

Writing a self-hosting compiler for a purely functional language

(work in progress)

Balázs Kömüves
[bkomuves@gmail.com]

2021.03.24.

What is this?

This is an **experiment** to write a minimalistic self-hosting compiler in a purely functional language. It's intended both as a learning experience (for myself) and an experiment in bootstrapping.

Goals:

- ▶ self-hosting compiler
- ▶ very small code-base
- ▶ bootstrappable using a Haskell implementation
- ▶ generated code should have “acceptable” quality

Non-goals:

- ▶ a fully-featured language
- ▶ efficient compilation (the compiler can be slow)
- ▶ bootstrapping “real” compilers from this code base

Why self-host?

Why do bootstrapping / self-hosting?

- ▶ I like simple, self-contained stuff
- ▶ self-hosting is a philosophically desirable property
- ▶ future-proofness: If you can bootstrap from something simple, it won't bit-rot that easily
- ▶ it's a test of the maturity of the compiler
- ▶ the compiler itself is a very good test program!

Exhibit A: Maru, a tiny self-hosting LISP dialect was successfully revived and extended by Attila.

Exhibit B: The York Haskell Compiler (forked from `nhc98`) is now dead, apparently because *nobody knows how to build it anymore*.

The language

The syntax and semantics is based on (strict) Haskell - the idea is that the compiler should be compilable by both GHC and itself. This makes development much easier!

The language is basically untyped call-by-value lambda calculus with data constructors and recursive lets. During parsing, Haskell features like type annotations, data type declarations, imports etc are simply ignored - but the compiler itself is a well-typed program!

In fact, a type checker should be probably added, after all; but I wanted to keep it simple.

Language (non-)features

- ▶ no static type system (untyped lambda calculus)
- ▶ no data type declarations (constructors are arbitrary capitalized names)
- ▶ no module system - instead, we have C-style includes
- ▶ strict language (exceptions: if-then-else, logical and/or)
- ▶ ML-style side effects, but wrapped in a monad
- ▶ only simple pattern matching + default branch (TODO: nested patterns)
- ▶ no infix operators
- ▶ list construction syntax `[a,b,c]` is supported
- ▶ no indentation syntax (only curly braces), except for top-level blocks
- ▶ only line comments, starting at the beginning of the line
- ▶ no escaping in character / string constants yet
- ▶ (universal polymorphic equality comparison primop)

Compilation pipeline

1. lexing and parsing
2. collecting string literals
3. partition recursive lets into strongly connected components
4. TODO: eliminate pattern matching into simple branching on constructors
5. recognize primops
6. collect data constructors
7. scope checking & conversion to core language
8. TODO: inline small functions + some basic optimizations
9. closure conversion
10. TODO: compile to an SSA intermediate language
11. final code generation

The runtime system

The runtime system is relatively straightforward:

- ▶ there is a heap, which can only contain closures and data constructors
- ▶ there is stack (separate from the C stack) which contains pointers to the heap
- ▶ heap pointers are tagged using the lowest 3 bits
- ▶ heap objects fitting into 61 bits (ints, chars, nullary constructors, static functions) are not allocated, just stored in registers / on stack
- ▶ statically unknown application is implemented as a runtime primitive
- ▶ there is a simple copying GC - the GC roots are simply the content of the stack (plus statically known data)
- ▶ there are some debug features like printing heap objects
- ▶ primops: integer arithmetic; integer comparison; bitwise operations; lazy ifte/and/or; basic IO.

Current state of the experiment

Repo: <https://github.com/bkomuves/nanohs>

Current state:

- ▶ it can compile itself successfully;
- ▶ the source code is not as nice as it could be;
- ▶ the generated code is pretty horrible;
- ▶ there are some desirable features missing.

Code size:

- ▶ the compiler: ~ 1700 lines (required code)
- ▶ type annotations: $+500$ lines (ignored)
- ▶ the C runtime: ~ 600 lines (includes some debug features)

Mistakes I made

I made a huge amount of mistakes during the development, which made it much harder and longer than I expected.

- ▶ Trying to self-host before the compiler actually works. Writing a compiler is hard work, you don't want to do it in a limited language!
- ▶ Not figuring out the precise semantics upfront:
 - ▶ exact details of laziness vs. strictness
 - ▶ controlling exactly where heap / stack allocations can happen
 - ▶ recursive lets
- ▶ Not having debugging support from early on (for example: good error messages, threading names & source locations, RTS features, pretty printing AST-s)
- ▶ Trying to target assembly first, instead of a simpler target like C; it's just extra cognitive load
- ▶ Generatic code directly, without having a low-level IL
- ▶ Using de Bruijn indices (as opposed to levels) in generated code

Example GC bug - Marshalling from C strings

```
heap_obj marshal_from_cstring(char *cstring) { ...
```

```
v1:  obj    = heap_allocate(Cons,2);  
      obj[1] = cstring[0];  
      obj[2] = marshal_from_cstring(cstring+1);  
      return obj;
```

```
v2:  rest   = marshal_from_cstring(cstring+1);  
      obj    = heap_allocate(Cons,2);  
      obj[1] = cstring[0];  
      obj[2] = rest;  
      return obj;
```

```
v3:  rest   = marshal_from_cstring(cstring+1);  
      push(rest);  
      obj    = heap_allocate(Cons,2);  
      obj[1] = cstring[0];  
      obj[2] = pop();  
      return obj;
```

Future work

I'm not convinced that it is worth hacking on this toy compiler (as opposed to work on a full-featured “real” compiler), but in any case, here is a TODO list:

- ▶ add nested patterns, and patterns in lambda arguments
- ▶ do some optimizations, so that the generated code is better
- ▶ tail call elimination
- ▶ escaping in string constants
- ▶ compile to a low-level intermediate language first, and do codegen from that
- ▶ add a type system
- ▶ more backends: x86-64 assembly and/or machine code; some simple VM