CS 4110

Programming Languages & Logics

Lecture 24
Compiling with Continuations

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
λ-calculus with pairs and integers

Roadmap

CS 4120 in one lecture!

Source Language
λ-calculus with pairs and integers

$\downarrow$

Intermediate Language #1
λ-calculus in CPS

Roadmap

CS 4120 in one lecture!

Source Language
λ-calculus with pairs and integers

$\downarrow$

Intermediate Language #1
λ-calculus in CPS

$\downarrow$

Intermediate Language #2
λ-calculus in CPS + Closure Conversion

Roadmap

CS 4120 in one lecture!

Source Language
λ-calculus with pairs and integers

↓

Intermediate Language #1
λ-calculus in CPS

↓

Intermediate Language #2
λ-calculus in CPS + Closure Conversion

↓

Machine Code
Simple RISC-like Assembly

Source Language

We'll start from (untyped) λ-calculus with pairs and integers.

$$\begin{array}{rcl} e & ::= & x \\ & \mid & \lambda x.\, e \\ & \mid & e_1\, e_2 \\ & \mid & (e_1, e_2) \\ & \mid & \#i\, e \\ & \mid & n \\ & \mid & e_1 + e_2 \end{array}$$

Target Language

$$p \quad ::= \quad bb_1; bb_2; \ldots; bb_n$$

A program p consists of a series of *basic blocks* bb.

Target Language

$$
\begin{array}{rcl}
p & ::= & bb_1; bb_2; \ldots; bb_n \\
bb & ::= & lb : c_1; c_2; \ldots; c_n; \text{jump } x
\end{array}
$$

A basic block has a label *lb* and a sequence of commands *c*, ending with “jump.”

Target Language

$$
\begin{array}{rcl}
p & ::= & bb_1; bb_2; \ldots; bb_n \\
bb & ::= & lb : c_1; c_2; \ldots; c_n; \mathsf{jump}\ x \\
c & ::= & \mathsf{mov}\ x_1, x_2
\end{array}
$$

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

Target Language

$$
\begin{array}{rcl}
p & ::= & bb_1; bb_2; \ldots; bb_n \\
bb & ::= & lb : c_1; c_2; \ldots; c_n; \mathrm{jump}\ x \\
c & ::= & \mathrm{mov}\ x_1, x_2 \\
 & | & \mathrm{mov}\ x, n
\end{array}
$$

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

Target Language

$$
\begin{array}{rcl}
p & ::= & bb_1; bb_2; \ldots; bb_n \\
bb & ::= & lb : c_1; c_2; \ldots; c_n; \mathrm{jump}\ x \\
c & ::= & \mathrm{mov}\ x_1, x_2 \\
 & | & \mathrm{mov}\ x, n \\
 & | & \mathrm{mov}\ x, lb
\end{array}
$$

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

Target Language

$$
\begin{array}{rcl}
p & ::= & bb_1; bb_2; \ldots; bb_n \\
bb & ::= & lb : c_1; c_2; \ldots; c_n; \mathrm{jump}\ x \\
c & ::= & \mathrm{mov}\ x_1, x_2 \\
 & | & \mathrm{mov}\ x, n \\
 & | & \mathrm{mov}\ x, lb \\
 & | & \mathrm{add}\ x_1, x_2, x_3
\end{array}
$$

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

Target Language

$$
\begin{array}{rcl}
p & ::= & bb_1; bb_2; \ldots; bb_n \\
bb & ::= & lb : c_1; c_2; \ldots; c_n; \mathrm{jump}\ x \\
c & ::= & \mathrm{mov}\ x_1, x_2 \\
 & | & \mathrm{mov}\ x, n \\
 & | & \mathrm{mov}\ x, lb \\
 & | & \mathrm{add}\ x_1, x_2, x_3 \\
 & | & \mathrm{load}\ x_1, x_2[n]
\end{array}
$$

