

Evaluating Haskell in Haskell

by Matthew Naylor <mf@cs.york.ac.uk>

*There are several Haskell interpreters, such as Hugs and GHCi, but none (as far I'm aware) are implemented in Haskell. For performance reasons, C appears to be the implementation language of choice. This article presents a **simple** Haskell interpreter – called the Haskell Haskell interpreter, or **hhi** for short – that is **written in Haskell** and that performs quite competitively with Hugs and GHCi. The key idea is to inherit the graph reduction capabilities of the host Haskell implementation by compiling expressions to use a finite, fixed set of combinators, and then implementing these combinators just as Haskell functions. The idea is an old one, but I would like it not to be forgotten. I'll give timing comparisons with Hugs and GHCi on a handful of benchmark programs, and suggest a few avenues for future exploration.*

Core Language

A good way to start writing an interpreter for Haskell is to borrow the parser, type checker and desugarer of an existing Haskell implementation. And a particularly easy way to do this is to use Yhc [1]. Yhc can generate simple, untyped, core Haskell files (called Yhc.Core files) that can be easily read and processed using the Yhc.Core library [2].

However, although the syntax of Yhc.Core is very easy to understand, it is fairly rich. For example, expressions can be variables, constructors, functions, literals, applications, cases, lets, or lambdas. The annoying thing you realise when writing an evaluator for Yhc.Core is that there are four separate constructs for binding values to variables! Thankfully, using transformations such as **let elimination** and **Jansen's encoding**¹ [5] (based on Berarducci and Bohm's encoding [6]), Yhc.Core can be reduced to an even simpler core language, which I'll call **hhi core**, consisting

¹Bernie Pope points out that Stump [3] refers to this encoding as Scott's encoding, crediting a 1963 lecture by Dana Scott [4].

```
type Prog = [(FunId, Exp)]

type VarId = String

type FunId = String

data Exp = Ap Exp Exp
         | Lambda VarId Exp
         | Fun FunId
         | Var VarId
         | Int Int
         | Lam (Exp -> Exp)  -- only used internally
```

Figure 1: The syntax of hhi core

```
data List a = Nil | Cons a (List a)

powerset xs =
  case xs of
    Nil -> Cons Nil Nil
    Cons x xs -> let p = powerset xs in
                  p ++ map (Cons x) p
```

Figure 2: The powerset function in Yhc.Core (assuming suitable definitions of map and ++)

```
Nil = \n -> \c -> n

Cons = \x -> \xs -> \n -> \c -> c x xs

powerset =
  \xs -> xs (Cons Nil Nil)
           (\x xs -> (\p -> p ++ map (Cons x) p) (powerset xs))
```

Figure 3: The powerset function in hhi core

of just 0-arity function definitions, variables, lambdas, applications and literals. The syntax of `hhi` core is given formally in Figure 1. For simplicity, only integer literals are supported, and not, for example, `Float` and `Double`).

Figures 2 and 3 illustrate the difference between a simple program, the `powerset` function, in `Yhc.Core` and the same program in `hhi` core. For information about how to translate the former function into the latter, the interested reader is invited to download `hhi`, or to read Jan Martin Jansen’s paper about SAPL [5] and Simon Peyton Jones’s book [7] (which discusses let elimination amongst many other transformations).

Interpreting `hhi` core

Even after `Yhc.Core` has been reduced to a simple lambda calculus with literals and named expressions, there still remains the problem of writing an interpreter for it in Haskell. The most obvious approach, perhaps, is to define it in classic Haskell style, as a set of recursive equations that do case analysis on the expression to be reduced. An interpreter implemented in this way is shown in Figure 4. Again, for simplicity, only a handful of primitive functions over integers are supported.

