Continuations

We’ve seen continuations several times in this course already:

- As a way to implement break and continue
- As a way to make definitional translation more robust
- As an intermediate language in interpreters
Continuations

We’ve seen continuations several times in this course already:

- As a way to implement break and continue
- As a way to make definitional translation more robust
- As an intermediate language in interpreters

Now, we’ll use them to translate a functional language down to an assembly-like language.

The translation works as a recipe for compiling any of the features we have discussed over the past few weeks all the way down to hardware.
Roadmap

CS 4120 in one lecture!
Roadmap

CS 4120 in one lecture!

**Source Language**

$\lambda$-calculus with pairs and integers
Roadmap

CS 4120 in one lecture!

Source Language
\( \lambda \)-calculus with pairs and integers

Intermediate Language #1
\( \lambda \)-calculus in CPS
Roadmap

CS 4120 in one lecture!

Source Language
\(\lambda\)-calculus with pairs and integers

Intermediate Language #1
\(\lambda\)-calculus in CPS

Intermediate Language #2
\(\lambda\)-calculus in CPS + Closure Conversion
Roadmap

CS 4120 in one lecture!

Source Language
\(\lambda\)-calculus with pairs and integers

Intermediate Language #1
\(\lambda\)-calculus in CPS

Intermediate Language #2
\(\lambda\)-calculus in CPS + Closure Conversion

Machine Code
Simple RISC-like Assembly
Source Language

We’ll start from (untyped) \( \lambda \)-calculus with pairs and integers.

\[
e ::= x \\
| \quad \lambda x. e \\
| \quad e_1 e_2 \\
| \quad (e_1, e_2) \\
| \quad #i e \\
| \quad n \\
| \quad e_1 + e_2
\]
A program $p$ consists of a series of basic blocks $bb$. 
A basic block has a label $lb$ and a sequence of commands $c$, ending with “jump.”
Target Language

\[ p ::= bb_1; bb_2; \ldots; bb_n \]
\[ bb ::= lb : c_1; c_2; \ldots; c_n; \text{jump } x \]
\[ c ::= \text{mov } x_1, x_2 \]

Commands correspond to assembly language instructions and are largely self-evident.
Target Language

\[ p ::= bb_1; bb_2; \ldots; bb_n \]

\[ bb ::= lb : c_1; c_2; \ldots; c_n; \text{jump } x \]

\[ c ::= \text{mov } x_1, x_2 \]
\[ \quad \mid \text{mov } x, n \]

Commands correspond to assembly language instructions and are largely self-evident.
Target Language

\[ p ::= bb_1; bb_2; \ldots ; bb_n \]
\[ bb ::= lb : c_1; c_2; \ldots ; c_n ; \text{jump } x \]
\[ c ::= \text{mov } x_1, x_2 \]
\[ \quad | \quad \text{mov } x, n \]
\[ \quad | \quad \text{mov } x, lb \]

Commands correspond to assembly language instructions and are largely self-evident.
Target Language

\[
p ::= \ bb_1; \ bb_2; \ldots; \ bb_n
\]
\[
\ bb ::= \lb : \ c_1; \ c_2; \ldots; \ c_n; \ \text{jump} \ x
\]
\[
c ::= \\text{mov} \ x_1, x_2
\]
\[
| \ \text{mov} \ x, n
\]
\[
| \ \text{mov} \ x, \ lb
\]
\[
| \ \text{add} \ x_1, x_2, x_3
\]

Commands correspond to assembly language instructions and are largely self-evident.
Target Language

\[ p ::= bb_1; bb_2; \ldots ; bb_n \]
\[ bb ::= lb : c_1; c_2; \ldots ; c_n; \text{jump } x \]
\[ c ::= \text{mov } x_1, x_2 \]
\[ \quad | \quad \text{mov } x, n \]
\[ \quad | \quad \text{mov } x, lb \]
\[ \quad | \quad \text{add } x_1, x_2, x_3 \]
\[ \quad | \quad \text{load } x_1, x_2[n] \]

Commands correspond to assembly language instructions and are largely self-evident.
Target Language

\[ p ::= \ bb_1; \ bb_2; \ldots; \ bb_n \]
\[ bb ::= \ lb : c_1; c_2; \ldots; c_n; \text{jump } x \]
\[ c ::= \text{mov } x_1, x_2 \]
\[ \quad | \text{mov } x, n \]
\[ \quad | \text{mov } x, lb \]
\[ \quad | \text{add } x_1, x_2, x_3 \]
\[ \quad | \text{load } x_1, x_2[n] \]
\[ \quad | \text{store } x_1, x_2[n] \]