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

Target Language

$$
\begin{array}{rcl}
p & ::= & bb_1; bb_2; \ldots; bb_n \\
bb & ::= & lb : c_1; c_2; \ldots; c_n; \mathsf{jump}\ x \\
c & ::= & \mathsf{mov}\ x_1, x_2 \\
& | & \mathsf{mov}\ x, n \\
& | & \mathsf{mov}\ x, lb \\
& | & \mathsf{add}\ x_1, x_2, x_3 \\
& | & \mathsf{load}\ x_1, x_2[n] \\
& | & \mathsf{store}\ x_1, x_2[n]
\end{array}
$$

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

Target Language

$$
\begin{array}{rcl}
p & ::= & bb_1; bb_2; \ldots; bb_n \\
bb & ::= & lb : c_1; c_2; \ldots; c_n; \mathrm{jump}\ x \\
c & ::= & \mathrm{mov}\ x_1, x_2 \\
 & | & \mathrm{mov}\ x, n \\
 & | & \mathrm{mov}\ x, lb \\
 & | & \mathrm{add}\ x_1, x_2, x_3 \\
 & | & \mathrm{load}\ x_1, x_2[n] \\
 & | & \mathrm{store}\ x_1, x_2[n] \\
 & | & \mathrm{malloc}\ n
\end{array}
$$

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

$$
\begin{array}{rcl}
c & ::= & \mathrm{let}\, x = e \,\mathrm{in}\, c \\
& \mid & v_1\ v_2\ v_3 \\
& \mid & v_1\ v_2
\end{array}
$$

Commands c look like basic blocks.

Intermediate Language

$$
\begin{array}{rcl}
c & ::= & \text{let}\, x = e \text{ in } c \\
& | & v_1\; v_2\; v_3 \\
& | & v_1\; v_2 \\
e & ::= & v \mid v_1 + v_2 \mid (v_1, v_2) \mid (\#i\; v)
\end{array}
$$

There are no subexpressions in the language!

Intermediate Language

$$
\begin{array}{rcl}
c & ::= & \text{let}\, x = e \text{ in } c \\
& | & v_1\, v_2\, v_3 \\
& | & v_1\, v_2 \\
e & ::= & v \mid v_1 + v_2 \mid (v_1, v_2) \mid (\#i\, v) \\
v & ::= & n \mid x \mid \lambda x.\, \lambda k.\, c \mid \text{halt} \mid \underline{\lambda} x.c
\end{array}
$$

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 halt is the continuation to send the result of the entire program to.

CPS Translation

$$[\![x]\!]\,k \quad = \quad k\,x$$

CPS Translation

$$
\begin{aligned}
[\![x]\!]\,k &= k\,x \\
[\![n]\!]\,k &= k\,n
\end{aligned}
$$

CPS Translation

$$
\begin{aligned}
[\![x]\!]\, k &= k\, x \\
[\![n]\!]\, k &= k\, n \\
[\![(e_1 + e_2)]\!]\, k &= [\![e_1]\!]\big(\underline{\lambda} x_1 . [\![e_2]\!](\underline{\lambda} x_2 .\ \mathsf{let}\ z = x_1 + x_2\ \mathsf{in}\ k\, z)\big)
\end{aligned}
$$

CPS Translation

$$
\begin{aligned}
[\![x]\!]\,k &= k\,x \\
[\![n]\!]\,k &= k\,n \\
[\![(e_1 + e_2)]\!]\,k &= [\![e_1]\!]\big(\underline{\lambda} x_1. [\![e_2]\!](\underline{\lambda} x_2.\ \mathsf{let}\ z = x_1 + x_2\ \mathsf{in}\ k\,z)\big) \\
[\![(e_1, e_2)]\!]\,k &= [\![e_1]\!]\Big(\underline{\lambda} x_1. [\![e_2]\!]\big(\underline{\lambda} x_2.\ \mathsf{let}\ t = (x_1, x_2)\ \mathsf{in}\ k\,t\big)\Big)
\end{aligned}
$$