Unfortunately, the interpreter in Figure 4 is rather inefficient for a number of reasons, some trivial and some rather more serious:

- ▶ It looks up a function definition in a list every time a function is called.
- ▶ It uses an **explicit** substitution function that heavily traverses and deconstructs existing expressions, and reconstructs new ones. It is difficult to make this critical function as efficient as it would be in a low-level language like C.
- ▶ It has a serious **space leak**, making it somewhat impractical. The problem is that the `subst` function is lazy and may hold on to arguments even if they are not referenced in the expression under substitution.
- ▶ It’s non-strict, **but not lazy!** Lazy evaluation states that expressions are evaluated **at most once**, yet when `eval` applies an argument to a lambda, the argument is substituted, **unevaluated**, in the body of the lambda. If the argument is referenced more than once, evaluation of an expression may be repeated. It may seem trivial to fix this problem: just wrap the argument in a call to `eval`. The problem with this is that when a value participates as the argument in numerous beta reductions, then it becomes deeply wrapped in calls to `eval`, which is inefficient.

The good news is that these efficiency problems can be solved with an implementation that is actually simpler than the one in Figure 4!

```
interp :: Prog -> Exp
interp p = eval p main
  where
    main = fromJust (lookup "main" p)

eval :: Prog -> Exp -> Exp
eval p (Ap (Ap (Fun "ADD_W" a)) b) = arith2 p (+) a b
eval p (Ap (Ap (Fun "SUB_W" a)) b) = arith2 p (-) a b
eval p (Ap (Ap (Fun "EQ_W" a)) b) = logical2 p (==) a b
eval p (Ap (Ap (Fun "NE_W" a)) b) = logical2 p (/=) a b
eval p (Ap (Ap (Fun "LE_W" a)) b) = logical2 p (<=) a b
eval p (Ap f a) = eval p (subst v a b)
  where
    Lambda v b = eval p f
eval p (Fun f) = eval p (fromJust (lookup f p))
eval p e = e

subst :: VarId -> Exp -> Exp -> Exp
subst v e (Var w) = if v == w then e else Var w
subst v e (Ap e0 e1) = Ap (subst v e e0) (subst v e e1)
subst v e (Lambda x b) = Lambda x (if v == x then b else subst v e b)
subst v e b = b

arith2 p op a b = Int (op x y)
  where
    Int x = eval p a
    Int y = eval p b

logical2 p op a b = if op x y then true else false
  where
    Int x = eval p a
    Int y = eval p b

true = Lambda "t" $ Lambda "f" $ Var "t"
false = Lambda "t" $ Lambda "f" $ Var "f"
```

Figure 4: A naive interpreter

Efficiently interpreting hhi core

The fundamental problem with the interpreter in Figure 4 is that beta reduction is emulated on top of Haskell instead of utilising the host Haskell system's support for beta reduction. **What we want to do is take an expression in abstract syntax and replace its Lambda and Ap constructors with Haskell's built-in lambda and application constructs.** More concretely, we'd like to write a Haskell function to take a definition in abstract syntax such as

```
("flip", Lambda "f" (Lambda "a" (Lambda "b"
    (Ap (Ap (Var "f") (Var "b") (Var "a"))))))
```

and to return

```
("flip", Lam (\f -> Lam (\a -> Lam (\b ->
    ap (ap f b) a))))
```

where `ap (Lam f) a = f a`.

At first sight, the idea seems to fall down: it is not possible to write such a conversion as a Haskell function. The problem is that the body of a lambda must do some computation to construct the desired body of the combinator (`flip` in this case), and this computation must be done every time the lambda is applied.

So an arbitrary function definition occurring within an input program cannot have its `Lambda` and `Ap` nodes directly converted (by the interpreter) to Haskell lambdas (`Lam`) and applications (`ap`). However, if the function is known statically, **in advance**, there is no problem because it can be manually hard-coded into the interpreter. Now, if only there was a way to turn core expressions into applications of a fixed set of statically-known combinators then **all** necessary functions could be hard-coded. And, of course, thanks to Turner, there is!

Turner's SK compilation scheme is well documented [7, 8]. Applying it to `hhi` core leads to the simple, efficient interpreter shown in Figure 5. Notice that functions are no longer looked up in a list every time they are called. Instead, function names are linked to their bodies once and for all by `link`. All of the other efficiency concerns with the interpreter in Figure 4 disappear now that an explicit substitution function is unnecessary.

