Compilation

Hype for Types

March 30, 2021

Outline

Why compile?

- When we write code, we want to run the code.

Why compile?

- When we write code, we want to run the code.
- We could write a simple "expression evaluator". However, our code would be very slow.

Why compile?

- When we write code, we want to run the code.
- We could write a simple "expression evaluator". However, our code would be very slow.
- Instead, we want to "translate" our (high-level) functional code to (low-level) assembly code.

Why compile?

- When we write code, we want to run the code.
- We could write a simple "expression evaluator". However, our code would be very slow.
- Instead, we want to "translate" our (high-level) functional code to (low-level) assembly code.
- Then, we can take advantage of a computer's efficient hardware!

Why compile?

- When we write code, we want to run the code.
- We could write a simple “expression evaluator”. However, our code would be very slow.
- Instead, we want to “translate” our (high-level) functional code to (low-level) assembly code.
- Then, we can take advantage of a computer’s efficient hardware!

Why compile?

- When we write code, we want to run the code.
- We could write a simple “expression evaluator”. However, our code would be very slow.
- Instead, we want to “translate” our (high-level) functional code to (low-level) assembly code.
- Then, we can take advantage of a computer’s efficient hardware!

Main Idea

A *compiler* is simply a translator from one programming language to another.

How to compile?

Rather than going straight to Assembly, we'll want to use *intermediate languages*, composing smaller compiler *phases*.

Start with user language.

[1] For more information, take 15-417 for (1-5) and 15-411 for (6-7)!

How to compile?

Rather than going straight to Assembly, we'll want to use *intermediate languages*, composing smaller compiler *phases*.

Start with user language.

1. Elaborate (sugar-free, "simplified" language)

[1] For more information, take 15-417 for (1-5) and 15-411 for (6-7)!

How to compile?

Rather than going straight to Assembly, we'll want to use *intermediate languages*, composing smaller compiler *phases*.

Start with user language.

1. Elaborate (sugar-free, "simplified" language)
2. CPS Convert (turn functions into CPS)

[1] For more information, take 15-417 for (1-5) and 15-411 for (6-7)!

How to compile?

Rather than going straight to Assembly, we'll want to use *intermediate languages*, composing smaller compiler *phases*.

Start with user language.

1. Elaborate (sugar-free, "simplified" language)
2. CPS Convert (turn functions into CPS)
3. Closure Convert (remove binding/local scope)

[1] For more information, take 15-417 for (1-5) and 15-411 for (6-7)!

How to compile?

Rather than going straight to Assembly, we'll want to use *intermediate languages*, composing smaller compiler *phases*.

Start with user language.

1. Elaborate (sugar-free, "simplified" language)
2. CPS Convert (turn functions into CPS)
3. Closure Convert (remove binding/local scope)
4. Hoist (move local functions to top-level)

[1]For more information, take 15-417 for (1-5) and 15-411 for (6-7)!

How to compile?

Rather than going straight to Assembly, we'll want to use *intermediate languages*, composing smaller compiler *phases*.

Start with user language.

1. Elaborate (sugar-free, "simplified" language)
2. CPS Convert (turn functions into CPS)
3. Closure Convert (remove binding/local scope)
4. Hoist (move local functions to top-level)
5. Alloc (explicitly use malloc and garbage collection)

[1] For more information, take 15-417 for (1-5) and 15-411 for (6-7)!

How to compile?

Rather than going straight to Assembly, we'll want to use *intermediate languages*, composing smaller compiler *phases*.

Start with user language.

1. Elaborate (sugar-free, "simplified" language)
2. CPS Convert (turn functions into CPS)
3. Closure Convert (remove binding/local scope)
4. Hoist (move local functions to top-level)
5. Alloc (explicitly use malloc and garbage collection)
6. Abstract Assembly Generation

[1] For more information, take 15-417 for (1-5) and 15-411 for (6-7)!

How to compile?

Rather than going straight to Assembly, we'll want to use *intermediate languages*, composing smaller compiler *phases*.

Start with user language.

1. Elaborate (sugar-free, "simplified" language)
2. CPS Convert (turn functions into CPS)
3. Closure Convert (remove binding/local scope)
4. Hoist (move local functions to top-level)
5. Alloc (explicitly use malloc and garbage collection)
6. Abstract Assembly Generation
7. Register Allocation

[1] For more information, take 15-417 for (1-5) and 15-411 for (6-7)!