CPS Translation

$$
\begin{array}{rcl}
[\![x]\!]\,k & = & k\,x \\
[\![n]\!]\,k & = & k\,n \\
[\![(e_1 + e_2)]\!]\,k & = & [\![e_1]\!]\big(\underline{\lambda} x_1.[\![e_2]\!](\underline{\lambda} x_2.\ \mathsf{let}\, z = x_1 + x_2\ \mathsf{in}\ k\,z)\big) \\
[\![(e_1, e_2)]\!]\,k & = & [\![e_1]\!]\Big(\underline{\lambda} x_1.[\![e_2]\!]\big(\underline{\lambda} x_2.\ \mathsf{let}\, t = (x_1, x_2)\ \mathsf{in}\ k\,t\big)\Big) \\
[\![\#i\ e]\!]\,k & = & [\![e]\!](\underline{\lambda} t.\ \mathsf{let}\, y = \#i\ t\ \mathsf{in}\ k\,y)
\end{array}
$$

CPS Translation

$$
\begin{array}{rcl}
[\![x]\!]\,k & = & k\,x \\
[\![n]\!]\,k & = & k\,n \\
[\![(e_1 + e_2)]\!]\,k & = & [\![e_1]\!]\big(\underline{\lambda} x_1.[\![e_2]\!](\underline{\lambda} x_2.\ \mathsf{let}\, z = x_1 + x_2 \,\mathsf{in}\, k\,z)\big) \\
[\![(e_1, e_2)]\!]\,k & = & [\![e_1]\!]\Big(\underline{\lambda} x_1.[\![e_2]\!]\big(\underline{\lambda} x_2.\ \mathsf{let}\, t = (x_1, x_2) \,\mathsf{in}\, k\,t\big)\Big) \\
[\![\#i\;e]\!]\,k & = & [\![e]\!](\underline{\lambda} t.\ \mathsf{let}\, y = \#i\;t \,\mathsf{in}\, k\,y) \\
[\![\lambda x.\,e]\!]\,k & = & k\,(\lambda x.\,\lambda k'.\,[\![e]\!]\,k')
\end{array}
$$

CPS Translation

$$
\begin{array}{rcl}
[\![x]\!]\,k & = & k\,x \\
[\![n]\!]\,k & = & k\,n \\
[\![(e_1+e_2)]\!]\,k & = & [\![e_1]\!]\big(\underline{\lambda} x_1.[\![e_2]\!](\underline{\lambda} x_2.\ \mathsf{let}\,z = x_1 + x_2\ \mathsf{in}\ k\,z)\big) \\
[\![(e_1,e_2)]\!]\,k & = & [\![e_1]\!]\Big(\underline{\lambda} x_1.[\![e_2]\!]\big(\underline{\lambda} x_2.\ \mathsf{let}\,t = (x_1,x_2)\ \mathsf{in}\ k\,t\big)\Big) \\
[\![\#i\ e]\!]\,k & = & [\![e]\!](\underline{\lambda} t.\ \mathsf{let}\,y = \#i\ t\ \mathsf{in}\ k\,y) \\
[\![\lambda x.\,e]\!]\,k & = & k\,(\lambda x.\,\lambda k'.\,[\![e]\!]\,k') \\
[\![e_1\ e_2]\!]\,k & = & [\![e_1]\!]\Big(\underline{\lambda} f.[\![e_2]\!]\big(\underline{\lambda} v.f\,v\,k\big)\Big)
\end{array}
$$

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 $[\![(\lambda a. \#1\, a)\ (3,4)]\!]\ k$, using $k = \mathsf{halt}$.

$$\begin{array}{rl} & [\![(\lambda a.\ \#1\, a)\ (3,4)]\!]\ k \\ = & [\![\lambda a.\ \#1\, a]\!]\ (\underline{\lambda} f.\ [\![(3,4)]\!](\underline{\lambda} v.\ f\, v\, k)) \end{array}$$

Example

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

$$
\begin{array}{ll}
& [\![(\lambda a.\, \#1\, a)\ (3,4)]\!]\ k \\
= & [\![\lambda a.\, \#1\, a]\!]\ (\underline{\lambda} f.\, [\![(3,4)]\!](\underline{\lambda} v.\, f\, v\, k)) \\
= & (\underline{\lambda} f.\, [\![(3,4)]\!](\underline{\lambda} v.\, f\, v\, k))\ (\lambda a.\, \lambda k'.\, [\![\#1\, a]\!]\ k')
\end{array}
$$

