# Calculating Correct Compilers

Cameron Wong

Bahr/Hutton 2015, Pickard/Hutton 2021

2022-11-03

# Calculating Correct Compilers

# Compilers?

- Let $S$ and $T$ be programming languages
- We define a compiler from $S$ to $T$ to be a function

$$\texttt{compile} : \textsc{Syntax}_S \to \textsc{Syntax}_T \tag{1}$$

where

  - $S$ is the source (or surface) language
  - $T$ is the target language

# Compilers?

- Usually, we also want an actual implementation of the `compile` function
- This implementation language is called the <u>host</u> language

# Compilers?

- Ignore practical details like parsing, linking, etc
- For simplicity, assume that the input is well-behaved (no type errors, etc)
- This means that the `compile` function is a total function from syntax to syntax

# Calculating Correct Compilers

# Correct?

- Languages are defined by their <u>syntax</u> and <u>semantics</u>
- If `compile` acts on syntax, its *correctness* should talk about semantics

# Correct?

- What can we say about semantics?

# Correct?

- $e$ and `compile` $e$ should "mean the same thing"
- Let's restrict even further to <u>dynamic</u> (runtime) correctness
  - $e$ and `compile` $e$ should "<u>do</u> the same thing"

# Correct?

- This is actually very difficult to state formally!

# Correct?

- What <u>parts</u> of behavior need to be preserved?
  - Performance?
  - Memory?
- $S$ and $T$ may not even use the same execution model! Think
  - $S =$ lambda calculus
  - $T =$ turing machines

## Correct?

- Even just looking at "return values" takes some machinery
- Suppose
  - $e \rightsquigarrow^* v$ and
  - `compile` $e \rightsquigarrow^* \overline{v}$
- $v$ is in $S$, but $\overline{v}$ is in $T$!
- For example, if $S$ is Java and $T$ is assembly, $v$ could be some a complex object!

# Correct?

- Idea: define a relation $R$ on values$(S) \times$ values$(T)$
- Can then define correctness as
  - If $e \rightsquigarrow^* v$ and `compile` $e \rightsquigarrow^* \overline{v}$, then $R(v, \overline{v})$
- Could generalize and define the relation over $\textsc{Syntax}_S \times \textsc{Syntax}_T$, but this is typically much harder

# Correct?

- Another way: define some common semantic domain $D$, with denotation functions
  - $[\![\cdot]\!]_S$ : programs$(S) \to D$
  - $[\![\cdot]\!]_T$ : programs$(T) \to D$
- Then correctness is stated as

$$[\![e]\!]_S = [\![\texttt{compile } e]\!]_T \tag{2}$$

This is also known as a logical relation

# Correct?

- $D$ and $[\![\cdot]\!]$ exist outside of the languages $S$ and $T$, so they can be arbitrarily complicated
- Ex
    - Let $D = \mathbb{N} \cup \{\bot\}$ such that $[\![e]\!] = \bot$ if $e$ infinite-loops when executed

# Correct?

- For us: Our approach will look like the denotation method, embedded into the host language
  - That is, $[\![\cdot]\!]_S$ and $[\![\cdot]\!]_T$ will be host language functions, rather than purely metatheoretical

# Calculating Correct Compilers

# Setup

- Define the syntax of our source language as follows:

$$\begin{array}{l}\textbf{data}\ \texttt{Expr}\ \textbf{where}\\ \quad \texttt{Val}\ :\ \mathbb{N} \to \texttt{Expr}\\ \quad \texttt{Add}\ :\ \texttt{Expr} \to \texttt{Expr} \to \texttt{Expr}\end{array}$$

(where $\mathbb{N}$ is the type of natural numbers <u>in our host language</u>)

# Setup

- Define the semantics of our source via an interpreter:

```
eval : Expr → ℕ
eval (Val x) = x
eval (Add x y) = eval x + eval y
```

# Setup

- Notice that `eval` transforms a <u>source language</u> term into a <u>host language</u> value.
- This is exactly what we want for our denotation!

# Target language

- What about the target language?