The idea to use SK combinators for interpreting functional programs in functional language is not new. It was realised at least as long as 20 years ago by Lennart Augustsson in a lazy untyped functional interpreter called `SMALL` [9]. (Thanks to Colin Runciman for pointing me to the paper on `SMALL`.) I've simply rehabilitated the method to see how well it works in the modern, statically-typed language, Haskell. Another difference between `hhi` and `SMALL` is that `SMALL` interprets itself, bootstrapped by a simple reduction machine written in C.

```

interp :: Prog -> Exp
interp p = fromJust (lookup "main" bs)
  where
    bs = prims ++ map (\(f, e) -> (f, link bs e)) p

link bs (Ap f a) = link bs f ! link bs a
link bs (Fun f) = fromJust (lookup f bs)
link bs e = e

infixl 0 !
(Lam f) ! x = f x

prims = let (-->) = (,) in
  [ "I"   --> (Lam $ \x -> x)
  , "K"   --> (Lam $ \x -> Lam $ \y -> x)
  , "S"   --> (Lam $ \f -> Lam $ \g -> Lam $ \x -> f!x!(g!x))
  , "B"   --> (Lam $ \f -> Lam $ \g -> Lam $ \x -> f!(g!x))
  , "C"   --> (Lam $ \f -> Lam $ \g -> Lam $ \x -> f!x!g)
  , "S'"  --> (Lam $ \c -> Lam $ \f -> Lam $ \g -> Lam $ \x ->
              c!(f!x)!(g!x))
  , "B*"  --> (Lam $ \c -> Lam $ \f -> Lam $ \g -> Lam $ \x ->
              c!(f!(g!x)))
  , "C'"  --> (Lam $ \c -> Lam $ \f -> Lam $ \g -> Lam $ \x ->
              c!(f!x)!g)
  , "ADD_W" --> arith2 (+)      , "SUB_W" --> arith2 (-)
  , "EQ_W"  --> logical2 (==) , "NE_W"  --> logical2 (/=)
  , "LE_W"  --> logical2 (<=)
  ]

arith2 op = Lam $ \(Int a) -> Lam $ \(Int b) -> Int (op a b)

logical2 op =
  Lam $ \(Int a) -> Lam $ \(Int b) -> if op a b then true else false

true = Lam $ \t -> Lam $ \f -> t
false = Lam $ \t -> Lam $ \f -> f

```

Figure 5: An efficient interpreter

Program	Hugs	GHCi	hhi
Fib	12.3s	4.6s	1.2s
Queens	6.4s	6.3s	4.2s
Sudoku	2.0s	0.9s	1.4s
PermSort	3.0s	2.9s	4.2s
SumPuz	4.1s	3.4s	3.8s
OrdList	4.5s	3.7s	7.1s
Simplifier	2.4s	2.8s	7.2s

Figure 6: Timings of programs running on various Haskell interpreters.

Performance measurements

Timings of hhi, Hugs, and GHCi running a range of benchmark programs are shown in Table 6.

The fact that hhi performs so well on integer intensive benchmarks (Fib and Queens) is probably because Yhc removes type-class dictionaries in several cases, whereas GHCi and Hugs do not. The fairest comparison is OrdList, because it uses no type classes at all.

The main benchmark that hhi performs poorly on is Simplifier. Simplifier differs from the other benchmarks in that it uses quite a large algebraic data type (5 constructors) for logical formulae. It would appear that large data types are the source of the problem.

The public darcs repo for hhi is at:

<http://www.cs.york.ac.uk/fp/darcs/hhi/>

Avenues for exploration

Turner's combinators work quite well, but some quick experiments show that the addition of just a few new ones,

```
C0 = Lam $ \a -> Lam $ \b -> Lam $ \c -> c!a!b
C1 = Lam $ \a -> Lam $ \b -> Lam $ \c -> c!a
C2 = Lam $ \a -> Lam $ \b -> Lam $ \c -> Lam $ \d -> d!a!b
```

gives a performance improvement of around 30% in some cases. I tried these combinators because they represent commonly used Jansen-encoded constructors. For example, C0 encodes pairs and C2 encodes (:). This suggests that new combinators (beyond Turner's) will aid efficient reduction of Jansen-encoded constructors.