How to compile?

Rather than going straight to Assembly, we'll want to use *intermediate languages*, composing smaller compiler *phases*.

Start with user language.

1. Elaborate (sugar-free, "simplified" language)
2. CPS Convert (turn functions into CPS)
3. Closure Convert (remove binding/local scope)
4. Hoist (move local functions to top-level)
5. Alloc (explicitly use malloc and garbage collection)
6. Abstract Assembly Generation
7. Register Allocation

[1] For more information, take 15-417 for (1-5) and 15-411 for (6-7)!

How to compile?

Rather than going straight to Assembly, we'll want to use *intermediate languages*, composing smaller compiler *phases*.

Start with user language.

1. Elaborate (sugar-free, "simplified" language)
2. CPS Convert (turn functions into CPS)
3. Closure Convert (remove binding/local scope)
4. Hoist (move local functions to top-level)
5. Alloc (explicitly use malloc and garbage collection)
6. Abstract Assembly Generation
7. Register Allocation

End with assembly.[1]

[1] For more information, take 15-417 for (1-5) and 15-411 for (6-7)!

CPS Conversion

CPS and Assembly

In assembly, there's a difference between "values" and "expressions".

```
return (2 * 3) + (4 - 5)
```

We can only call a function on two "values", and we can only store an "expression".

```
tmp1 <- mul 2 3
tmp2 <- sub 4 5
tmp3 <- add tmp1 tmp2
ret tmp3
```

This looks like CPS!

```
mul 2 3 (fn tmp1 =>
sub 4 5 (fn tmp2 =>
add tmp1 tmp2 (fn tmp3 =>
ret tmp3)))
```

We add a syntactic distinction between *values* and *expressions*.

Why CPS?

Big Idea

Expressions operate on *values* and then pass along the result

Why CPS?

Big Idea

Expressions operate on *values* and then pass along the result (to a continuation).

We claim that turning our functions into CPS is useful. Why?

Main Idea

CPS makes control flow explicit. (Chooses the order in which to evaluate each expression.)

Bonus: Save stack space! Every function is tail-recursive, so no "stack overflow". (There's no "stack"!)

Language: Direct / Cps (Types)

τ	$::=$	$\tau_1 \to \tau_2$	function	τ	$::=$	τ **cont**	continuation
	$\mid$	$\tau_1 \times \tau_2$	binary product		$\mid$	$\tau_1 \times \tau_2$	binary product
	$\mid$	**unit**	nullary product		$\mid$	**unit**	nullary product
	$\mid$	**bool**	booleans		$\mid$	**bool**	booleans

Language: Direct / Cps (Expressions)

e	$::=$	x	variable
	$\mid$	$\lambda x : \tau.\ e$	lambda function
	$\mid$	$\langle\rangle$	unit
	$\mid$	$\langle e_1, e_2\rangle$	tuple
	$\mid$	$\textbf{true}/\textbf{false}$	boolean literal
	$\mid$	$e_1\ e_2$	function app.
	$\mid$	$\textbf{fst}(e)$	first projection
	$\mid$	$\textbf{snd}(e)$	second projection
	$\mid$	$\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2$	
	$\mid$	$\textbf{print}\ e$	printing effect

v	$::=$	x	variable
	$\mid$	$\textbf{catch}(x : \tau).\ e$	continuation
	$\mid$	$\langle\rangle$	unit
	$\mid$	$\langle v_1, v_2\rangle$	tuple
	$\mid$	$\textbf{true}/\textbf{false}$	boolean literal
e	$::=$	$\textbf{throw}(v_1, v_2)$	throw
	$\mid$	$\textbf{fst}(v; x.e)$	first projection
	$\mid$	$\textbf{snd}(v; x.e)$	second projection
	$\mid$	$\textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2$	
	$\mid$	$\textbf{print}\ (v; e)$	printing effect

Type Directed Translation

Idea

In addition to typechecking our code, we'll output a translation.

Type Directed Translation

Idea

In addition to typechecking our code, we'll output a translation.

- Instead of τ type, we have $\tau \rightsquigarrow \tau'$.

Notation

Today, rather than $[v/x]e$, we'll write $e[x \mapsto v]$ for convenience.