Commands correspond to assembly language instructions and are largely self-evident.
The only un-RISC-y command is malloc. It allocates $n$ words of space and places its address into a special register $r_0$. Ignoring garbage, it can be implemented as simply as “add $r_0, r_0, -n$.”
Intermediate Language

c ::= let x = e in c

| v_1 v_2 v_3
| v_1 v_2

Commands c look like basic blocks.
Intermediate Language

\[
\begin{align*}
  c & ::= \text{let } x = e \text{ in } c \\
  & \quad | \quad v_1 \; v_2 \; v_3 \\
  & \quad | \quad v_1 \; v_2 \\
  e & ::= v \mid v_1 + v_2 \mid (v_1, v_2) \mid (#i \; v)
\end{align*}
\]

There are no subexpressions in the language!
Abstractions encoding continuations are marked with an underline. These are called *administrative lambdas* and can be eliminated at compile time.
CPS Translation

The contract of the translation is that $[e]k$ will evaluate $e$ and pass its result to the continuation $k$.

To translate an entire program, we use $k = \text{halt}$, where \text{halt} is the continuation to send the result of the entire program to.
\[ [x] k = kx \]
CPS Translation

\[
[x] \ k \ = \ k \ x \\
[n] \ k \ = \ k \ n
\]
CPS Translation

\[ [x] k = k x \]
\[ [n] k = k n \]
\[ [(e_1 + e_2)] k = [e_1] (\lambda x_1. [e_2] (\lambda x_2. \text{let } z = x_1 + x_2 \text{ in } k z)) \]
CPS Translation

\[
[x] k = k x
\]
\[
[n] k = k n
\]
\[
[(e_1 + e_2)] k = [e_1](\lambda x_1. [e_2](\lambda x_2. \text{let } z = x_1 + x_2 \text{ in } k z))
\]
\[
[(e_1, e_2)] k = [e_1](\lambda x_1. [e_2](\lambda x_2. \text{let } t = (x_1, x_2) \text{ in } k t))
\]
\[
\begin{align*}
[x] k &= k x \\
nk &= kn \\
[(e_1 + e_2)] k &= [e_1] (\forall x_1. [e_2] (\forall x_2. \text{let } z = x_1 + x_2 \text{ in } k z)) \\
[(e_1, e_2)] k &= [e_1] (\forall x_1. [e_2] (\forall x_2. \text{let } t = (x_1, x_2) \text{ in } k t)) \\
[#i e] k &= [e] (\forall t. \text{let } y = #i t \text{ in } k y)
\end{align*}
\]
CPS Translation

\[
\begin{align*}
[x] k &= k x \\
[n] k &= k n \\
[(e_1 + e_2)] k &= [e_1] (\lambda x_1. [e_2] (\lambda x_2. \text{let } z = x_1 + x_2 \text{ in } k z)) \\
[(e_1, e_2)] k &= [e_1] (\lambda x_1. [e_2] (\lambda x_2. \text{let } t = (x_1, x_2) \text{ in } k t)) \\
[#i e] k &= [e] (\lambda t. \text{let } y = #i t \text{ in } k y) \\
[\lambda x. e] k &= k (\lambda x. \lambda k'. [e] k')
\end{align*}
\]
CPS Translation

\[
[x] \ k = \ k \ x \\
[n] \ k = \ k \ n \\
[(e_1 + e_2)] \ k = \ [e_1] (\lambda x_1. [e_2] (\lambda x_2. \text{let } z = x_1 + x_2 \text{ in } k \ z)) \\
[(e_1, e_2)] \ k = \ [e_1] (\lambda x_1. [e_2] (\lambda x_2. \text{let } t = (x_1, x_2) \text{ in } k \ t)) \\
[#i e] \ k = \ [e] (\lambda t. \text{let } y = #i t \text{ in } k \ y) \\
[\lambda x. e] \ k = \ k (\lambda x. \lambda k'. [e] k') \\
[e_1 e_2] \ k = \ [e_1] (\lambda f. [e_2] (\lambda v. f \ v \ k))
\]
Example

Let’s translate the expression \([(\lambda a.\#1 a) (3, 4)] k\), using \(k = \text{halt}\).
Example

Let’s translate the expression \([((\lambda a. \#1 a) (3, 4)) k]\), using \(k = \text{halt}\).

