MicroHs: A Small Compiler for Haskell

Lennart Augustsson
Unaffiliated
Goteborg, Sweden
lennart@augustsson.net

Abstract

MicroHs is a compiler for Haskell2010. It translates Haskell to **SKI** style combinators via λ-calculus. The runtime system is quite small with few dependencies.

CCS Concepts: • **Software and its engineering → Compilers**; **Interpreters**; *Runtime environments*; **Functional languages**.

Keywords: Haskell, compilers, combinators, runtime system

ACM Reference Format:
Lennart Augustsson. 2024. MicroHs: A Small Compiler for Haskell. In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium (Haskell '24), September 6–7, 2024, Milan, Italy.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3677999.3678280

1 Introduction

MicroHs is a compiler and interactive system for Haskell2010 [2]. It also implements many common extensions. It translates Haskell, via λ-calculus, to **SKI** combinators, following in the footsteps of Turner [4].

The source code for the compiler is less than 10000 lines of Haskell, out of which the type checker is about 30%.

The runtime system uses a mark-sweep garbage collector. Additionally, it is quite modular in the sense that it can be compiled with or without features like file access *etc.*. Excluding most features, the runtime only basically needs memory allocation and can thus be compiled for very simple systems, *e.g.*, micro-controllers running on the bare metal.

The runtime system (with all features) is less than 5000 lines of C.

2 Language

MicroHs implements the full Haskell2010 language and libraries, with some exceptions and extensions.

Haskell '24, September 6–7, 2024, Milan, Italy

ACM ISBN 979-8-4007-1102-2/24/09
https://doi.org/10.1145/3677999.3678280

```
BangPatterns ConstraintKinds DoAndIfThenElse
DuplicateRecordFields EmptyDataDecls
ExistentialQuantification ExtendedDefaultRules
FlexibleContexts FlexibleInstance ForeignFunctionInterface
FunctionalDependencies GADTs GADTsyntax IncoherentInstances
KindSignatures MonoLocalBinds MultiParamTypeClasses
NamedFieldPuns NegativeLiterals NoMonomorphismRestriction
NoStarIsType OverlappingInstances OverloadedRecordDot
OverloadedRecordUpdate OverloadedStrings PolyKinds
RankNTypes RecordWildCards QualifiedDo ScopedTypeVariables
StandaloneKindSignatures TupleSections TypeLits
TypeSynonymInstances UndecidableInstances
UndecidableSuperClasses ViewPatterns
```

Figure 1. Language Extensions

Missing features (nothing fundamental, just due to my laziness):

- There is no deriving of the `Read` class.
- The types `Float` and `Double` have the same number of bits. The size is the word size of the platform, *i.e.*, 32 bit float on 32 bit machines and 64 bit floats on 64 bit machines.
- The FFI is basically limited to `foreign import ccall`.
- Some FFI libraries are missing, like `Foreign.StablePtr`.

A number of Haskell language extensions are also available, see Fig. 1. These are permanently enabled; there is no picking and choosing, you get what you get.

The implemented extensions are GHC compatible, but GHC has many more extensions which can make porting code from GHC to MicroHs difficult. MicroHs will probably gain more extensions over time, but its goal is not to be the kitchen sink.

3 Compiler Overview

The compiler structure is mostly unremarkable.

As each module is encountered, all of its imports are automatically compiled first (with mutual recursion handled the same way as GHC). When a module has been compiled, it is saved (as a combinator graph) in a cache in the compiler to avoid recompilation in this run of the compiler.

3.1 Lexing and Parsing

The tokenization is done by a handwritten lexer and a combinator parser. A difficulty with parsing Haskell is that in addition to being sensitive to indentation, there is also a rule

that specifies that certain parse errors should be considered ending a block. To implement this the lexer is actually a state machine. The lexer is given the input list of characters and returns a state machine with two operations: *get-next-token* and *pop-layout*. By using both of these, the parser has enough communication with the lexer to implement the layout processing correctly.

3.2 Type Checking

The type checker follows Peyton Jones et al. [3] with some extensions. In addition to checking the types of terms, the compiler also needs to check the kinds of types, as well as the sorts of kinds (MicroHs uses a stratified type system; no *type-in-type*). It is in fact the same type checker that is used for all of these, since terms, types, and kinds share the same abstract syntax.

As terms are type checked, the checker collects constraints. Most of the constraints are type equalities, and these are solved incrementally using unification in the usual way. Some constraints are related to overloading (*e.g.*, `Num a`) and these are collected and solved at certain points (*e.g.*, when a top level definition is done).

Overloading is implemented using dictionaries—like most Haskell compilers do. The type checker rewrites the input term by inserting dictionaries for all overloaded identifiers, and these dictionaries are then defined when the type checker solves the constraints.

