\ c9 -> x3 (c9 x2))

f :: ((c -> c) -> c) -> c
f x1 = x1 (\ c6 -> c6)

lift0 :: a -> ((a -> r) -> r)
lift0 x1 x2 = x2 x1

lift1 :: (a -> b) -> ((a -> c) -> c) -> (b -> c) -> c
lift1 x1 x2 x3 = x2 (\ c8 -> x3 (x1 c8))

lift2 :: (a -> b -> c) -> ((a -> r) -> r) ->
    ((b -> r) -> r) -> (c -> r) -> r
lift2 x1 x2 x3 x4 = x3 (\ c17 -> x2 (\ c19 -> x4 (x1 c19 c17)))

type K = Integer

main = print (start id)
```

Assembly: Circular Programming with Recursive do

by Russell O'Connor ⟨roconnor@alumni.uwaterloo.ca⟩

Monads and higher order functions make Haskell particularly well suited for creating domain specific languages. This article will discuss a simple implementation of an assembly language within Haskell. This example will make use of circular programming, and particularly use of recursive do. Circular programming is an under-utilized programming idiom in Haskell. I hope this article inspires readers to use circular programming in their code.

The Assembly Language

In the end we want to build an assembly language library that will allow users to write code that looks as follows:

```
testMC :: AssemblyCode
testMC = mdo {
  start <- label;
      iout '>' T Reg7;
      inp  Reg0;
      high Reg1;
      ljeq Reg0 Reg1 end T Reg4 Reg5 Reg6 Reg7;
      out  Reg0;
      ljmp start T Reg6 Reg7;
  end <- label;
      halt;
  }
```

Assembly language is a sequence of machine code instructions that operates on registers and memory, etc. for a particular machine. Various points in the

sequence of instructions are labeled. Jump instructions move the control flow to a particular label when a given condition is satisfied. Labeling of instruction points is necessary because several different `AssemblyCode` objects are often linked together later. Therefore, the actual locations of the instructions is not known until link time.

The `assemble` function will take an `AssemblyCode` object and encode it as machine code (in this case `Word32`s). The machine code can be written to standard out with the following code.

```
main = mapM out (assemble testMC)
 where
  out x = putStr [x0,x1,x2,x3]
   where
    x0 = toEnum . fromIntegral $ (x `shiftR` 24)
    x1 = toEnum . fromIntegral $ (x `shiftR` 16) .&. 0xff
    x2 = toEnum . fromIntegral $ (x `shiftR` 8) .&. 0xff
    x3 = toEnum . fromIntegral $ x .&. 0xff
```

Haskell has become an assembler!

This program does not evaluate the assembly instructions. The only purpose is to compute the machine code for sequence of assembly instructions. In particular it computes the location of the labels and correctly builds the jump instructions. Let's see how to implement `AssemblyCode`.

The Universal Machine

The 2006 ICFP Programming Contest committee recovered 2000 year old tablets that contain a specification for ancient universal machine. A transcription of the tablet can be found at `http://icfpcontest.org/um-spec.txt`. Don't worry, there are no spoilers here. The goal here is to create an assembler for the specified machine.

According to the specification the machine has eight registers.

```
data Register =
  Reg0 | Reg1 | Reg2 | Reg3 | Reg4 | Reg5 | Reg6 | Reg7
    deriving (Eq, Show, Enum, Bounded)
```

There are fourteen instructions and each operation specifies which registers it operates on.

```
data Instruction r =
    CMove r r r
  | Read r r r
  | Write r r r
  | Add r r r
  | Mul r r r
  | Div r r r
  | Nand r r r
  | Halt
  | Alloc r r
  | Free r
  | Output r
  | Input r
  | Load r r
  | Orth r Word32
    deriving (Eq, Show)
```

Here I have abstracted over the type of registers. This extra flexibility may come in handy later.

Each instruction is encoded as a 32-bit word according to the specification.