Avenue 1 (New Combinators). *Add new combinators to hhi, and measure the performance improvement.*

A possible sledge-hammer solution would be to write a program that generates a large number of suitable combinators – see, for example, the interesting work on director strings by Kennaway [10] and Stoye [11]. Alternatively, data types and functions from the Prelude could be added directly as combinators to hhi.

Another way to have fun with hhi might be to experiment with Haskell’s evaluation strategy.

Avenue 2 (Strict Haskell). *Adjust hhi to do strict evaluation, and compare with lazy evaluation on some existing Haskell programs.*

Redefining application as `Lam f ! a = seq a (f a)` and adding a few strictness annotations to the `Exp` data type might work, but it has not yet been tried. Since case expressions have been turned into function applications, we would have to take special care not to evaluate all the case alternatives before the case subject. (Although, it wouldn’t be incorrect to do so, just inefficient for nullary constructors like `True` and `False`.)

Another possibility would be to experiment with new language features in Haskell.

Avenue 3 (Non-determinism). *Add a non-deterministic choice operator to Haskell,*

`(?) :: a -> a -> a`

so that the evaluator now delivers zero or more results rather than just one. For example, evaluating `(1?2)+3` would give both 4 and 5 as results.

This feature gives Haskell an evaluation strategy similar to **narrowing**, as found in the functional logic language Curry [12]. Unfortunately, it seems necessary to adopt a little `unsafePerformIO` to implement this feature correctly. It is difficult to add side-effects to the interpreter while maintaining the host implementation’s support for graph reduction; `interp` is a pure function, and turning it into a monadic one loses laziness (either evaluation order or sharing of computations).

The interested reader is encouraged to explore these or other aspects of evaluating Haskell using hhi – it should prove to be a happy playground!

Acknowledgements

Thanks to Wouter Swierstra and Bernie Pope for their helpful comments, suggestions, and corrections.

About the author

Matthew Naylor is a member of the programming group at York, working on a thesis about hardware-assisted and target-directed evaluation.

References

- [1] The Yhc Team. The York Haskell Compiler - user's guide. <http://www.haskell.org/haskellwiki/Yhc> (February 2007).
- [2] Dmitry Golubovsky, Neil Mitchell, and Matthew Naylor. Yhc.Core - from Haskell to Core. **The Monad.Reader**, (7):pages 45–61 (April 2007).
- [3] Aaron Stump. Directly reflective meta-programming. **Journal of Higher Order and Symbolic Computation**, page to appear (2008).
- [4] Dana Scott. A system of functional abstraction (1968). Lectures delivered at University of California, Berkeley, Cal., 1962/63. Photocopy of a preliminary version, issued by Stanford University, September 1963, furnished by author in 1968.
- [5] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In **Trends in Functional Programming**, volume 7. Intellect (2007).
- [6] Alessandro Berarducci and Corrado Böhm. A self-interpreter of lambda calculus having a normal form. In **CSL '92: Selected Papers from the Workshop on Computer Science Logic**, pages 85–99. Springer-Verlag, London, UK (1993).
- [7] Simon Peyton Jones. **The Implementation of Functional Programming Languages**. Computer Science. Prentice-Hall (1987).
- [8] D. A. Turner. A new implementation technique for applicative languages. **Softw., Pract. Exper.**, 9(1):pages 31–49 (1979).
- [9] Lennart Augustsson. Small – a small interactive functional system. Technical Report 28, Programming Methodology Group, University of Göteborg and Chalmers University of Technology (1986).
- [10] Richard Kennaway and Ronan Sleep. Director strings as combinators. **ACM Transactions on Programming Languages and Systems**, 10(4):pages 602–626 (1988).
- [11] William Stoye. The Implementation of Functional Languages using Custom Hardware. PhD Thesis, University of Cambridge (1985).
- [12] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A truly functional-logic language. In **ILPS'95 Post Conference Workshop on Declarative Languages for the Future**. Portland State University and ALP, Melbourne University (1995).