Example

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

$$
\begin{array}{ll}
 & [\![(\lambda a.\,\#1\ a)\ (3,4)]\!]\ k \\
= & [\![\lambda a.\,\#1\ a]\!]\ (\underline{\lambda} f.\,[\![(3,4)]\!](\underline{\lambda} v.\,f\,v\,k)) \\
= & (\underline{\lambda} f.\,[\![(3,4)]\!](\underline{\lambda} v.\,f\,v\,k))\ (\lambda a.\,\lambda k'.\,[\![\#1\ a]\!]\ k') \\
= & (\underline{\lambda} f.\,[\![3]\!]\Big(\underline{\lambda} x_1.[\![4]\!](\underline{\lambda} x_2.\ \text{let}\ b = (x_1,x_2)\ \text{in}\ (\underline{\lambda} v.\,f\,v\,k)\ b)\Big) \\
 & \qquad (\lambda a.\,\lambda k'.\,[\![\#1\ a]\!]\ k')
\end{array}
$$

Example

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

$$
\begin{aligned}
& [\![(\lambda a.\ \#1\ a)\ (3,4)]\!]\ k \\
= \quad & [\![\lambda a.\ \#1\ a]\!]\ (\underline{\lambda} f.\ [\![(3,4)]\!](\underline{\lambda} v.\ f\ v\ k)) \\
= \quad & (\underline{\lambda} f.\ [\![(3,4)]\!](\underline{\lambda} v.\ f\ v\ k))\ (\lambda a.\ \lambda k'.\ [\![\#1\ a]\!]\ k') \\
= \quad & (\underline{\lambda} f.\ [\![3]\!]\Big(\underline{\lambda} x_1.[\![4]\!](\underline{\lambda} x_2.\ \text{let}\ b = (x_1, x_2)\ \text{in}\ (\underline{\lambda} v.\ f\ v\ k)\ b)\Big) \\
& \qquad (\lambda a.\ \lambda k'.\ [\![\#1\ a]\!]\ k') \\
= \quad & (\underline{\lambda} f.\ \Big(\underline{\lambda} x_1.\ (\underline{\lambda} x_2.\ \text{let}\ b = (x_1, x_2)\ \text{in}\ (\underline{\lambda} v.\ f\ v\ k)\ b)\ 4\Big)\ 3) \\
& \qquad (\lambda a.\ \lambda k'.\ [\![\#1\ a]\!]\ k')
\end{aligned}
$$

Example

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

$$
\begin{array}{rl}
& [\![(\lambda a.\ \#1\ a)\ (3,4)]\!]\ k \\
= & [\![\lambda a.\ \#1\ a]\!]\ (\underline{\lambda} f.\ [\![(3,4)]\!](\underline{\lambda} v.\ f\ v\ k)) \\
= & (\underline{\lambda} f.\ [\![(3,4)]\!](\underline{\lambda} v.\ f\ v\ k))\ (\lambda a.\ \lambda k'.\ [\![\#1\ a]\!]\ k') \\
= & (\underline{\lambda} f.\ [\![3]\!]\Big(\underline{\lambda} x_1.[\![4]\!](\underline{\lambda} x_2.\ \text{let}\ b = (x_1,x_2)\ \text{in}\ (\underline{\lambda} v.\ f\ v\ k)\ b)\Big) \\
& \qquad (\lambda a.\ \lambda k'.\ [\![\#1\ a]\!]\ k') \\
= & (\underline{\lambda} f.\ \Big(\underline{\lambda} x_1.\ (\underline{\lambda} x_2.\ \text{let}\ b = (x_1,x_2)\ \text{in}\ (\underline{\lambda} v.\ f\ v\ k)\ b)\ 4\Big)\ 3) \\
& \qquad (\lambda a.\ \lambda k'.\ [\![\#1\ a]\!]\ k') \\
= & (\underline{\lambda} f.\ \Big(\underline{\lambda} x_1.\ (\underline{\lambda} x_2.\ \text{let}\ b = (x_1,x_2)\ \text{in}\ (\underline{\lambda} v.\ f\ v\ k)\ b)\ 4\Big)\ 3) \\
& \qquad (\lambda a.\ \lambda k'.\ [\![a]\!](\underline{\lambda} t.\ \text{let}\ y = \#1\ t\ \text{in}\ k'\ y))
\end{array}
$$

Optimization

Clearly, the translation generates a lot of administrative λs!