```
encode :: Instruction Register -> Word32
encode (CMove a b c) = encodeOp 0 a b c
encode (Read a b c)  = encodeOp 1 a b c
encode (Write a b c) = encodeOp 2 a b c
encode (Add a b c)   = encodeOp 3 a b c
encode (Mul a b c)   = encodeOp 4 a b c
encode (Div a b c)   = encodeOp 5 a b c
encode (Nand a b c)  = encodeOp 6 a b c
encode Halt          = encodeOp 7 minBound minBound minBound
encode (Alloc b c)   = encodeOp 8 minBound b c
encode (Free c)      = encodeOp 9 minBound minBound c
encode (Output c)    = encodeOp 10 minBound minBound c
encode (Input c)     = encodeOp 11 minBound minBound c
encode (Load b c)    = encodeOp 12 minBound b c
encode (Orth a v)    =
  (13 `shiftL` 28) .|. (convert a `shiftL` 25) .|. v

convert = fromIntegral . fromEnum
```

```
encodeOp op a b c =
  (op `shiftL` 28) .|.
  (convert a `shiftL` 6) .|.
  (convert b `shiftL` 3) .|.
  convert c

encodeAsm = map encode
```

Now that the instructions are defined, it is time to build an assembler.

The Haskell 98 Solution

My very first attempt at writing an assembler was awful. The tricky part of an assembler is getting the jump instructions to jump to the right place. My first program output the encoded instructions, plus a table of labels and instruction locations. Then I had to rerun my program to pass in the correct association list between labels and instructions. Like TEX, it needed to be run more than once. Certainly there is a better way than running a program twice.

The following is my second attempt. I create a particular RWS (Reader-Writer-State) monad. This monad will be used to sequence assembly instructions and labels.

```
newtype AssemblyCodeMonad a =
  AssemblyCodeMonad
     (RWS [(Label,Location)]
          [Either (Instruction Register) (Label,Location)]
          (Location, Integer)
          a)
  deriving (Monad, MonadReader a, MonadWriter a, MonadState a)

type AssemblyCode = AssemblyCodeMonad ()
```

The monad reads an association list of type `[(Label,Location)]`. This association list maps labels to locations. For the moment, assume that this map will be given as an oracle. The monad writes either an instruction, or a label with a location. Finally, the state consists of the current instruction location, and a second counter that will be used to generate fresh label names.

There are two kinds of labels, a user defined, (hopefully) globally unique string; and an automatically generated unique label.

```
data Label = Label String | Fresh Integer
    deriving (Eq, Show)
```

According to the universal machine specification each instruction is located in a platter indexed by a 32-bit number in the '0' array. I make a new type, `Location`, for reasons that will become clear. Making a new type is also good programming practice.

```
newtype Location = Location Word32
```

The most basic operation for this monad is outputting an instruction. This command writes the instruction and increases the location counter.

```
writeInstruction :: Instruction Register -> AssemblyCode
writeInstruction i = do
  tell [Left i]
  modify (\(Location a,b) -> (Location (succ a), b))
```

For each instruction I make a corresponding `AssemblyCode` command that writes the instruction.

```
-- I rename condition move (cmove) to the more descriptive
-- move-not-equal-zero (mneqz)
mneqz cond src dst = writeInstruction (CMove dst src cond)
read arr ix dst    = writeInstruction (Read dst arr ix)
write src arr ix   = writeInstruction (Write arr ix src)
add src1 src2 dst  = writeInstruction (Add dst src1 src2)
mul src1 src2 dst  = writeInstruction (Mul dst src1 src2)
div src1 src2 dst  = writeInstruction (Div dst src1 src2)
nand src1 src2 dst = writeInstruction (Nand dst src1 src2)
halt               = writeInstruction Halt
alloc size dst     = writeInstruction (Alloc dst size)
free ptr           = writeInstruction (Free ptr)
out src            = writeInstruction (Output src)
inp dst            = writeInstruction (Input dst)
load ptr offset    = writeInstruction (Load ptr offset)

-- Orth can only write 25 bit numbers
set25 val dst      = writeInstruction (Orth dst val)
```

These basic instructions are a little primitive to do serious programming with. One can sequence these primitive instructions and create new instructions. A normal assembler has some small macro language for this, but I have the entire Haskell language avaiable for making "macros".

Some of these new instructions will need to use temporary registers for their work. I choose to separate the temporary registers from source and destination registers by introducing a separate data type `T` with a single constructor.

