Compilers: The goal

What’s our goal with compilers?
• Take a high level language, turn it into a low level language
• In a *semantics preserving* way.

\[
\begin{array}{c}
[e] : \text{ML} \\
\text{Safe} \\
[e] : \text{ASM}
\end{array}
\]
What’s our goal with compilers?
- Take a high level language, turn it into a low level language
- In a *semantics preserving* way.

The *meaning* of the program.
Compilers: The goal

What’s our goal with compilers?

- Take a high level language, turn it into a low level language
- In a *semantics preserving* way.

\[
\begin{align*}
[e] & : \text{ML} \\
\text{Safe} & : \text{ASM}
\end{align*}
\]

The *meaning* of the program.

Program in ASM corresponding to \( e \).
Compilers: The goal

What’s our goal with compilers?
• Take a high level language, turn it into a low level language
• In a *semantics preserving* way.

The meaning of the program.
Program in ASM corresponding $e$.

Some programs in ASM don’t make sense in ML.
Why Compilers?

Why do we need compilers?

• May have an interpreter for the language.
• But assembly runs faster.
• Doesn’t necessarily need to compile to assembly.
  – Can compile to C.
• In fact, each transformation is a compiler:
  – Between intermediate forms.
  – Easier to manipulate.
• In this lecture, we consider intermediate languages for functional languages.
  – Such as Continuation Passing Style (CPS).
  – Which makes control explicit.
Compilers: like onions

Types, safety

OCaml

Core ML

CPS

ASM

Intermediate code

Danger

Speed (typically)
Compilers for *functional* languages

What’s hard about compiling functional languages?

• We’re used to assembly language which is procedural
• We can’t pass around functions.

• Or can we?
  – Function pointers?

Main idea

To compile an OCaml program, find an equivalent C (like) program.
Compilers for *functional* languages

What’s hard about compiling functional languages?

- We’re used to assembly language which is procedural
- We can’t pass around functions.

• Or can we?
  – Function pointers?

Main idea
To compile an OCaml program, find an equivalent C (like) program.
Functional languages have closures

```ocaml
# let call_me_maybe x = 
  let i = ref x in 
  let inc x = (i := !i + x); !i in inc ;;
val call_me_maybe : int -> int -> int = <fun>

# let my_f = call_me_maybe 42 ;;
val my_f : int -> int = <fun>

# my_f 12 ;;
- : int = 54
```

Here, inc can touch i (not just its parameters!)

**Code** + **Data**
What does OCaml have that C doesn’t?

We take OCaml as our functional language, and C as our procedural language. C readily translates into assembly.

- HOFs
- Closures (code with data)
- Garbage Collection

Need to translate all these things into our low level language!
What does OCaml have that C doesn’t?

We take OCaml as our functional language, and C as our procedural language. C readily translates into assembly.

- HOFs
- Closures (code with data)
- Garbage Collection

HOFs as we see them in FP.

Not going to discuss.

Need to translate all these things into our low level language!
What does OCaml have that C doesn’t?

We take OCaml as our functional language, and C as our procedural language. C readily translates into assembly.  

• HOFs  
• Closures (code with data)  
• Garbage Collection

HOFs as we see them in FP. Not going to discuss.

Need to translate all these things into our low level language!
Intermediate stages for functional compilers

To accommodate HOFs as return values, a slightly different set of intermediate forms is used:

• OCaml
• Core ML
  – Simple desugaring to $\lambda$-ish calculs
• Continuation passing style (CPS)
  – Transformation from $\lambda$ to CPS
• Optimization of CPS
  – Clever tricks, $\beta$ reduction, for example.
• Closure conversion
  – Make all functions “top level:” take nested functions and pull them to the top (as in C).
• Register allocation, assembly language, etc...
Where to begin?

- OCaml
- Core ML
- CNF
- ASM
Starting point: Core ML

We start with a language called Core ML:

- Most of OCaml compiles into it simply enough.
- Very few operators.
- Maintains real semantic content.

