Programming Languages and Trust

The what, why, and how (and the hacks)

Veit Heller

January 8, 2019

Datengarten | CCCB

whoami

- I work at a consultancy.
- I hack on languages in my free time.
- zepto, Carp, cspfuck...
- I'm secretly a turtle.

Compilers

How do programming languages work?

- We usually start with a messy dichotomy and juxtapose compilers and interpreters.
  - Compilers transform source code into some form of executable code.
  - Interpreters take in source code and evaluate it directly.

How do programming languages really work?

- Most “real” implementations transform their source code first in some way.
- This representation has many names: Abstract Syntax Tree (AST), Intermediate Representation (IR), Byte Code, Bit Code, et al.
- And what about transpilers? Oh my.

How do programming languages really work?

- We have some sort of pipeline:
  - Takes in source code,
  - Transforms, and
  - Spits out another representation or evaluates it directly.

A pipeline?

A pipeline?

```
void eval(char* str) {
  int tape[30000];
  int head = 0;
  for (int i = 0; i<30000; i++) tape[i] = 0;
  while(*str) {
    switch(*str) {
      case '+': tape[head]++; break;
      case '-': tape[head]--; break;
      case '>': head++; break;
      case '<': head--; break;
      case '.': printf("%c", tape[head]); break;
      case ',': scanf("%c", (char*)&tape[head]); break;
      case '[': if(!tape[head]) str = search_end_loop(str); break;
      case ']': if(tape[head]) str = search_begin_loop(str); break;
    }
    ++str;
  }
}
```

Listing 1: A silly Brainfuck VM

A pipeline?

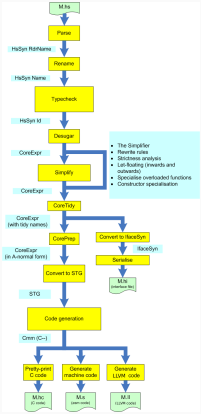

Figure 1: The GHC pipeline.

This picture was sourced from https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/HscMain.

Why pipelines?

- Huge pipelines like this might seem overkill for simple languages.
- These days, they buy us modularity, clarity, and a low barrier of entry.
- Independent passes are great! I can finally do proper testing!
- The extreme end of this spectrum is nanopass compilers (cool stuff!).

Passes Frontend—Tokenizer/Lexer/Parser

- Most language implementations have a parser.
- Tokenizer/Lexer splits stream up into tokens (Lex/Flex/ANTLR...).
- Parser generates IR/AST from source or tokens.

Passes Frontend—Tokenizer/Lexer/Parser

- Parsers are generally more interesting.
- You can generate them (YACC/Bison/ANTLR...)—but that's boring.
- You can alse write them completely manually (recursive descent is popular) or use a framework.
- Personal favorites: recursive descent, GLL (Generalized Left-to-Right, Leftmost derivation), OMeta, Parsec.
- Rule of Thumb: Don't overengineer it early on. It's easy to replace.

Passes Frontend—Typechecking

- You can have fun with types!
- A simple, but powerful model is Hindley-Milner, with sum and product types (think Haskell or OCaml).
- It buys you a "simple" type inference algorithm for functions.
- You can go wild! Dependent types, Liquid Types, and a whole bunch of things I don't understand!
- You don't need a pass for that, but even in dynamic languages sometimes this makes sense because specialization leads to better generated code.

Passes Backend—Optimizations

- Heavily language-dependent!

⇒ Example: Haskell is heavily optimized in a way that doesn't make sense elsewhere—because it is lazy, reordering is extremely important. And don't forget currying!

- It's not always higher level to lower level; more on that later.
- It's not black magic, although it often feels like it is.
- If you use LLVM you already have dozens of passes available to you.

Passes Backend—Optimizations

What does [-] mean in Brainfuck?

Passes Backend—Optimizations

It means “set the current value under the tape to zero”.

$\Rightarrow$ We can remove the loop entirely!