```
data T = T

not src dst = nand src src dst

neg src dst T tmp = do
      set25 0 tmp
      not tmp tmp
      mul src tmp dst

sub src1 src2 dst T tmp = do
      neg src2 src2 T tmp
      add src1 src2 dst
-- return src2 to its original value only if
-- src2 isn't the destination
      unless (dst==src2) (neg src2 src2 T tmp)

shftG op n src dst T tmp
 | n < 25 = do
      set25 (2^n) tmp
      op src tmp dst
 | otherwise = do
      shftG op 24 src dst T tmp
      shftG op (n-24) dst dst T tmp

shftr = shftG div
shftl = shftG mul

set32 val dst T tmp = do
      set25 (val `shiftR` 16) dst
      shftl 16 dst dst T tmp
      set25 (val .&. 0xFFFF) tmp
      add tmp dst dst

jmp offset T tmp = do
      set25 0 tmp
      load tmp offset
```

In assembly programming it is typical to jump to a labeled instruction rather than jumping to a location stored in a register. The `lset` function loads a location into a register. The `ljmp` calls `lset` to loads a location and then `jmp` to jumps to it. Finally, the `loc` function translates a label into a location. I postpone the

definition of `loc` for the moment.

```
lset l dst T tmp = do {
  Location i <- loc l;
      set32 i dst T tmp;
  }

ljmp l T tmp1 tmp2 = do {
      lset l tmp1 T tmp2;
      jmp tmp1 T tmp2;
  }
```

In order to jump to labels, one needs to make the labels. This is the second thing the monad can write. A label statement gets the current instruction counter and associates the label with it.

```
label :: Label -> AssemblyCode
label l = do
  (a,b) <- get
  tell [Right (l,a)]
```

There are two types of labels. The most common type is unique string provided by the user

```
labelName :: String -> AssemblyCode
labelName s = label (Label s)
```

However, when making "macros" a user supplied globally unique string is impossible because each instance of the macro would reuse the same label name. So instead we use the monad to generate a fresh label name.

```
fresh :: AssemblyCodeMonad Label
fresh = do
  (a,b) <- get
  put (a, succ b)
  return (Fresh b)
```

Labels are transformed into to locations by asking the oracle. The oracle returns an association list of all labels and locations. The function `loc` asks the oracle for this list and then looks up the label in the association list.

```
loc :: Label -> AssemblyCodeMonad Location
loc l = do
  labels <- ask
  let ~(Just i) = lookup l labels
  return i
```

What does this ~ mean? The ~ makes the match irrefutable. This means that the actual pattern match is delayed until `i` is demanded. Actually all let expressions are irrefutable, so the ~ is not necessary; however, I leave it here for emphasis. I will show later that the lazy match here is very important.

The jump-if-equal-zero "macro" provides an example of using labels. The `jeqz` function generates a unique label whose value is assigned to the `skip` variable. The location at the end of the macro is given this label. The label is used when it is passed to the `lset` function.

```
jeqz cond offset T tmp1 tmp2 = do {
  skip <- fresh;
      lset skip tmp1 T tmp2;
      mneqz cond offset tmp1;
      jmp offset T tmp1;
  label skip;
  }
```

Notice that the location of the label is queried by `lset` before the location for the label is set. Thanks to the oracle this is possible. But how does one build this oracle? This is handled by the `link` function which runs the RWS monad.

```
link :: AssemblyCode -> [Instruction Register]
link (AssemblyCodeMonad mc) = asm
 where
  ((), _, output) = runRWS mc labels (Location 0,0)
  (asm, labels) = splitEithers output

splitEithers :: [Either a b] -> ([a], [b])
splitEithers = foldr accum ([], [])
 where
  accum = either (first . (:)) (second . (:))
```

The `AssemblyCodeMonad` is run starting with inital state `(Location 0,0)`. The return value, and final state are discarded. The output is split into two pieces, the assembly instructions and the association lists of labels and locations. The assembly instructions is the returned as result of `link`. The assocation list of labels

and locations is feed back as input to become the oracle for the RWS monad. This circularity of passing the output of `runRWS` back as the input of `runRWS` is called tying-the-knot, and it is the essential part of circular programming.

Here is an example of using the `AssemblyCode` monad to generate machine code for the UM. The assembly program repeatedly writes a prompt and echos one character until the end of input.