# Target language?

- We'll compile `Expr` to a yet-undefined stack-based language

```
data Code where
  -- To be determined
```

# Target language

- Since `eval` outputs a natural number, our stack should contain nats:

  $$\textbf{type}\ \texttt{Stack = [}\mathbb{N}\texttt{]}$$

- We'll also define an interpreter on these stacks:

  $$\texttt{exec : Code} \rightarrow \texttt{Stack} \rightarrow \texttt{Stack}$$

# The Plan

- Goal: Derive the definitions of `Code` and `exec` at the same time, using our (TBD) correctness theorem as a guide
- As mentioned, we'll use denotations to state the correctness principle

## Correctness

- What to choose for our semantic domain $D$?

## Correctness

- Since `eval` returns $\mathbb{N}$, we could choose $D = \mathbb{N}$
- Our denotations would be

$$
\begin{aligned}
[\![e]\!]_{\texttt{Expr}} &= \texttt{eval}\ e \\
[\![\overline{e}]\!]_{\texttt{Code}} &= \texttt{head}\ (\texttt{exec}\ \overline{e}\ \texttt{[]})
\end{aligned}
$$

- This suggests the following correctness theorem:

$$\forall e\ .\ \texttt{exec (compile}\ e\texttt{)}\ \texttt{[]} = \texttt{eval}\ e :: \texttt{[]} \tag{3}$$

- "executing this code from an empty stack gives the right answer"

# Correctness

- It turns out that this doesn't work! [1]
- Instead, we need something stronger:

$$\forall e, s \;.\; \texttt{exec (compile } e\texttt{)}\; s = \texttt{eval}\; e :: s \tag{4}$$

[1] To see why, try to directly prove this theorem for the final `compile` function we produce at the end

# Denotations?

- Aside: What is the denotation function here?

# Denotations?

- Choose $D = \texttt{Stack} \to \texttt{Stack}$:

$$\llbracket e \rrbracket_{\texttt{Expr}} = \lambda s.\texttt{eval}\ e :: s$$
$$\llbracket \overline{e} \rrbracket_{\texttt{Code}} = \lambda s.\texttt{exec}\ e\ s$$

- Define equality as extensional equality, e.g. $f_1 = f_2$ when $f_1(x) = f_2(x)$.

# The Plan, restated

- At last, all the pieces are in place.
- The plan:
  - Induct on the structure of `Expr`.
  - Use equation 4 to define `exec` and `compile`.
- Because we used equation 4 in this derivation, we get the correctness of `compile` for free!

## Some induction

Case: $e = \texttt{Val}\ x$

- Want to find $\texttt{exec}$, $c$ solving

$$\texttt{exec}\ c\ s = x :: s \tag{5}$$

# Some induction

Case: $e = \texttt{Val}\ x$

- Currently, `Code` has <u>no</u> syntax!
- So, make a new constructor `PUSH` for this case.
  - $x$ is used on the RHS but doesn't appear on the LHS, so `PUSH` must take $x$ as an argument
  - Then, define

```
compile (Var x) = PUSH x
exec (PUSH x) s = x :: s
```

## More induction

Case: $e = \texttt{Add}\ e_1\ e_2$

- Need to solve

$$\texttt{exec}\ c\ s = (\texttt{eval}\ e_1 + \texttt{eval}\ e_2) :: s \tag{6}$$

- As before, we define new syntax form `ADD`

# More induction

Case: $e = \texttt{Add}\ e_1\ e_2$

- Idea: Have `Add` take two arguments, as before

  compile (Add $e_1$ $e_2$) = ADD (eval $e_1$) (eval $e_2$)
  exec (ADD $x$ $y$) $s$ = $x + y$ :: $s$

- Problem: `compile` uses `eval`.[2].

[2]We don't actually want to rule out `eval` entirely; some compilers actually will call `eval` on fragments of the source, usually for optimization. We just want to rule out degenerate compilers like this one, which cheat by effectively only translating the return value.

