Compilation

Hype for Types

October 7, 2024

Why compile?

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

Why compile?

- When we write code, we want to run the code
- Common strategy for running the code: interpreter and compiler

Why compile?

- When we write code, we want to run the code
- Common strategy for running the code: interpreter and compiler
- 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
- Common strategy for running the code: interpreter and compiler
- 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
- Common strategy for running the code: interpreter and compiler
- 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
- Common strategy for running the code: interpreter and compiler
- 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*

How to compile?

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

Front End

1. Parsing

How to compile?

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

Front End

1. Parsing
2. Elaboration (de-sugaring)

How to compile?

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

Front End

1. Parsing
2. Elaboration (de-sugaring)
3. Typechecking (disallow malformed programs)

How to compile?

Middle/Back End

4. CPS Conversion

[1] For more information, take 15-411 (only covers 1-3, 7-10)

How to compile?

Middle/Back End

4. CPS Conversion
5. Hoisting

[1] For more information, take 15-411 (only covers 1-3, 7-10)

How to compile?

Middle/Back End

4. CPS Conversion
5. Hoisting
6. Memory Allocation

[1] For more information, take 15-411 (only covers 1-3, 7-10)

How to compile?

Middle/Back End

4. CPS Conversion
5. Hoisting
6. Memory Allocation
7. Analysis/Optimizations

[1] For more information, take 15-411 (only covers 1-3, 7-10)

How to compile?

Middle/Back End

4. CPS Conversion
5. Hoisting
6. Memory Allocation
7. Analysis/Optimizations
   - Control Flow Graphs

[1] For more information, take 15-411 (only covers 1-3, 7-10)

How to compile?

Middle/Back End

4. CPS Conversion
5. Hoisting
6. Memory Allocation
7. Analysis/Optimizations
   - Control Flow Graphs
   - Dataflow Analysis

[1] For more information, take 15-411 (only covers 1-3, 7-10)

How to compile?

Middle/Back End

4. CPS Conversion
5. Hoisting
6. Memory Allocation
7. Analysis/Optimizations
    - Control Flow Graphs
    - Dataflow Analysis
    - Often involves making a program functional (SSA form)

[1] For more information, take 15-411 (only covers 1-3, 7-10)

How to compile?

Middle/Back End

4. CPS Conversion
5. Hoisting
6. Memory Allocation
7. Analysis/Optimizations
   - Control Flow Graphs
   - Dataflow Analysis
   - Often involves making a program functional (SSA form)
8. Register Allocation

[1] For more information, take 15-411 (only covers 1-3, 7-10)

How to compile?

Middle/Back End

4. CPS Conversion
5. Hoisting
6. Memory Allocation
7. Analysis/Optimizations
   - Control Flow Graphs
   - Dataflow Analysis
   - Often involves making a program functional (SSA form)
8. Register Allocation
9. Instruction Selection (assembly)

[1] For more information, take 15-411 (only covers 1-3, 7-10)

How to compile?

Middle/Back End

4. CPS Conversion
5. Hoisting
6. Memory Allocation
7. Analysis/Optimizations
   - Control Flow Graphs
   - Dataflow Analysis
   - Often involves making a program functional (SSA form)
8. Register Allocation
9. Instruction Selection (assembly)

[1] For more information, take 15-411 (only covers 1-3, 7-10)

How to compile?

Middle/Back End

4. CPS Conversion
5. Hoisting
6. Memory Allocation
7. Analysis/Optimizations
   - Control Flow Graphs
   - Dataflow Analysis
   - Often involves making a program functional (SSA form)
8. Register Allocation
9. Instruction Selection (assembly)

[1] For more information, take 15-411 (only covers 1-3, 7-10)

Middle End

Middle End - Hoisting

4. CPS Conversion
5. **Hoisting**
6. Memory Allocation

Move local functions to top level. But what to do with local variables?

```
let outer (x : int) =
  let inner (y : int) = x + y in
  inner
```

Multiple approaches!