```
assemble = encodeAsm . link

testMC :: AssemblyCode
testMC = do {
  labelName "start";
      iout '>' T Reg7;
      inp  Reg0;
      high Reg1;
      ljeq Reg0 Reg1 (Label "end") T Reg4 Reg5 Reg6 Reg7;
      out  Reg0;
      ljmp (Label "start") T Reg6 Reg7;

  labelName "end";
      halt;
}

main = mapM out (assemble testMC)
 where
  out x = putStr [x0,x1,x2,x3]
   where
    x0 = toEnum . fromIntegral $ (x `shiftR` 24)
    x1 = toEnum . fromIntegral $ (x `shiftR` 16) .&. 0xff
    x2 = toEnum . fromIntegral $ (x `shiftR` 8) .&. 0xff
    x3 = toEnum . fromIntegral $ x .&. 0xff
```

You can build this test program by compiling this with GHC using the command `ghc --make -main-is Haskell98Solution Haskell98Solution.lhs`

How Does It Work?

That was the example, but how does it work, or rather why does it work? As we all know Haskell is a lazy language which means it has an unusal order of evaluation equivalent to the so called normal order evaluation. Normal order evaluation has the property that if any evaluation order reduces to a result, then normal order

will also. This means one can forget what the actual order of evaluation is. So long as one can find some order of evaluation that delivers a result, then one knows the Haskell code give the same result.

In this case, imagine `runRWS` first going through and computing all the instruction names, without filling in the the instruction parameter values. At the same time it creates the association list of labels and locations. Imagine that only after this association list is created does the evaluator go through and fill in the parameters for the instructions. Because only the parameters for the instructions require the association list this evalutation order works. This means that whatever order Haskell uses, it will compute the same result.

Time Travel

That is the official explaination, but I prefer the sci-fi explaination. When we pass the output of `runRWS` into the input for the oracle we are actually sending the data backwards in time. So when `loc` queries the oracle we get a result from the future.

Time travel is a very dangrous business. One false move and you can create a temporal paradox that will destory the universe (which in this case means that the computation will diverge). When programming with values from the future, it is important never, **never**, to do anything with the values that **might** change the future. This is the temporal prime directive.

Here is an example of flagrant disregard for the temporal prime directive. I thought it would be a good idea to have the RWS monad fail in case the lookup of a label failed. So I originally wrote the following:

```
loc :: Label -> AssemblyCodeMonad Location
loc l = do
  labels <- ask
  lookup l labels
```

However, this violates the temporal prime directive. If the lookup fails then the entire monad fails. This would mean that the association list is never created. The association list would never be sent back in time. Without an assocation list we would never be able to inspect it in order to determine that our label isn't in it. Voila, temporal paradox. If you use the code above, then the whole compuation diverges.

Temporal paradoxes can be more subtle than this. What do I mean by doing something that **might** change the future? When you do a case analysis on the data from the future, all the data that you touch from then on becomes tainted and must not come in contact with the data that will be sent back in time. This is why I emphasise the irrefultable pattern match in `loc`. If a pattern match

actually occured at that point in the monad, the rest of the monad commands would become tainted. Fortunately the `let` expression only taints the variable `i`.

Another example of a temporal paradox would be using `set` instead of `set32` for in the definition of `lset`

```
set val dst T tmp | val < 2^25 = set25 val dst
                  | otherwise = set32 val dst T tmp

lset l dst T tmp = do
  Location i <- loc l
        set i dst T tmp
```

The `set` function does a case analysis on `val`, to see which of `set25` or `set32` to use. The problem is that doing such an analysis on a location from the future might change the number of instruction made, which in turn could change the location of the label. Even if all the locations will be less than 2^{25}, the case analysis here is enough to cause a paradox.

Because data from the future is so volitle, I wrap the location up with a `newtype`. The user of the module has no way to directly access the contents of a `Location` because the Location constructor is not exported. This helps prevent the user from causing temporal paradoxes.

Before moving on, here are a couple more examples of "macros" that one can define.

```
-- immediate out
iout c T tmp = do {
      set25 (fromIntegral $ fromEnum c) tmp;
      out tmp;
  }

zero dst = do {
      set25 0 dst;
  }

-- set register to all ones
high dst = do {
      zero dst;
      not dst dst;
  }
```

```
jeq cond1 cond2 offset T tmp1 tmp2 tmp3 = do {
      sub cond1 cond2 tmp1 T tmp2;
      jeqz tmp1 offset T tmp2 tmp3;
  }

ljeqz cond l T tmp1 tmp2 tmp3 = do {
      lset l tmp1 T tmp2;
      jeqz cond tmp1 T tmp2 tmp3;
  }

ljeq cond1 cond2 l T tmp1 tmp2 tmp3 tmp4 = do {
      lset l tmp1 T tmp2;
      jeq cond1 cond2 tmp1 T tmp2 tmp3 tmp4;
  }
```

The Recursive Do Solution

We can do even better than the `Haskell98Solution` with a language extension. GHC has a keyword `mdo` that stands for μ-do, or recursive do. There are two major differences between a `do` block and an `mdo` block. In an `mdo` block variables assigned to cannot be repeated. So the code

```
mdo -- ILLEGAL EXAMPLE
 c <- getChar
 c <- getChar
```

is not acceptable. The second major difference is that you can refer to variables that have not yet been assigned.