# More induction

Case: $e = \texttt{Add}\ e_1\ e_2$

- Instead, pass arguments to `ADD` on the stack $s$:

  $$\texttt{exec ADD } (x :: y :: s) \texttt{ = } x + y \texttt{ :: } s$$

# More induction

Case: $e = \texttt{Add}\ e_1\ e_2$

- Now, use the inductive hypotheses to define $c$:

$$
\begin{aligned}
& \texttt{eval}\ \ e_1 + \texttt{eval}\ \ e_2 :: s \\
= \ & \langle \text{definition of } \texttt{exec} \rangle \\
& \texttt{exec}\ \ \texttt{ADD}\ (\texttt{eval}\ \ e_1 :: \texttt{eval}\ \ e_2 :: s) \\
= \ & \langle \text{inductive hypothesis, } e_1 \rangle \\
& \texttt{exec}\ \ \texttt{ADD}\ (\texttt{exec}\ (\texttt{compile}\ e_1)\ (\texttt{eval}\ \ e_2 :: s)) \\
= \ & \langle \text{inductive hypothesis, } e_2 \rangle \\
& \texttt{exec}\ \ \texttt{ADD}\ (\texttt{exec}\ (\texttt{compile}\ e_1)\ (\texttt{exec}\ (\texttt{compile}\ e_2)\ s))
\end{aligned}
$$

# More induction

Case: $e = \texttt{Add}\ e_1\ e_2$

- We're actually stuck again, so we need to define more syntax

$$\texttt{exec}\ (c_1\ \texttt{+++}\ c_2)\ s\ \texttt{=}\ \texttt{exec}\ c_2\ (\texttt{exec}\ c_1\ s)$$

- Can finalize with

$$\texttt{compile}\ (\texttt{Add}\ e_1\ e_2)\ \texttt{=}\ \texttt{compile}\ e_1\ \texttt{+++}\ e_2\ \texttt{+++}\ \texttt{ADD}$$

# Recap

```
data Code where
  PUSH : ℕ → Code
  ADD : Code
  _+++_ : Code → Code → Code

compile : Expr → Code
compile (Var x) = PUSH x
compile (Add e₁ e₂) = compile e₁ +++ e₂ +++ ADD

exec (PUSH x) s = x :: s
exec ADD (x :: y :: s) = x + y :: s
exec (c₁ +++ c₂) s = exec c₂ (exec c₁ s)
```

## Extensions: Dependent Types

- Can be used formalize our idea of "well-formed" inputs
- Can also be used to make `exec` more safe (can rule out `exec ADD []`)

# Extensions: Exceptions

- Need to use code continuations

  `compile' : Expr → Code → Code`

- Actually, this complicates our denotations:
  - The `compile'` function takes both `Expr` (source) *and* `Code` (target) arguments
  - It's not immediately obvious how to define $[\![\cdot]\!]_{\texttt{Expr}}$ in this model

# How to represent exceptions?

- Define a "chained" compiler from `Expr` to `Code`:

$$\texttt{Expr} \xrightarrow{\lambda e.(e,\texttt{HALT})} \texttt{Expr} \times \texttt{Code} \xrightarrow{\texttt{compile'}} \texttt{Code}$$

$$\texttt{compile}\ e\ \texttt{=}\ \texttt{compile'}\ e\ \texttt{HALT}$$

- Now we can <u>switch</u> denotations halfway!
- Correctness of $\lambda e.(e, \texttt{HALT})$ is obvious
- Correctness of `compile'` can be defined straightforwardly:

$$\begin{aligned} D &= \texttt{Stack} \\ [\![(e, c)]\!]_{\texttt{Expr}\times\texttt{Code}} &= \texttt{exec}\ c\ \texttt{(eval}\ e :: s\texttt{)} \\ [\![(\overline{e}, c)]\!]_{\texttt{Code}} &= \texttt{exec}\ \texttt{(comp}\ \overline{e}\ c\texttt{)}\ s \end{aligned}$$