Demo I—enter cspfuck

Trust

Trusting Trust

- In 1984, Ken Thompson was rightfully awarded the Turing Award.
- He wrote a three-page paper on a scary idea: malicious compilers.

The idea

- Have you heard about Quines? They are self-replicating programs.
- Have you heard about bootstrapping compilers? They are compilers that can compile themselves.
- What if we compile a “buggy” version of our compiler and ship it?

The idea

"The actual bug I planted in the compiler would match code in the UNIX "login" command. The replacement code would miscompile the login command so that it would accept either the intended encrypted password or a particular known password. Thus if this code were installed in binary and the binary were used to compile the login command, I could log into that system as any user."

— Ken Thompson, Reflections on Trusting Trust, page 3

The idea

"Such blatant code would not go undetected for long. Even the most casual perusal of the source of the C compiler would raise suspicions."

— Ken Thompson, Reflections on Trusting Trust, page 3

The idea

"This [second approach] simply adds a second Trojan horse to the one that already exists. The second pattern is aimed at the C compiler. [...] First we compile the modified source with the normal C compiler to produce a bugged binary. We install this binary as the official C. We can now remove the bugs from the source of the compiler and the new binary will reinsert the bugs whenever it is compiled. Of course, the login command will remain bugged with no trace in source anywhere."

— Ken Thompson, Reflections on Trusting Trust, page 3

Takeaways

- It has historically mostly been regarded as a scary practical joke by compiler engineers.
- As compilers get more complex (and more modular), a malicious "optimization" pass can ever more easily be inserted into the compiler.
- If someone pointed at an obscure piece of assembly and told you that "this simple optimization buys us a 10% speed gain on auto-vectorized code in ARM", would you just believe them?
- Nota Bene: There is a workaround described by David A. Wheeler, but it assumes that there is no non-determinism in a compiler, but there often is (see, for instance, GCC's `-fguess-branch-probability`).

Demo II—enter Michael Arntzenius

References I

- These slides: `https://github.com/hellerve/talks`
- A talk on Nanopass Compilers by Andrew Keep:
  `https://www.youtube.com/watch?v=Os7FE3J-U5Q`
- Ken Thompson, Reflections on Trusting Trust:
  `https://www.win.tue.nl/ aeb/linux/hh/thompson/trust.html`
- Michael Arntzenius' reflections on Trusting Trust:
  `https://github.com/rntz/rotten`
- A blog post on how I sped up cspfuck:
  `https://blog.veitheller.de/Speeding_up_an_Interpreter.html`
- My favorite talks on compilers and interpreters:
  `https://github.com/hellerve/programming-talks#compilersinterpreters`
- Some of my favorite (and loathed) papers:
  `https://github.com/hellerve/ptolemy/blob/master/done.md`
- A list of LLVM passes: `http://llvm.org/docs/Passes.html`

References II

- Elizabeth Scott and Adrian Johnstone, GLL Parsing: `http://dotat.at/tmp/gll.pdf`
- Daan Leijen and Erik Meijer, Parsec: Direct Style Monadic Parser Combinators for the Real World: `https://www.microsoft.com/en-us/research/publication/parsec-direct-styl`
- Ana Bove and Peter Dybjer, Dependent Types at Work: `http://www.cse.chalmers.se/%7Epeterd/papers/DependentTypesAtWork.pdf`
- Alessandro Warth, Experimenting with Programming Languages (The original thesis on OMeta): `www.vpri.org/pdf/tr2008003_experimenting.pdf`
- Patrick Rondon et al, Liquid Types: `http://goto.ucsd.edu/~rjhala/liquid/liquid_types.pdf`
- David A. Wheeler, Fully Countering Trusting Trust through Diverse Double-Compiling (DDC) - Countering Trojan Horse attacks on Compilers: `https://dwheeler.com/trusting-trust/`

Thank you!

Questions?

Slides at https://github.com/hellerve/talks