```
mdo
 a <- return b
 b <- [1..3]
 return a
```

You can see that if multiple assignments were allowed, it would be unclear which assignment a variable refers to. Unfortunately `mdo` does not work with every monad. For `mdo` to work an instance of `MonadFix` is required. Fortunately most monads are instances of `MonadFix`.

Using a variable before its assignment carries the same risks of time travel discussed in the previous section. The value is from the future and one must not do anything with it that **might** change the future.

Let us recreate our assembler for use with `mdo`.

```
newtype AssemblyCodeMonad a =
  AssemblyCodeMonad (RWS () [Instruction Register] Location a)
  deriving (Monad, MonadFix)

type AssemblyCode = AssemblyCodeMonad ()
```

The `AssemblyCodeMonad` is simpler this time. There is no oracle, so I use `()` for the reader part. The writer part only outputs instructions. The state needs to only keeps track of the current instruction number.

```
newtype Location = Location Word32
```

The `writeInstruction` function is a little simpler than before because the `AssemblyCodeMonad` is simpler.

```
writeInstruction :: Instruction Register -> AssemblyCode
writeInstruction i = AssemblyCodeMonad $ do
  tell [i]
  modify (\(Location l) -> Location (succ l))
```

Commands for making `AssemblyCode` instructions are exactly the same as before.

```
mneqz cond src dst = do
  writeInstruction $ CMove dst src cond
```

I postpone the rest of the instructions until the end and move directly to managing labels. The `lset` command is exactly the same as before.

```
lset (Location v) dst T tmp = set32 v dst T tmp
```

Acquiring a label is easier than before. Instead of the command taking a label name, all it does is return the location of the current instruction counter

```
label :: AssemblyCodeMonad Location
label = AssemblyCodeMonad get
```

For an example of using label, consider again the jump-if-equal-zero instruction.

```
jeqz cond offset T tmp1 tmp2 = mdo {
      lset skip tmp1 T tmp2;
      mneqz cond offset tmp1;
      jmp offset T tmp1;
  skip <- label;
  return () }
```

Here the a label named `skip` is created near the end, but the recursive do allows one to refer to this label at the beginning with the `lset` instruction. This is the same sort of circular programming that we had in the `Haskell98Solution`, but now `mdo` is handling all the work. Also notice that one no longer needs to use `fresh` to create a globally unique name. One uses Haskell variables for the label names, and they have the scope of the entire `mdo` block.

The `link` function now simply executes the monad. The `mdo` structure handles all the circular programming, so that there is no need to tie the knot by hand.

```
link :: AssemblyCode -> [Instruction Register]
link (AssemblyCodeMonad mc) = asm
 where
  ((), _, asm) = runRWS mc () (Location 0)
```

The result is that `mdo` allows one to create an assembler syntax with label that works in a easy and clear way. Look at our example program from the introduction again.

```
testMC :: AssemblyCode
testMC = mdo {
  start <- label;
        iout '>' T Reg7;
        inp  Reg0;
        high Reg1;
        ljeq Reg0 Reg1 end T Reg4 Reg5 Reg6 Reg7;
        out  Reg0;
        ljmp start T Reg6 Reg7;