\([((\lambda a. \#1 a) (3, 4)) k]\)
Example

Let’s translate the expression $\llbracket (\lambda a. \#1 a) (3, 4) \rrbracket k$, using $k = \text{halt}$.

$$\llbracket (\lambda a. \#1 a) (3, 4) \rrbracket k$$

$$\quad = \llbracket \lambda a. \#1 a \rrbracket (\lambda f. \llbracket (3, 4) \rrbracket (\lambda v. f v k))$$
Example

Let’s translate the expression $\llbracket (\lambda a. \#1 a) (3, 4) \rrbracket k$, using $k = \text{halt}$.

\[
\begin{align*}
    \llbracket (\lambda a. \#1 a) (3, 4) \rrbracket k \\
    = \llbracket \lambda a. \#1 a \rrbracket (\lambda f. \llbracket (3, 4) \rrbracket (\lambda v. f v k)) \\
    = (\lambda f. \llbracket (3, 4) \rrbracket (\lambda v. f v k)) (\lambda a. \lambda k'. \llbracket \#1 a \rrbracket k')
\end{align*}
\]
Example

Let’s translate the expression $\llbracket (\lambda a. \#1 a) (3, 4) \rrbracket k$, using $k = \text{halt}$.

\[
\begin{align*}
\llbracket (\lambda a. \#1 a) (3, 4) \rrbracket k \\
= \llbracket \lambda a. \#1 a \rrbracket (\lambda f. \llbracket (3, 4) \rrbracket (\lambda v. f v k)) \\
= (\lambda f. \llbracket (3, 4) \rrbracket (\lambda v. f v k)) (\lambda a. \lambda k'. \llbracket \#1 a \rrbracket k') \\
= (\lambda f. \llbracket 3 \rrbracket (\lambda x_1. \llbracket 4 \rrbracket (\lambda x_2. \text{let } b = (x_1, x_2) \text{ in } (\lambda v. f v k) b)) (\lambda a. \lambda k'. \llbracket \#1 a \rrbracket k')
\end{align*}
\]
Example

Let’s translate the expression $\llbracket (\lambda a. \#1 a) (3, 4) \rrbracket k$, using $k = \text{halt}$.

\[
\llbracket (\lambda a. \#1 a) (3, 4) \rrbracket k
= \llbracket \lambda a. \#1 a \rrbracket (\lambda f. \llbracket (3, 4) \rrbracket (\lambda v. f v k))
= (\lambda f. \llbracket (3, 4) \rrbracket (\lambda v. f v k)) (\lambda a. \lambda k'. \llbracket \#1 a \rrbracket k')
= (\lambda f. [3] (\lambda x_1. [4] (\lambda x_2. \text{let } b = (x_1, x_2) \text{ in } (\lambda v. f v k) b))
\quad (\lambda a. \lambda k'. \llbracket \#1 a \rrbracket k')
= (\lambda f. (\lambda x_1. (\lambda x_2. \text{let } b = (x_1, x_2) \text{ in } (\lambda v. f v k) b) 4) 3)
\quad (\lambda a. \lambda k'. \llbracket \#1 a \rrbracket k')
\]
Example

Let’s translate the expression \( [(\lambda a. \#1 a) (3, 4)] k \), using \( k = \text{halt} \).

\[
[(\lambda a. \#1 a) (3, 4)] k \\
= [\lambda a. \#1 a] (\lambda f. [(3, 4)] (\lambda v. f \, v \, k)) \\
= (\lambda f. [(3, 4)] (\lambda v. f \, v \, k)) (\lambda a. \lambda k'. [\#1 a] k') \\
= (\lambda f. [3] (\lambda x_1. [4] (\lambda x_2. \text{let } b = (x_1, x_2) \text{ in } (\lambda v. f \, v \, k) \, b)) (\lambda a. \lambda k'. [\#1 a] k') \\
= (\lambda f. (\lambda x_1. (\lambda x_2. \text{let } b = (x_1, x_2) \text{ in } (\lambda v. f \, v \, k) \, b) \, 4) \, 3) (\lambda a. \lambda k'. [\#1 a] k') \\
= (\lambda f. (\lambda x_1. (\lambda x_2. \text{let } b = (x_1, x_2) \text{ in } (\lambda v. f \, v \, k) \, b) \, 4) \, 3) (\lambda a. \lambda k'. [a](\lambda t. \text{let } y = \#1 t \text{ in } k' \, y))
\]
Optimization

Clearly, the translation generates a lot of administrative $\lambda$s!
Optimization

Clearly, the translation generates a lot of administrative $\lambda$s!