Cosmetic transformations:
- Desugar control (if/else)
  - Lambdas
  - Pattern matching
  - Exceptions
  - etc...
Introduce “helper” functions

```plaintext
let x = 23
in if (x < 0)
  then -x
  else x

\( (\lambda x. \text{fif } (x < 0) \(-x\) x) \) 23

\text{fif } b \ e_1 \ e_2 \text{ is a primitive function which, if } b \text{ computes to true, computes } e_1, \text{ and if not, computes } e_2.
```
Desugaring \texttt{let} as \(\lambda\)

If you stare at it properly, you can see that \texttt{let} is simply \(\lambda\) placed at the appropriate place!

\[
\texttt{let } x_1 = e_1 \texttt{ in } \texttt{let } x_2 = e_2 \texttt{ in } \ldots \texttt{ in } e
\]

\[
\rightarrow
\]

\[
(\lambda x_1 (\lambda x_2 \ldots (e) e_n \ldots ) e_1)
\]
Pattern matching

Want to compile pattern matching to set of more primitive operations.

- Core idea: remember the order of matches is important.
- Reason based on decision trees.
- Compile to a chain of IFs.

\[
\text{match } a \text{ with}
\begin{align*}
  & (\text{false}, \text{nil}) \rightarrow \text{nil} \\
  & (\text{true}, \text{w}) \rightarrow \text{w} \\
  & (\text{false}, \text{x::nil}) \rightarrow \text{x::x::nil} \\
  & (\text{false}, \text{y::z}) \rightarrow \text{z}
\end{align*}
\]

\[
\text{let } (a_1, a_2) = a \text{ in}
\begin{align*}
  & \text{if } (a_1 = \text{false } \&\& a_2 = \text{nil}) \text{ then nil} \\
  & \text{else if } (a_1 = \text{true}) \text{ then } a_2 \\
  & \text{else }
\end{align*}
\]

...
Pattern matching

Want to compile pattern matching to set of more primitive operations.

- Core idea: remember the order of matches is important.
- Reason based on decision trees.
- Compile to a chain of IFs.

This is only one way to do it. There exist optimized methods to produce minimal decision trees.

```plaintext
match a with
  | (false, nil) → nil
  | (true, w) → w
  | (false, x::nil) → x::x::nil
  | (false, y::z) → z
```

```plaintext
let (a_1, a_2) = a in
  if (a_1 = false && a_2 = nil) then
    nil
  else
    else if (a_1 = true) then
      a_2
    else
      ...
```
Pattern matching

Want to compile pattern matching to set of more primitive operations.

- Core idea: remember the order of matches is important.
- Reason based on decision trees.
- Compile to a chain of IFs.

```
let (a₁, a₂) = a in
if (a₁ = false && a₂ = nil) then
    nil
else
    else if (a₁ = true) then
        a₂
    else
        ... 
```

This is only one way to do it. There exist optimized methods to produce minimal decision trees.
type name = Name of string | Int of int
type constant = { name : name; constr : bool;
    arity : int }
type var = string
type expr =
    | Var of var
    | Const of constant
    | Fun of var * expr
    | App of expr * expr
Current state of things

- OCaml
- Core ML
- CNF
- ASM
Continuations: where to go next

The next stage in the compiler uses continuations. It’s best to show the idea of continuations by example...

\[
\text{let } \text{add_one_subtract_two } x = \\
(\lambda k \ x. \ k \ (x + 1)) \\
(\lambda y. \ (y - 2))
\]

Continuations continue the computation. Once you compute a piece of the result, you invoke the continuation, and it carries it the rest of the way.
In Continuation Passing Style (CPS):

- every function takes an extra continuation argument.
- When functions compute their result, they hand it to the continuation argument.
- Corrolary: No function ever returns to its caller!
  – We aren’t building up stack frames!

```python
let pyth x y k =
(\*^k x x (\ x_2 .
  \ y y (\ y_2 .
    \ x_2 y_2 (\ x_2^2 + \ y_2^2 (\ sqrt^k x_2^2 + y_2^2 k))))))
```

A nice consequence: because functions never return, instead of CALLing them, we can simply JUMP to them. In a way, writing in CPS ’linearizes’ our program.
Another example...

```plaintext
let factorial n =
  if n=0 then 1 else n * factorial (n - 1)

CPS(·)

let factorial n k =
  ∑ n 0 (λ b.
    if b
      then (k 1)
      else (¬ k n 1
          (λ nm1.
            (factorial nm1
              (λ f (∗ k n f k))))))

This continuation keeps growing larger and larger every time!
```
Contrast with tail recursive version...