  end <- label;
        halt;
  }
```

The code inside the `mdo` block looks like an assembly language, but it really is Haskell. Because it is Haskell, all the functionality of the Haskell language is available to the programmer for the "macro" language of this assembler.

Now that you have seen one example of how `mdo` can be used, I hope you it will become part of your toolkit for solving problems. For some practice with recursive do, try the exercises at the end of the article.

The rest of the instructions and "macros" are included in the appendix.

About the author

Russell O'Connor has a bachelor's degree in pure mathematics and computer science from the University of Waterloo. He is currently pursuing a PhD on efficient correct exact real arithmetic in the Foundations Group at Radboud University Nijmegen.

Exercises

1. Show that you can make mutually dependent assembly modules using `mdo`. Write one module that consumes and echo characters until the `<` character is read. When that happens make a tail call (a jump) to a `Location` passed as a parameter. Write another module that consumes and does not echo characters until the `>` character is read. When that happens make a tail call to a `Location` passed as a parameter. Have both modules return the `Location` of where they begin. Now put the two together to create a program that removes "tags" (strings between and including `<` and `>`) from input.

2. Create a new type of command to write to a data segment that will be located at the end of the machine code. It should have the following signature:

   ```
   data :: [Word32] -> MachineCodeM Location
   ```

   Use circular programming to initialize the data segment counter to the final value of the instruction counter.

3. The `set32` instruction is quite long. It would be faster to load the location from the data segment. Unfortunately to load from the data segment requires loading the location in the data segment where the location that you want is stored. Move the data segment to the beginning of the address space and limit data segment locations to 25 bits. Now implement `set32` by storing the location in the data segment and loading the location (using `set25`). Revel in the amount of circular programming you now have, but don't forget to add a jump instruction at the very beginning to jump over the data segment.

4. Implement `push` and `pop` instructions that operate on a specified fixed size stack allocated in array 1.

5. Implement `call` and `return` instructions that saves and loads registers using the stack developed in the previous exercise.

6. It may be advantageous to constantly keep one register set to 0 or one register set high. Reduce the number of registers in the assembly language and

rewrite the macros to take advantage of one register that is constantly 0 or one register that is constantly high.

Appendix

```
read arr ix dst    = writeInstruction $ Read dst arr ix
write src arr ix   = writeInstruction $ Write arr ix src
add src1 src2 dst  = writeInstruction $ Add dst src1 src2
mul src1 src2 dst  = writeInstruction $ Mul dst src1 src2
div src1 src2 dst  = writeInstruction $ Div dst src1 src2
nand src1 src2 dst = writeInstruction $ Nand dst src1 src2
halt               = writeInstruction $ Halt
alloc size dst     = writeInstruction $ Alloc dst size
free ptr           = writeInstruction $ Free ptr
out src            = writeInstruction $ Output src
inp dst            = writeInstruction $ Input dst
load ptr offset    = writeInstruction $ Load ptr offset
-- Orth can only write 25 bit numbers
set25 val dst      = writeInstruction $ Orth dst val

data T = T

not src dst =
      nand src src dst

neg src dst T tmp = do
      set25 0 tmp
      not tmp tmp
      mul src tmp dst

sub src1 src2 dst T tmp = do
      neg src2 src2 T tmp
      add src1 src2 dst
-- return src2 to it's original value only if
-- src2 isn't the destination
      unless (dst==src2) $ neg src2 src2 T tmp

shftG op n src dst T tmp
 | n < 25 = do
      set25 (2^n) tmp
      op src tmp dst
 | otherwise = do
      shftG op 24 src dst T tmp
      shftG op (n-24) dst dst T tmp
```

```
shftr = shftG div
shftl = shftG mul

set32 val dst T tmp = do
      set25 (val `shiftR` 16) dst
      shftl 16 dst dst T tmp
      set25 (val .&. 0xFFFF) tmp
      add tmp dst dst

jmp offset T tmp = do
      set25 0 tmp
      load tmp offset

ljmp l T tmp1 tmp2 = do
      lset l tmp1 T tmp2
      jmp tmp1 T tmp2

-- immediate out
iout c T tmp = do {
      set25 (fromIntegral $ fromEnum c) tmp;
      out tmp;
  }

zero dst = do {
      set25 0 dst;
  }

-- set register to all ones
high dst = do {
      zero dst;
      not dst dst;
  }

jeq cond1 cond2 offset T tmp1 tmp2 tmp3 = do {
      sub cond1 cond2 tmp1 T tmp2;
      jeqz tmp1 offset T tmp2 tmp3;
  }
```

```
ljeqz cond l T tmp1 tmp2 tmp3 = do {
      lset l tmp1 T tmp2;
      jeqz cond tmp1 T tmp2 tmp3;
  }

ljeq cond1 cond2 l T tmp1 tmp2 tmp3 tmp4 = do {
      lset l tmp1 T tmp2;
      jeq cond1 cond2 tmp1 T tmp2 tmp3 tmp4;
  }

assemble = encodeAsm . link
```