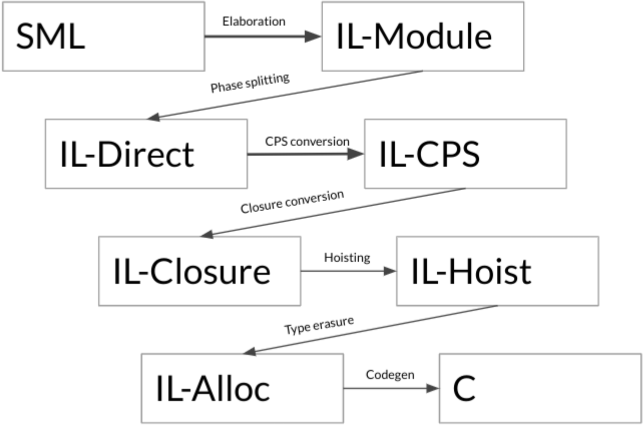
# Why?

- Why would we <u>want</u> to do this?
- What use is a compiler with an unknown target language?

# Some compiler architecture

- Compilers for real languages don't usually go directly from source to target
- Instead, they compile through a series of intermediate representations

# Some compiler architecture



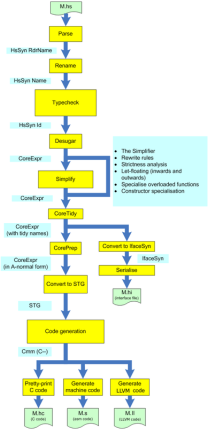

Crary 2020, CMU 15-417 Higher-Order Typed Compilation

# Why?

- Each of those "IL-" phases are full-fledged programming languages.
- If each step is individually correct, then it should[3] be correct to pipeline them together!

[3] It *is* possible to have compiler phases that are individually correct but incorrect when composed (or when composed in a bad order), but this tends to arise from incomplete language specifications

# More compiler architecture (Glasgow Haskell Compiler)




Marlow/Peyton-Jones, The Architecture of Open Source Applications

# Why?

- How were those intermediate forms designed?
- The developers constructed them ad-hoc to have certain properties, then separately reasoned (formally or informally) that the translation is correct

# Why?

- What if we could build the intermediate forms and its translation in a correct-by-construction way?

# Question break

# Final Project Concepts

- What if we could do all of that, but applied to something else?

# Final Project Concepts

- What if we could do all of that, but applied to ~~something else~~ semantics themselves?

# Final Project Concepts

- Suppose we have the typing rules for some programming language, defined by a function

$$\texttt{typeof} : \texttt{Expr} \to \mathbf{Maybe}\ \texttt{Type}$$

- The most basic notions of type safety are the twin theorems of progress and preservation:
  - Progress: Well-typed programs don't get stuck
  - Preservation: Well-typed programs remain well-typed

# Final Project Concepts

- What can equational reasoning tell us about these theorems?
- Progress can be witnessed by a function

```
data Progress : Set where
  STEP : Expr → Progress
  DONE : Progress

progress : (e : Expr) → ∃τ.(typeof e ≡ τ) → Progress
```

- If we assume all expressions are well-typed and `DONE` for now, we get

```
step : Expr → Expr
```

# Final Project Concepts

- This means we can define preservation as the equation

  $\texttt{typeof (step } e\texttt{) = typeof } e$

  (ignoring ill-typed expressions and values for brevity)

# Final Project Concepts

- Can we use this equation, and the definition of `typeof`, to construct an appropriate definition of `step`?
-

# Final Project Concepts

- Can we use this equation, and the definition of `typeof`, to construct an appropriate definition of `step`?
- (Yes, it works, and it's so trivial that it got rejected from a workshop earlier this year)

# Making it cooler

- Could we derive the `typeof` function itself? (from `step`? From the inference rules?)
- Does this generalize to more interesting systems?
- Can we generate the interpreter mechanically, instead of performing the calculation by hand?

# Idea: Gradual typing

- In a gradually-typed system, we have the "gradual guarantee"

  "Replacing a type annotation with `dyn` won't change the result"
- Can we phrase this as an equation? Something like

$$\texttt{eval}\ (\texttt{gradualize}\ e) = \texttt{eval}\ e \tag{7}$$

- Is this sufficient to derive `eval`?
  - Probably not, but maybe with some additional constraints