Middle End - Hoisting

```
let outer (x : int) : int -> int =
  let inner (y : int) = x + y
  inner
```

Straightforward solution: Partial Application + Lambda Lifting

1. Turn local variables into function variables
2. Introduce "partial application" structure for functions

```
let inner (x : int) (y : int) = x + y

let outer (x : int) = pApp (inner, x)
```

Middle End - Hoisting

```
let outer (x : int) : int -> int =
  let inner (y : int) = x + y
  inner
```

Straightforward solution: Partial Application + Lambda Lifting

1. Turn local variables into function variables
2. Introduce "partial application" structure for functions

```
let inner (x : int) (y : int) = x + y

let outer (x : int) = pApp (inner, x)
```

`pApp (pApp (inner, 5), 6)==>* inner 5 6`

Middle End - Memory Allocation

4. CPS Conversion
5. Hoisting
6. **Memory Allocation**

Create memory representations of program values:

- Primitives (ex. `int`)
- Functions (are values!)
- Datatypes

Memory Allocation - Background

Stack: primitives, small program values

Heap: larger, more complicated values (ex. non-constant constructors, closures, records)

When we store something on the heap, the memory often looks something like this:

header	payload

Memory Allocation - ADTs

Problem

How are Algebraic Datatypes in OCaml represented in memory?

Memory Allocation - ADTs

Problem

How are Algebraic Datatypes in OCaml represented in memory?

```
type t = Apple | Orange | Pear | Kiwi
```

Memory Allocation - ADTs

Problem

How are Algebraic Datatypes in OCaml represented in memory?

```
type t = Apple | Orange | Pear | Kiwi
```

Just represent each constructor as an integer!

Apple	0
Orange	1
Pear	2
Kiwi	3

Memory Allocation - ADTs

Problem

How are ADTs in OCaml *with arguments* represented in memory?

```
type t = Apple | Orange of int | Pear of string | Kiwi
```

Memory Allocation - ADTs

Problem

How are ADTs in OCaml *with arguments* represented in memory?

```
type t = Apple | Orange of int | Pear of string | Kiwi
```

The arguments could be large, so let's allocate these on the heap:

size of block	tag	payload
header		

The non-parameterized constructors will remain integers, while the parameterized constructors will be pointers to memory on the heap.

Memory Allocation - ADTs

Sidenote: in OCaml the numbering for parameterized constructors is separate from non-parameterized constructors:

Tags	
Apple	0
Orange	0
Pear	1
Kiwi	1

Memory Allocation - ADTs

Sidenote: in OCaml the numbering for parameterized constructors is separate from non-parameterized constructors:

Tags	
Apple	0
Orange	0
Pear	1
Kiwi	1

Question

Why would it make sense to have separate numberings?

Memory Allocation - ADTs

Sidenote: in OCaml the numbering for parameterized constructors is separate from non-parameterized constructors:

Tags	
Apple	0
Orange	0
Pear	1
Kiwi	1

Question

Why would it make sense to have separate numberings?

Answer: idk ask the developers (probably some optimization scheme)

Memory Allocation - Lists

```
type list = Nil | Cons of int * list
let mylist = Cons (1, Cons (2, Cons (3, Nil)))
```

Memory Allocation - Lists

```
type list = Nil | Cons of int * list
let mylist = Cons (1, Cons (2, Cons (3, Nil)))
```

Question

How would `mylist` be represented in memory?

Memory Allocation - Lists

```
type list = Nil | Cons of int * list
let mylist = Cons (1, Cons (2, Cons (3, Nil)))
```

Question

How would `mylist` be represented in memory?

A linked-list!

Memory Allocation - Lists

```
type list = Nil | Cons of int * list
let mylist = Cons (1, Cons (2, Cons (3, Nil)))
```

Question

How would `mylist` be represented in memory?

A linked-list! Although this may be inefficient, so we can "unroll" to put multiple elements at one node in the linked-list.

Memory Allocation - Lists

```
type list = Nil | Cons of int * list
let mylist = Cons (1, Cons (2, Cons (3, Nil)))
```

Question

How would `mylist` be represented in memory?

A linked-list! Although this may be inefficient, so we can "unroll" to put multiple elements at one node in the linked-list.
At a high level it looks something like this:

```
type list =
    Nil
  | One  of int
  | Two  of int * int
  | Rest of int * int * int * list
```

Memory Allocation - Closures

Question

How should we represent closures?

Memory Allocation - Closures

Question

How should we represent closures?

After lambda-lifting, all function bodies are top-level functions.

Memory Allocation - Closures

Question

How should we represent closures?