\[
\textbf{let factorial n =} \\
\quad f\_helper n 1
\]

\[
\textbf{let f\_helper n a =} \\
\quad \textbf{if} \ n = 0 \ \textbf{then} \ a \ \textbf{else} \\
\quad f\_helper (n-1) (n*a)
\]

\[
\textbf{CPS(·)}
\]

\[
\textbf{let factorial n k =} f\_helper n 1 k
\]

\[
\textbf{let f\_helper n a k =} \\
\quad \equiv_k n 0 \\
\quad (\lambda b. \\
\quad \quad \textbf{if} \ b \ (k \ a) \ \textbf{else} \\
\quad \quad \quad (-^k n 1 \\
\quad \quad \quad \quad (\lambda \ nm1. \\
\quad \quad \quad \quad \quad (*^k n a \\
\quad \quad \quad \quad \quad \quad (\lambda nta. \\
\quad \quad \quad \quad \quad \quad \quad (f\_aux nm1 nta k)))))
\]
Kicking off the computation

Now that we have this lingering continuation hanging everywhere, how do we *start* the computation?

\[
\text{let } \text{pyth } x \ y \ k = \\
(\ast^k x \ x \ (\lambda x_2. \\
(\ast^k y \ y \ (\lambda y_2. \\
(\ast^k y_2 \ (\lambda x_2 \ y_2 \ (\lambda x_2 \ y_2 \ (\lambda \ x_2 \ y_2 \ (\lambda \ x_2 \ y_2 \ (\sqrt{k} \ x_2 \ y_2 \ k)))))))))
\]

pyth 1 2 (???)

How about: \(\lambda x. x\) or \(\lambda x. \text{"print } x\) ?

In the general case, this corresponds to where you ”stop” the computation. (*Delimited* continuations let you play with this idea more.)
Transforming to CPS

We can translate all of the constructs in Core ML into CPS using a simple transform:

**The CPS Transform**

Flip each computation point ”inside out” so that we can explicitly see where to go next.

\[
\text{let square} = \\
\lambda x k . \\
*^{k'} x x k
\]

We attempt to linearize the computation, so that we can take the atomic pieces, build up results, and then pass them off to the rest of the computation (the continuation)
\[ \text{CPS}(\cdot) \]

\begin{align*}
\text{CPS}[c] & = \lambda k.(k\ c) \\
\text{CPS}[x] & = \lambda k.(k\ x) \\
\text{CPS}[\lambda x.e] & = \lambda k\ x.\text{CPS}(e) \\
\text{CPS}[e\ e_1] & = \lambda k.\text{CPS}[e_1](\lambda k_1.\text{CPS}[e])
\end{align*}
Example of using the $\text{CPS}(\cdot)$ transform

$\text{inc} (2 + 3)$

App\left(\text{Const}\left(\text{	extquotesingle\textquotesingle} inc\text{	extquotesingle\textquotesingle}\right), \right.

\text{App}\left(\right.

\text{App}\left(\text{Const}\left(\text{	extquotesingle\textquotesingle} +\text{	extquotesingle\textquotesingle}\right), \text{Const}(2), \right.

3\right)\right)

\left.\right)$

$\lambda k. k 3$

$\left(\lambda k x. k (+ 2 x)\right)$

$\left(\lambda k y. k (\text{inc} y)\right)$
Generating low level code from CPS

OCaml → Core ML → CNF → ASM
Generating low level code from CPS

We now have code in CPS, but how can we generate low level code from it?

• First idea, interpret APPLYs as jumps.
Closure Conversion (Lambda Lifting)

- C doesn’t have closures.
- If we want to compile our CPS to a format with only top level functions, we need (the equivalent of) closures.
- Basic idea: look at the function, assume an extra "environment" table with which you can look up variables over which the function closes.
Conclusion

• Compilers for functional languages work similarly in spirit to those for procedural languages.
• In both cases we slowly ’’work down’’ to some lower level implementation while preserving the semantics.
• But most real machines don’t have features amenable to the implementation of functional languages.
• To solve this problem, we apply various tricks linearize the code and make it look like C