To make the code more efficient and compact, we will optimize using some simple rewriting rules to eliminate administrative $\lambda$s
Clearly, the translation generates a lot of administrative λs!

To make the code more efficient and compact, we will optimize using some simple rewriting rules to eliminate administrative λs.

We can eliminate applications to variables by copy propagation:

\[(\lambda x. e) y \rightarrow e[y/x]\]
Optimization

Clearly, the translation generates a lot of administrative λs!

To make the code more efficient and compact, we will optimize using some simple rewriting rules to eliminate administrative λs

We can eliminate applications to variables by copy propagation:

\[(\lambda x. e) y \rightarrow e[y/x]\]

Other lambdas can be converted into lets:

\[(\lambda x. c) v \rightarrow \text{let } x = v \text{ in } c\]
Optimization

Clearly, the translation generates a lot of administrative $\lambda$s!

To make the code more efficient and compact, we will optimize using some simple rewriting rules to eliminate administrative $\lambda$s:

We can eliminate applications to variables by copy propagation:

$$\left(\lambda x.e\right) y \rightarrow e\{y/x\}$$

Other lambdas can be converted into lets:

$$\left(\lambda x.c\right) v \rightarrow \text{let } x = v \text{ in } c$$

We can also perform administrative $\eta$-reductions:

$$\lambda x. k x \rightarrow k$$
Example, Redux

After applying these rewrite rules to the expression we had previously, we obtain:

\[
\begin{align*}
let f &= \lambda a. \lambda k'. \text{let } y = \#1 a \text{ in } k'y \text{ in} \\
let x_1 &= 3 \text{ in} \\
let x_2 &= 4 \text{ in} \\
let b &= (x_1, x_2) \text{ in} \\
f b k
\end{align*}
\]

This is starting to look a lot more like our target language!
Optimization

Writing these optimizations separately makes it easier to define the CPS conversion uniformly, without worrying about efficiency.
Optimization

Writing these optimizations separately makes it easier to define the CPS conversion uniformly, without worrying about efficiency.

We may not be able to remove all administrative lambdas. Any that cannot be eliminated using the rules above are converted into “real” lambdas.
Roadmap

Source Language
\( \lambda \)-calculus with pairs and integers

Intermediate Language #1
\( \lambda \)-calculus in CPS

Intermediate Language #2
\( \lambda \)-calculus in CPS + Closure Conversion
Closure Conversion

The next step is to bring all $\lambda$s to the top level, with no nesting.

\[
P ::= \begin{cases} 
    \text{let } x_f = \lambda x_1. \ldots \lambda x_n. \lambda k. c \text{ in } P \\
    \text{let } x_c = \lambda x_1. \ldots \lambda x_n. c \text{ in } P \\
    c 
\end{cases}
\]

\[
c ::= \begin{cases} 
    \text{let } x = e \text{ in } c | x_1 x_2 \ldots x_n 
\end{cases}
\]

\[
e ::= n | x | \text{halt} | x_1 + x_2 | (x_1, x_2) | \#i x 
\]

This translation requires the construction of closures that capture the free variables of the lambda abstractions and is known as closure conversion.
Closure Conversion

The main part of the translation is:

\[
\left[ \lambda x. \lambda k. \ c \right] \sigma =
\]

let \((c', \sigma') = \left[ c \right] \sigma\) in

let \(y_1, \ldots, y_n = fvs(\lambda x. \lambda k. \ c')\) in

\((f \ y_1 \ \ldots \ y_n, \ \sigma'[f \mapsto \lambda y_1. \ldots \lambda y_n. \ \lambda x. \ \lambda k. \ c'])\) where \(f\) fresh
The main part of the translation is:

$$\llbracket \lambda x. \lambda k. \ c \rrbracket \sigma =$$

let \((c', \sigma') = \llbracket c \rrbracket \sigma\) in

let \(y_1, \ldots, y_n = fvs(\lambda x. \lambda k. \ c')\) in

\((f \ y_1 \ \ldots \ y_n, \ \sigma' [f \mapsto \lambda y_1. \ \ldots \ \lambda y_n. \ \lambda x. \ \lambda k. \ c'])\) where \(f\) fresh

The translation of \(\lambda x. \lambda k. \ c\) above first translates the body \(c\), then creates a new function \(f\) parameterized on \(x\) as well as the free variables \(y_1\) to \(y_n\) of the translated body.
Closure Conversion

