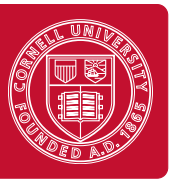
CS 4110 – Programming Languages and Logics
Lecture #18: Compiling with Continuations



Because continuations expose control explicitly, they make a good intermediate language for compilation—control is exposed explicitly in machine code as well. We can show this by writing a translation from a full-featured functional language down to an assembly-like language. This translation will give us a fairly complete recipe for compiling any of the language features we have discussed over the past few lectures all the way down to hardware.

1 Source language

Our source language looks like the lambda calculus with tuples and numbers. We assume the standard call-by-value semantics.

$$e \quad ::= \quad n \mid x \mid \lambda x.\, e \mid e_1\, e_2 \mid (e_1, e_2) \mid \#i\, e \mid e_1 + e_2$$

The target language looks like a simple assembly language:

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

A program p consists of a series of *basic blocks* bb, each with a distinct label lb. Each basic block contains a sequence of commands c and ends with a jump instruction. Commands correspond to assembly language instructions and are largely self-evident; the only one that is high-level is the malloc instruction, which allocates n words of space and places the address of the space into a special register r_0. (This can be implemented as simply as add r_0, r_0, $-n$ if we are not worried about garbage.)

The jump instruction is an indirect jump. It makes the program counter take the value of the argument register: essentially, jump x acts like mov *pc*, x.

2 Intermediate language #1

The first intermediate language, IL1, is in continuation-passing style:

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

There are a few things to note about the intermediate language:

- Lambda abstractions corresponding to continuations are marked with a underline. These are considered *administrative lambdas* that we will eliminate at compile time, either by reducing them or by converting them to real lambdas.
- There are no subexpressions in the language (e does not occur in its own definition).
- Commands c look like basic blocks:

$$
\begin{array}{l}
\mathsf{let}\ x_1 = e_1\ \mathsf{in} \\
\quad \mathsf{let}\ x_2 = e_2\ \mathsf{in} \\
\qquad \ddots \\
\qquad\quad \mathsf{let}\ x_n = e_n\ \mathsf{in} \\
\qquad\qquad v_0\ v_1\ v_2
\end{array}
$$

- Lambdas are not closed and can occur inside other lambdas.

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 $k = \mathsf{halt}$, where halt is the continuation to send the result of the entire program to. Here is the translation from the source to the first intermediate language:

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

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

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

Clearly, the translation generates a lot of administrative lambdas, which will be quite expensive if they are compiled into machine code. To make the code more efficient and compact, we will optimize it using some simple rewriting rules to eliminate administrative lambdas. We can eliminate unnecessary application to a variable, by copy propagation:

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

Other unnecessary administrative lambdas can be converted into lets:

$$(\underline{\lambda} x.c)v \to \mathsf{let}\ x = v\ \mathsf{in}\ c$$

We can also perform administrative η-reductions:

$$\underline{\lambda} x.k\ x \to k$$

If we apply these rules to the expression above, we get

```
let f = λ a. λ k'. let y = #1 a in k' y in
  let x1 = 3 in
    let x2 = 4 in
      let b = (x1, x2) in
        f b k
```

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

The idea of separating administrative terms from real terms and performing a compile-time simplification—often known as *partial evaluation*—is powerful and can be used in many other contexts. Here, it allows us to write a very simple CPS conversion that treats all continuations uniformly, and perform a number of control optimizations. Note that we may not be able to remove all administrative lambdas. Any that cannot be eliminated using the rules above are converted into real lambdas.

3 Intermediate Language #1 → Intermediate Language #2

The next step is the translation from IL1 to IL2. In this intermediate language, all lambdas are at the top level, with no nesting:

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

The key idea behind the translation from IL1 to IL2 is to "lift" all lambdas up to the top level while preserving lexical scope. More specifically, this translation requires the construction of *closures* that capture the free variables of the lambda abstractions. This translation is known as *closure conversion*.

The main part of the translation is captured by the following:

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

The translation takes an expression possibly containing nested lambdas and an environment σ mapping variables to lambdas. It produces a lambda-free expression and an extended environment. Intuitively, the environment collects up the functions that must be lifted to the top level of the program. The translation of $\lambda x.\, \lambda k.\, c$ in the definition 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 $(f\ y_n\ \ldots\ y_n)$. Overall, this has the effect of eliminating all nested lambdas.

4 Intermediate Language #2 $\rightarrow$ Assembly

The final translation from IL2 to assembly is given in Figure 1. Note that ra is the name of the dedicated register that holds the return address. In addition, we assume an infinite supply of registers. We need to do register allocation and possibly spill registers to a stack to obtain working code.

Finally, note that 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.

$$
\begin{aligned}
\mathcal{P}[\![c]\!] = \;& \mathsf{main} : \mathcal{C}[\![c]\!]; \\
& \mathsf{halt} : \\
\mathcal{P}[\![\mathsf{let}\ x_f = \lambda x_1.\ \ldots \lambda x_n.\, \lambda k.\, c\ \mathsf{in}\ p]\!] = \;& x_f : \mathsf{mov}\ x_1, a_1; \\
& \quad \vdots \\
& \mathsf{mov}\ x_n, a_n; \\
& \mathsf{mov}\ k, ra; \\
& \mathcal{C}[\![c]\!]; \\
& \mathcal{P}[\![p]\!] \\
\mathcal{P}[\![\mathsf{let}\ x_c = \lambda x_1.\ \ldots \lambda x_n.\, c\ \mathsf{in}\ p]\!] = \;& x_c : \mathsf{mov}\ x_1, a_1; \\
& \quad \vdots \\
& \mathsf{mov}\ x_n, a_n; \\
& \mathcal{C}[\![c]\!]; \\
& \mathcal{P}[\![p]\!] \\
\mathcal{C}[\![\mathsf{let}\ x_1 = x_2\ \mathsf{in}\ c]\!] = \;& \mathsf{mov}\ x_1, x_2;\ \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![\mathsf{let}\ x = x_1 + x_2\ \mathsf{in}\ c]\!] = \;& \mathsf{add}\ x_1, x_2, x; \\
& \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![\mathsf{let}\ x = (x_1, x_2)\ \mathsf{in}\ c]\!] = \;& \mathsf{malloc}\ 2; \\
& \mathsf{mov}\ x, r_0; \\
& \mathsf{store}\ x_1, x[0]; \\
& \mathsf{store}\ x_n, x[1]; \\
& \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![\mathsf{let}\ x = \#i\ x_1\ \mathsf{in}\ c]\!] = \;& \mathsf{load}\ x, x_1[n]; \\
& \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![x\ k\ x_1\ \ldots\ x_n]\!] = \;& \mathsf{mov}\ a_1, x_1; \\
& \quad \vdots \\
& \mathsf{mov}\ a_n, x_n; \\
& \mathsf{mov}\ ra, k; \\
& \mathsf{jump}\ x
\end{aligned}
$$

Figure 1: Compilation to assembly.