Type Directed Translation

Idea

In addition to typechecking our code, we'll output a translation.

- Instead of τ type, we have $\tau \rightsquigarrow \tau'$.
- Instead of $\Gamma \vdash e : \tau$, we have $\Gamma \vdash e : \tau \rightsquigarrow_k e'$.

Notation

Today, rather than $[v/x]e$, we'll write $e[x \mapsto v]$ for convenience.

Type Translation

$$\frac{}{\textbf{unit} \leadsto \textbf{unit}}$$

Type Translation

$$\frac{}{\textbf{unit} \rightsquigarrow \textbf{unit}} \qquad \frac{}{\textbf{bool} \rightsquigarrow \textbf{bool}}$$

Type Translation

$$\frac{}{\textbf{unit} \leadsto \textbf{unit}} \qquad \frac{}{\textbf{bool} \leadsto \textbf{bool}}$$

$$\frac{\tau_1 \leadsto \tau_1' \qquad \tau_2 \leadsto \tau_2'}{\tau_1 \times \tau_2 \leadsto \tau_1' \times \tau_2'}$$

Type Translation

$$\frac{}{\textbf{unit} \rightsquigarrow \textbf{unit}} \qquad \frac{}{\textbf{bool} \rightsquigarrow \textbf{bool}}$$

$$\frac{\tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{\tau_1 \times \tau_2 \rightsquigarrow \tau_1' \times \tau_2'} \qquad \frac{\tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{\tau_1 \rightarrow \tau_2 \rightsquigarrow}$$

Type Translation

$$\frac{}{\textbf{unit} \leadsto \textbf{unit}} \qquad \frac{}{\textbf{bool} \leadsto \textbf{bool}}$$

$$\frac{\tau_1 \leadsto \tau_1' \qquad \tau_2 \leadsto \tau_2'}{\tau_1 \times \tau_2 \leadsto \tau_1' \times \tau_2'} \qquad \frac{\tau_1 \leadsto \tau_1' \qquad \tau_2 \leadsto \tau_2'}{\tau_1 \to \tau_2 \leadsto (\tau_1' \times \tau_2'\ \textbf{cont})\ \textbf{cont}}$$

Type Translation

$$\frac{}{\textbf{unit} \rightsquigarrow \textbf{unit}} \qquad \frac{}{\textbf{bool} \rightsquigarrow \textbf{bool}}$$

$$\frac{\tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{\tau_1 \times \tau_2 \rightsquigarrow \tau_1' \times \tau_2'} \qquad \frac{\tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{\tau_1 \to \tau_2 \rightsquigarrow (\tau_1' \times \tau_2' \ \textbf{cont})\ \textbf{cont}}$$

Curry-Howard Isomorphism

$A \Rightarrow B$ is, classically, $\neg(A \wedge \neg B)$. (Get hype!)

Expression Translation

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \rightsquigarrow_k k\ x}$$

Expression Translation

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \rightsquigarrow_k k\ x}$$

$$\frac{}{\Gamma \vdash \langle\rangle : \textbf{unit} \rightsquigarrow_k k\ \langle\rangle}$$

Expression Translation

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \rightsquigarrow_k k\ x} \qquad \frac{}{\Gamma \vdash \langle\rangle : \textbf{unit} \rightsquigarrow_k k\ \langle\rangle}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow_{k_1} e_1' \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow_{k_2} e_2'}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \rightsquigarrow_k e_1'[k_1 \mapsto r_1.\ e_2'[k_2 \mapsto r_2.\ k\ \langle r_1, r_2 \rangle]]}$$

Expression Translation

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \rightsquigarrow_k k\ x} \qquad \frac{}{\Gamma \vdash \langle\rangle : \textbf{unit} \rightsquigarrow_k k\ \langle\rangle}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow_{k_1} e_1' \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow_{k_2} e_2'}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2 \rightsquigarrow_k e_1'[k_1 \mapsto r_1.\ e_2'[k_2 \mapsto r_2.\ k\ \langle r_1, r_2\rangle]]}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \rightsquigarrow_{k_0} e'}{\Gamma \vdash \textbf{fst}(e) : \tau_1 \rightsquigarrow_k e'[k_0 \mapsto r.\ \textbf{fst}(r; r_1.\ k\ r_1)]}$$