The main part of the translation is:

\[
\begin{align*}
[\lambda x. \lambda k. \ c] \ \sigma &= \\
&\text{let } (c', \sigma') = [c] \ \sigma \text{ in} \\
&\text{let } y_1, \ldots, y_n = \text{fvs}(\lambda x. \lambda k. \ c') \text{ in} \\
&(f \ y_1 \ldots \ y_n, \ \sigma'[f \mapsto \lambda y_1. \ldots \lambda y_n. \lambda x. \lambda k. \ c']) \text{ where } f \ \text{fresh}
\end{align*}
\]

The translation of \( \lambda x. \lambda k. \ c \) above first translates the body \( c \), then creates a new function \( f \) parameterized on \( x \) as well as the free variables \( y_1 \) to \( y_n \) of the translated body.

It then adds \( f \) to the environment \( \sigma \) replaces the entire lambda with \((f \ y_n \ldots \ y_n)\).
Closure Conversion

The main part of the translation is:

\[
\begin{align*}
\llbracket \lambda x. \lambda k. \ c \rrbracket \ \sigma &= \\
&\text{let } (c', \sigma') = \llbracket c \rrbracket \ \sigma \text{ in} \\
&\text{let } y_1, \ldots, y_n = \text{fvs}(\lambda x. \lambda k. \ c') \text{ in} \\
&(f y_1 \ldots y_n, \ \sigma'[f \mapsto \lambda y_1. \ldots \lambda y_n. \ \lambda x. \ \lambda k. \ c']) \text{ where } f \text{ fresh}
\end{align*}
\]

The translation of \( \lambda x. \lambda k. \ c \) above first translates the body \( c \), then creates a new function \( f \) parameterized on \( x \) as well as the free variables \( y_1 \) to \( y_n \) of the translated body.

It then adds \( f \) to the environment \( \sigma \) replaces the entire lambda with \((f y_n \ldots y_n)\).

When applied to an entire program, this has the effect of eliminating all nested \( \lambda \)s.
**Roadmap**

**Source Language**
\( \lambda \)-calculus with pairs and integers

**Intermediate Language #1**
\( \lambda \)-calculus in CPS

**Intermediate Language #2**
\( \lambda \)-calculus in CPS + Closure Conversion

**Machine Code**
Simple RISC-like Assembly
\[ P[c] = \text{main}: C[c]; \]

\[ \text{halt} : \]
\( \mathcal{P}[\text{let } x_f = \lambda x_1. \ldots \lambda x_n. \lambda k. \ c \ \text{in } p] = \ x_f : \text{mov } x_1, a_1; \\
\quad \vdots \\
\quad \text{mov } x_n, a_n; \\
\text{mov } k, ra; \\
C[c]; \\
\mathcal{P}[p] \)
\[
\mathcal{P}[\text{let } x_c = \lambda x_1. \ldots \lambda x_n. c \text{ in } p] = x_c : \text{mov } x_1, a_1;
\]
\[
\vdots
\]
\[
\text{mov } x_n, a_n;
\]
\[
C[c];
\]
\[
\mathcal{P}[p]
\]
\[ C[\text{let } x = n \text{ in } c] = \text{mov } x, n; \]

\[ C[c] \]
$C[\text{let } x_1 = x_2 \text{ in } c] = \text{mov } x_1, x_2;$

$C[c]$
Code Generation

\[ C[\text{let } x = x_1 + x_2 \text{ in } c] = \text{add } x_1, x_2, x; \]  
\[ C[c] \]
Code Generation

\[ C\left[\text{let } x = (x_1, x_2) \text{ in } c\right] = \text{malloc 2;} \]
\[ \text{mov } x, r_0; \]
\[ \text{store } x_1, x[0]; \]
\[ \text{store } x_2, x[1]; \]
\[ C[c] \]
$C[\text{let } x = \#i x_1 \text{ in } c] = \text{load } x, x_1[i-1];$ 

$C[c]$
$$C[\sigma_1^T x_1 \ldots x_n] = \text{mov } a_1, x_1;$$

$$\vdots$$

$$\text{mov } a_n, x_n;$$

$$\text{mov } ra, k;$$

$$\text{jump } x$$
Final Thoughts

Note that we assume an infinite supply of registers. We would need to do register allocation and spill registers to a stack.

Also, while this translation is very simple, it is not particularly efficient. For example, we are doing a lot of register moves when calling functions and when starting the function body, which could be optimized.