3.3 Desugaring

Desugaring is pretty straightforward and follows the description from the Haskell report on how to translate Haskell to Core Haskell.

After this step all that remains is λ-calculus with recursive **let**, data constructors, **case**-expressions, primitives, and literals.

Non-recursive **let**-expressions are simply translated into an application, whereas recursive **let**-expressions are translated using the **Y** combinator.

3.4 Encoding Data Types

Part of desugaring is getting rid of data constructors and **case**-expressions. Data types can be encoded using just Scott-encoding [5]. A problem with this is that with a large number of constructors the λ-terms get quite large. MicroHs uses Scott encoding up to a cutoff point (5 constructors). For more constructors, each constructors is encoded as a pair (which is Scott encoded) consisting of an `Int` tag (the constructor number) and a tuple (also Scott encoded) of all the arguments. Using only Scott-encoding makes the MicroHs compiler about 25% slower.

After this step, there is only λ-calculus, primitives, and literals left.

3.5 Optimisation

The compiler has a single optimization: simplifying recursive calls where a prefix of the arguments remains the same. *E.g.*,

```
map f [] = []
map f (x:xs) = f x : map f xs
```

is transformed to

```
map f = g
  where g [] = []
        g (x:xs) = f x : g xs
```

This optimization is important for recursive overloaded functions. An overloaded function typically has the same dictionary argument in the recursive calls. This transformation makes it possible to do reductions given only the dictionary. This will make the combinator reduction automagically specialize overloaded functions for each call site.

3.6 Code Generation

Code generation is simply bracket abstraction to remove all lambdas and replace them with a fixed set of combinators. This follows Turner [4] exactly. *I.e.*, code generation first produces **SKI** naïvely and in the same pass applies rewrites to simplify those into the other combinators.

Recursion is handled differently on the top level and locally. Local recursion is translated into the **Y** combinator. At the top level, each definition is just translated separately, so at the end we have a number of definitions that refer to each other by name. This exactly the format of a serialized graph (see Section 6.3) and is what the runtime system expects as the input.

3.6.1 Generating Files.

The output of the compiler is a C file that contains the combinator code for the compiled program. It is simply the combinator tree rendered as a text in postfix form, but in the form of an `int8[]` array. To reduce the size, the text can optionally be compressed by a simple Lempel-Ziv compressor [6].

If the program contains FFI calls to C functions, the generated C file will also have stubs that do the necessary type conversions and call their C function.

3.7 Interactive System

Instead of generating a file with the combinators, the combinator tree can be converted into the term it represents by simply replacing all the combinators and primitives with their semantic equivalent. This way the "compiler" can run the code instead of generating a file. A sketch of the code is in Fig. 2; it looks scary, but for type-checked programs it is safe.

Combined with a simple REPL this becomes an interactive system, akin to GHCI.

```
data Exp = App Exp Exp | Comb String | Int Integer

trans :: Exp -> Any
trans (App e1 e2) = unsafeCoerce (trans e1) (trans e2)
trans (Comb s)    = fromJust $ lookup s combinators
trans (Int i)     = unsafeCoerce i

combinators :: [(String, Any)]
combinators = [
  [("S", unsafeCoerce $ \ f g x -> f x (g x))
  ,("K", unsafeCoerce $ \ x y -> x)
  ,("+", unsafeCoerce   (+)
  , ...
  ]
```

Figure 2. Translating syntax to values

4 Bootstrapping

The MicroHs compiler can (of course) recompile itself. To make it easy to get up and running with MicroHs, the distribution contains not just the source code for the compiler, but also the combinator code in the form of a C file.

So to get a running compiler, you only need a C compiler and minimal C libraries. On an ordinary laptop/desktop, the C compiler can compile the combinator file together with the runtime system in about 1-2s. To get a running MicroHs simply do:

```
git clone git@github.com:augustss/MicroHs.git
make
```

The MicroHs compiler can then recompile itself from scratch in about 20s.

5 Combinators

The set of combinators used, Fig. 3, is the standard **SKI** with usual extensions of **BC**. There are also the variants, **S' B' C'**, suggested by Turner [4]. There is an explicit **Y** combinator, since it gets special treatment in the runtime system—it creates a cycle instead of copying.

In addition, there are some combinators that are quite *ad hoc*. They were added after looking (programatically) at the combinator code and finding some repeated patterns. Some of these correspond to common idioms, *e.g.*, **A** is `True`, **P** is `(,)`, **O** is `(:)`, and **U** does pair pattern matching (= `uncurry`). Others are just common patterns. This is a set that works well; each combinator was added after carefully measuring that it improved things (*e.g.*, adding **K5** makes it slower).

I also tried a more principled approach, as described in Kiselyov [1], but it resulted in slower and bigger code.