Expression Translation

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \rightsquigarrow_k k\ x} \qquad \frac{}{\Gamma \vdash \langle\rangle : \textbf{unit} \rightsquigarrow_k k\ \langle\rangle}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow_{k_1} e_1' \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow_{k_2} e_2'}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \rightsquigarrow_k e_1'[k_1 \mapsto r_1.\ e_2'[k_2 \mapsto r_2.\ k\ \langle r_1, r_2 \rangle]]}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \rightsquigarrow_{k_0} e'}{\Gamma \vdash \textbf{fst}(e) : \tau_1 \rightsquigarrow_k e'[k_0 \mapsto r.\ \textbf{fst}(r; r_1.\ k\ r_1)]}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \rightsquigarrow_{k_0} e'}{\Gamma \vdash \textbf{snd}(e) : \tau_2 \rightsquigarrow_k e'[k_0 \mapsto r.\ \textbf{snd}(r; r_2.\ k\ r_2)]}$$

Expression Translation

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow_{k_0} e' \qquad \tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{\begin{array}{c}\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2 \rightsquigarrow_k \\ k\ (\mathbf{catch}(p : \tau_1' \times \tau_2'\ \mathbf{cont}).\ \mathbf{fst}(p; x.\mathbf{snd}(p; r.e'[k_0 \mapsto v.\ \mathbf{throw}(r, v)])))\end{array}}$$

Expression Translation

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow_{k_0} e' \qquad \tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{\begin{array}{c}\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2 \rightsquigarrow_k \\ k\ (\mathbf{catch}(p : \tau_1' \times \tau_2'\ \mathbf{cont}).\ \mathbf{fst}(p; x.\mathbf{snd}(p; r.e'[k_0 \mapsto v.\ \mathbf{throw}(r, v)])))\end{array}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \rightsquigarrow_{k_1} e_1' \qquad \Gamma \vdash e_2 : \tau_1 \rightsquigarrow_{k_2} e_2' \qquad \tau_2 \rightsquigarrow \tau_2'}{\Gamma \vdash e_1\ e_2 : \tau_2 \rightsquigarrow_k e_1'[k_1 \mapsto f.\ e_2'[k_2 \mapsto x.\ \mathbf{throw}(f, \langle x, \mathbf{catch}(r : \tau_2').\ k\ r\rangle)]]}$$

Expression Translation

$$\frac{}{\Gamma \vdash \textbf{true} : \textbf{bool} \rightsquigarrow_k k\ \textbf{true}}$$

Expression Translation

$$\frac{}{\Gamma \vdash \textbf{true} : \textbf{bool} \leadsto_k k\ \textbf{true}} \qquad \frac{}{\Gamma \vdash \textbf{false} : \textbf{bool} \leadsto_k k\ \textbf{false}}$$

Expression Translation

$$\frac{}{\Gamma \vdash \textbf{true} : \textbf{bool} \rightsquigarrow_k k\ \textbf{true}} \qquad \frac{}{\Gamma \vdash \textbf{false} : \textbf{bool} \rightsquigarrow_k k\ \textbf{false}}$$

$$\frac{\Gamma \vdash b : \textbf{bool} \rightsquigarrow_{k_b} b' \qquad \Gamma \vdash e_1 : \tau \rightsquigarrow_{k_1} e_1' \qquad \Gamma \vdash e_2 : \tau \rightsquigarrow_{k_2} e_2'}{\begin{array}{c}\Gamma \vdash \textbf{if}\ b\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \tau \rightsquigarrow_k \\ b'[k_b \mapsto r.\ \textbf{if}\ r\ \textbf{then}\ e_1'[k_1 \mapsto k]\ \textbf{else}\ e_2'[k_2 \mapsto k]]\end{array}}$$

Conclusion

There's Plenty More!

Writing compilers is a difficult, yet rewarding, enterprise.
If this lecture seems cool, we recommend you consider 15-417 and 15-411!

Summary

- Compilers are "language translators", and often compositions of smaller "language translators".
- Types guide our thinking when we implement the translations!
  - Each language is "real", complete with types and an evaluation strategy for all well-typed programs.
  - Bonus: we can do optimization at any point without worrying about special "invariants"!
  - Easier to debug, too. If output code doesn't typecheck, it's a bug.
- By thinking compositionally, we slowly transform high-level code into Assembly.