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

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:

$$(\underline{\lambda} x.e)\, y \rightarrow e\{y/x\}$$

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:

$$(\underline{\lambda} x.e)\, y \rightarrow e\{y/x\}$$

Other lambdas can be converted into lets:

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

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:

$$(\underline{\lambda} x.e)\, y \rightarrow e\{y/x\}$$

Other lambdas can be converted into lets:

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

We can also perform administrative η-reductions:

$$\underline{\lambda} x.k\, x \rightarrow k$$

Example, Redux

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

let $f = \lambda\, a.\, \lambda\, k'.$ let $y = \#1\ a$ in $k'\ y$ in
let $x_1 = 3$ in
let $x_2 = 4$ in
let $b = (x_1,\ x_2)$ in
$f\ b\ k$

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
λ-calculus with pairs and integers

Intermediate Language #1
λ-calculus in CPS

Intermediate Language #2
λ-calculus in CPS + Closure Conversion

Closure Conversion

The next step is to bring all λs to the top level, with no nesting.

$$
\begin{array}{rcl}
P & ::= & \mathrm{let}\, x_f = \lambda x_1.\ \ldots \lambda x_n.\, \lambda k.\, c \;\mathrm{in}\; P \\
& | & \mathrm{let}\, x_c = \lambda x_1.\ \ldots \lambda x_n.\, c \;\mathrm{in}\; P \\
& | & c \\
c & ::= & \mathrm{let}\, x = e \;\mathrm{in}\; c \mid x_1\, x_2 \ldots\ x_n \\
e & ::= & n \mid x \mid \mathrm{halt} \mid x_1 + x_2 \mid (x_1, x_2) \mid \#i\, x
\end{array}
$$

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:

$$\begin{array}{l}
[\![\lambda x.\, \lambda k.\ c]\!]\,\sigma = \\
\quad \text{let}\,(c', \sigma') = [\![c]\!]\,\sigma \text{ in} \\
\quad \text{let}\, y_1, \ldots, y_n = \mathit{fvs}(\lambda x.\, \lambda k.\, c') \text{ in} \\
\quad (f\, y_1\ \ldots\ y_n,\ \sigma'[f \mapsto \lambda y_1.\ \ldots \lambda y_n.\, \lambda x.\, \lambda k.\ c']) \text{ where } f \text{ fresh}
\end{array}$$

Closure Conversion

The main part of the translation is:

$$
\begin{array}{l}
[\![\lambda x.\, \lambda k.\ c]\!]\,\sigma = \\
\quad \text{let}\,(c', \sigma') = [\![c]\!]\,\sigma \text{ in} \\
\quad \text{let}\, y_1, \ldots, y_n = fvs(\lambda x.\, \lambda k.\, c') \text{ in} \\
\quad (f\, y_1\ \ldots\ y_n,\ \sigma'[f \mapsto \lambda y_1.\ \ldots \lambda y_n.\, \lambda x.\, \lambda k.\ c']) \text{ where } f \text{ fresh}
\end{array}
$$

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:

$$
\begin{array}{l}
[\![\lambda x.\, \lambda k.\ c]\!]\, \sigma = \\
\quad \text{let}\ (c', \sigma') = [\![c]\!]\, \sigma\ \text{in} \\
\quad \text{let}\ y_1, \ldots, y_n = \mathit{fvs}(\lambda x.\, \lambda k.\, c')\ \text{in} \\
\quad (f\, y_1\ \ldots\ y_n,\ \sigma'[f \mapsto \lambda y_1.\ \ldots \lambda y_n.\, \lambda x.\, \lambda k.\ c'])\ \text{where}\ f\ \text{fresh}
\end{array}
$$

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 σ replaces the entire lambda with $(f\, y_n\ \ldots\ y_n)$.