Runtime Optimization. As pointed out in Turner [4], combinators have a *self-optimising* property. This is simply the fact that functions can reduce just given a prefix of the arguments. For example:

```
f x y = x * y
g = f 2
```

S *x y z*	*x z (y z)*
K *x y*	*x*
I *x*	*x*
B *x y z*	*x (y z)*
C *x y z*	*x z y*
S' *x y z w*	*x (y w) (z w)*
B' *x y z w*	*x y (z w)*
C' *x y z w*	*x (y w) z*
A *x y*	*y*
U *x y*	*y x*
Y *x*	*x (**Y** x)*
Z *x y z*	*x y*
P *x y z*	*z x y*
R *x y z*	*y z x*
O *x y z w*	*w x y*
K2 *x y z*	*x*
K3 *x y z w*	*x*
K4 *x y z w v*	*x*
C'B *x y z w*	*x z (y w)*

Figure 3. Combinators

After using `g` once, the definition simply becomes (showing the non-combinator version for readability):

```
g y = 2 * y
```

This is a very important property for overloaded functions because it means that calls to overloaded functions will specialize automatically at runtime. For example:

```
inc :: Num a => a -> a
inc x = x + 1

iinc :: Int -> Int
iinc = inc
```

The combinator code for these two definitions becomes (after one use of `iinc`):

```
inc = U (S' C (U (Z (Z (Z (Z (Z K))))))
        ((C (U (K (K4 A)))) (P K (O 1 K))))

iinc = C + 1
```

This optimization happens completely automatically just by the way that combinators reduction works.

The large number of combinators in `inc` are there to select `(+)` and the (implicit) `fromInteger` from the `Num` dictionary.

6 Runtime System

The runtime system is written in C. It has code for memory management, FFI, file IO, and (of course) an interpreter for the combinator code that handles all the graph rewriting of combinators and primitives.

```
void eval(NODEPTR n) {
 for(;;) {
  tag = GETTAG(n);
  switch(tag) {
  case T_IND:  n = INDIR(n); break;
  case T_AP:   PUSH(n); n = FUN(n); break;
  case T_INT:  RET;
  case T_S:    GCCHECK(2); CHKARG3; GOAP(new_ap(x, z), new_ap(y, z));             /* S x y z = x z (y z) */
  case T_SS:   GCCHECK(3); CHKARG4; GOAP(new_ap(x, new_ap(y, w)), new_ap(z, w));  /* S' x y z w = x (y w) (z w) */
  case T_K:                CHKARG2; GOIND(x);                                     /* K x y = *x */
  case T_A:                CHKARG2; GOIND(y);                                     /* A x y = *y */
  case T_U:                CHKARG2; GOAP(y, x);                                   /* U x y = y x */
  case T_I:                CHKARG1; GOIND(x);                                     /* I x = *x */
  case T_Y:                CHKARG1; GOAP(x, n);                                   /* n@(Y x) = x n */
  case T_B:    GCCHECK(1); CHKARG3; GOAP(x, new_ap(y, z));                        /* B x y z = x (y z) */
  case T_BB:   GCCHECK(2); CHKARG4; GOAP(new_ap(x, y), new_ap(z, w));             /* B' x y z w = x y (z w) */
  case T_Z:                CHKARG3; GOAP(x, y);                                   /* Z x y z = x y */
  case T_C:    GCCHECK(1); CHKARG3; GOAP(new_ap(x, z), y);                        /* C x y z = x z y */
  case T_CC:   GCCHECK(2); CHKARG4; GOAP(new_ap(x, new_ap(y, w)), z);             /* C' x y z w = x (y w) z */
  case T_P:    GCCHECK(1); CHKARG3; GOAP(new_ap(z, x), y);                        /* P x y z = z x y */
  case T_R:    GCCHECK(1); CHKARG3; GOAP(new_ap(y, z), x);                        /* R x y z = y z x */
  case T_O:    GCCHECK(1); CHKARG4; GOAP(new_ap(w, x), y);                        /* O x y z w = w x y */
  case T_ADD:  ARITHBIN(+);
  case T_SUB:  ARITHBIN(-);
  case T_IO_BIND:          RET;
  ...
  }
 }
}
```

Figure 4. Combinator Evaluator

```
void exec(NODEPTR n) {
 for(;;) {
  eval(n);
  tag = GETTAG(n);
  switch(tag) {
  case T_IND:       n = INDIR(n); break;
  case T_AP:        PUSH(n); n = FUN(n); break;
  case T_IO_BIND:   CHECKIO(2);   x = execio(ARG(TOP(1))); f = ARG(TOP(2)); n = new_ap(f, x); POP(3); break;
  case T_IO_THEN:   CHECKIO(2); (void)execio(ARG(TOP(1))); n = ARG(TOP(2));                   POP(3); break;
  case T_IO_RETURN: CHECKIO(1); n = ARG(TOP(1)); RETIO(n);
  case T_IO_OPEN:   ...
  case T_IO_CCALL:  ...
  case T_IO_CATCH:  ... setjmp(jbuf); ...
  case T_IO_THROW:  ... longjmp(jbuf, 1); ...
  ...
  default: ERR("non-IO encountered");
  }
 }
}
```