After lambda-lifting, all function bodies are top-level functions.
Function constants = function pointers
Closures = struct with function pointer & partial application arguments (or environment map)

Middle End - CPS

4. **CPS Conversion**
5. Hoisting
6. Memory Allocation

(deep breath) Buckle up

CPS Conversion

Why CPS?

CPS conversion rewrites functions to ensure every function call is a tail call

Main Idea

CPS makes control flow explicit - everything is represented as a jump to the next continuation.

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

Remember continuations?

```
signature CONT =
sig
  type 'a cont
  val letcc : ('a cont -> 'a) -> 'a
  val throw : 'a cont -> 'a -> 'b
  val catch : ('a -> void) -> 'a cont
end
```

Remember continuations?

```
signature CONT =
sig
  type 'a cont
  val letcc : ('a cont -> 'a) -> 'a
  val throw : 'a cont -> 'a -> 'b
  val catch : ('a -> void) -> 'a cont
end
```

$$\frac{\Gamma, k : \tau\ \ \texttt{cont} \vdash e : \tau}{\Gamma \vdash \texttt{letcc}\ \ k\ \ \texttt{in}\ \ e : \tau}$$

$$\frac{\Gamma \vdash k : \tau\ \ \texttt{cont} \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{throw}\ \ k\ \ e : \tau'}$$

CPS Translation

Function Translation

$\tau_1 \to \tau_2$ becomes $(\tau_1 \times (\tau_2\ \textbf{cont}))\ \textbf{cont}$

CPS Translation

Function Translation

$\tau_1 \to \tau_2$ becomes $(\tau_1 \times (\tau_2 \textbf{ cont}))\textbf{ cont}$

Logically $\tau_1 \to \tau_2$ is $\phi_1 \supset \phi_2$. Since continuation corresponds to classical logic, this is equivalent to $\neg(\phi_1 \wedge \neg\phi_2)$, which is $(\tau_1 \times (\tau_2 \textbf{ cont}))\textbf{ cont}$.

CPS Translation

Function Translation

$\tau_1 \to \tau_2$ becomes $(\tau_1 \times (\tau_2\ \textbf{cont}))\ \textbf{cont}$

Logically $\tau_1 \to \tau_2$ is $\phi_1 \supset \phi_2$. Since continuation corresponds to classical logic, this is equivalent to $\neg(\phi_1 \wedge \neg\phi_2)$, which is $(\tau_1 \times (\tau_2\ \textbf{cont}))\ \textbf{cont}$.

`val f : int -> int = fn x => add (x, x)` where

`add : int * int -> int`

CPS Translation

Function Translation

$\tau_1 \to \tau_2$ becomes $(\tau_1 \times (\tau_2$ **cont**$))$ **cont**

Logically $\tau_1 \to \tau_2$ is $\phi_1 \supset \phi_2$. Since continuation corresponds to classical logic, this is equivalent to $\neg(\phi_1 \land \neg\phi_2)$, which is $(\tau_1 \times (\tau_2$ **cont**$))$ **cont**.

```
val f : int -> int = fn x => add (x, x)
```
where
```
add : int * int -> int
```

Translates to:

```
val f = catch (fn (x, k)=> throw addCPS ((x, x), k))
```
where
```
addCPS : ((int * int)* (int cont))cont
```

CPS Translation

Function Translation

$\tau_1 \to \tau_2$ becomes $(\tau_1 \times (\tau_2\ \textbf{cont}))\ \textbf{cont}$

Logically $\tau_1 \to \tau_2$ is $\phi_1 \supset \phi_2$. Since continuation corresponds to classical logic, this is equivalent to $\neg(\phi_1 \wedge \neg\phi_2)$, which is $(\tau_1 \times (\tau_2\ \textbf{cont}))\ \textbf{cont}$.

```
val f : int -> int = fn x => add (x, x)
```
where
```
add : int * int -> int
```

Translates to:

```
val f = catch (fn (x, k)=> throw addCPS ((x, x), k))
```
where
```
addCPS : ((int * int)* (int cont))cont
```

To call `f`:

```
letcc (fn res => throw f (5, res))
```

Conclusion

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.

There's Plenty More!

Writing a compiler is very hard, but rewarding (because compilers are useful, unlike PL theory).
If this lecture seems cool, consider taking 15-411 - Compiler Design. Also take 15-417 - HOT Compilation![2]

[2] Frank is teaching it next semester! Yippee!