Closure Conversion

The main part of the translation is:

$$
\begin{array}{l}
[\![\lambda x.\, \lambda k.\ c]\!]\, \sigma = \\
\quad \text{let}\, (c', \sigma') = [\![c]\!]\, \sigma \text{ in} \\
\quad \text{let}\, y_1, \ldots, y_n = \mathit{fvs}(\lambda x.\, \lambda k.\, c') \text{ in} \\
\quad (f\, y_1\ \ldots\ y_n,\ \sigma'[f \mapsto \lambda y_1.\ \ldots \lambda y_n.\, \lambda x.\, \lambda k.\ c']) \text{ where } f \text{ fresh}
\end{array}
$$

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 σ 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 λs.

Roadmap

Source Language
λ-calculus with pairs and integers

↓

Intermediate Language #1
λ-calculus in CPS

↓

Intermediate Language #2
λ-calculus in CPS + Closure Conversion

↓

Machine Code
Simple RISC-like Assembly

Code Generation

$$\begin{aligned}\mathcal{P}[\![c]\!] = \ & \mathsf{main} : \mathcal{C}[\![c]\!]; \\ & \mathsf{halt} :\end{aligned}$$

Code Generation

$$
\begin{aligned}
\mathcal{P}[\![\mathsf{let}\, x_f = \lambda x_1.\ \ldots \lambda x_n.\, \lambda k.\, c \,\mathsf{in}\, p]\!] = \ & x_f : \mathsf{mov}\, x_1, a_1; \\
& \quad \vdots \\
& \mathsf{mov}\, x_n, a_n; \\
& \mathsf{mov}\, k, ra; \\
& \mathcal{C}[\![c]\!]; \\
& \mathcal{P}[\![p]\!]
\end{aligned}
$$

Code Generation

$$
\begin{aligned}
\mathcal{P}[\![\mathsf{let}\, x_c = \lambda x_1.\ \ldots \lambda x_n.\, c\ \mathsf{in}\ p]\!] = \ & x_c : \mathsf{mov}\, x_1, a_1; \\
& \quad \vdots \\
& \mathsf{mov}\, x_n, a_n; \\
& \mathcal{C}[\![c]\!]; \\
& \mathcal{P}[\![p]\!]
\end{aligned}
$$

Code Generation

$$\begin{aligned}\mathcal{C}[\![\mathsf{let}\, x = n \;\mathsf{in}\; c]\!] = \;& \mathsf{mov}\, x, n;\\ & \mathcal{C}[\![c]\!]\end{aligned}$$

Code Generation

$$
\begin{aligned}
\mathcal{C}[\![\mathsf{let}\ x_1 = x_2\ \mathsf{in}\ c]\!] = \ & \mathsf{mov}\ x_1, x_2; \\
& \mathcal{C}[\![c]\!]
\end{aligned}
$$

Code Generation

$$\begin{aligned}\mathcal{C}[\![\mathsf{let}\,x = x_1 + x_2\ \mathsf{in}\ c]\!] = \ & \mathsf{add}\,x_1, x_2, x;\\ & \mathcal{C}[\![c]\!]\end{aligned}$$

Code Generation

$$
\begin{aligned}
\mathcal{C}[\![\text{let } x = (x_1, x_2) \text{ in } c]\!] = \;& \text{malloc } 2; \\
& \text{mov } x, r_0; \\
& \text{store } x_1, x[0]; \\
& \text{store } x_2, x[1]; \\
& \mathcal{C}[\![c]\!]
\end{aligned}
$$

Code Generation

$$
\begin{aligned}
\mathcal{C}[\![\mathsf{let}\, x = \#i\, x_1 \;\mathsf{in}\; c]\!] = \;& \mathsf{load}\, x, x_1[i-1]; \\
& \mathcal{C}[\![c]\!]
\end{aligned}
$$

Code Generation

$$
\begin{aligned}
\mathcal{C}[\![x\,k\,x_1\ \ldots\ x_n]\!] = \ & \mathrm{mov}\,a_1, x_1; \\
& \quad \vdots \\
& \mathrm{mov}\,a_n, x_n; \\
& \mathrm{mov}\,ra, k; \\
& \mathrm{jump}\,x
\end{aligned}
$$

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.