Figure 5. Combinator Executor

6.1 Combinator Interpreter

The interpreter is divided into the evaluation, `eval()`, which runs pure code and the executor, `exec()`, which executes IO actions. The latter is what `unsafePerformIO` calls to run an IO action.

A sketch of the C code for these two are in Fig. 4 and Fig. 5.

6.2 Memory Management

Memory is managed with the granularity of cells. A cell is big enough to contain two pointers. For an (always binary) application node it contains two pointers, with one bit stolen from one of the pointers to indicate that it is an application. It can also contain a combinator or a primitive, as well as the primitive types of machine integers and floating point

numbers. The size of the `Int` and `FloatW` Haskell types is determined by the size of a pointer, typically 64 or 32 bits.

Garbage collection is done by first doing a mark phase of all nodes reachable from the roots (which is the top of the graph). Marking is not done in the nodes themselves, but rather in a separate bitmap. There is no separate sweep phase—instead on each node allocation, the mark bitmap is used to find the next free cell. Doing it this way is a substantial speedup over having a separate sweep phase.

Arrays are supported by simply calling `malloc()` to allocate memory for the whole array. This adds some minor complication to the garbage collector, but not much.

6.3 Serialization

The runtime system also contains code to serialize and deserialize a combinator graph. Serialization will preserve sharing and cycles. Any values can be serialized, with the exception of FFI pointers into C memory.

Serialization is a very powerful and convenient feature. *E.g.*, the program can be written out after runtime specialization has happend. Another use case is to send computations between computers.

The current implementation does not have safe deserialization, *i.e.*, there is no check that the deserialized graph has the expected type.

7 Packages and Compilation Cache

To speed up compilation, one does not want to compile everything from source all the time. MicroHs does not have any conventional separate compilation, but two other means.

A package is a set of Haskell modules (like a Hackage package) that are compiled together. When compilation is done, the internal state of the compiler cache is serialized and saved. When the compiler needs to use a package it simply deserializes the saved package and merges it into the current compilation cache.

Furthermore, when compiling a project, MicroHs can save the internal compilation cache and at the next start of the compiler it will read in the cache again. To make sure that any updated files get recompiled, all cached files have an MD5 checksum of the original file which is then checked as the old cache is loaded. Any updated files will be recompiled. This greatly speeds up repeated compilations.

It is worth mentioning that MicroHs has its own version of Cabal: MicroCabal (git@github.com:augustss/MicroCabal.git). The reason for this is that the Cabal tool is written in Haskell with so many GHC specific extensions that it cannot be compiled with MicroHs (or any other non-GHC Haskell compiler I know of). MicroCabal, on the other hand, is written in Haskell98 and can also be compiled by, *e.g.*, Hugs.

8 Future Work

There is a lot of possible future work:

- Implement more extensions. The Hackage repository is very, very GHC centric and without the same extensions as GHC, most packages cannot be compiled.
- Identify frequently used combinator subgraphs and JIT them.
- Better package management.
- ...

9 Conclusions

MicroHs is a simple and small compiler. Try it if you need a runtime with minimal dependencies.

Acknowledgements

I would like to acknowledge the contributors to the project: Gergő Érdi, James Hobson, David Johnson, Robert Krook, Janus Troelsen, and Jan-Willem Maessen. Additional thanks Jessica Augustsson for the edit.

I would also like to thank package authors (very few) who use Haskell2010 instead of Haskell with GHC-only extensions.

References

[1] Oleg Kiselyov. 2018. λ to SKI, Semantically - Declarative Pearl. In *Fuji International Symposium on Functional and Logic Programming*. https://api.semanticscholar.org/CorpusID:19138014

[2] Simon Marlow et al. 2010. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)* (2010).

[3] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17, 1 (2007), 1–82. https://doi.org/10.1017/S0956796806006034

[4] David Turner. 1979. A new implementation technique for applicative languages. *Software: Practice and Experience* 9 (1979). https://api.semanticscholar.org/CorpusID:40541269

[5] Wikipedia. 2024. Mogensen–Scott encoding — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Mogensen%E2%80%93Scott%20encoding&oldid=1217059412. [Online; accessed 04-June-2024].

[6] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714

Received 2024-06-03; accepted 2024-07-05