CS 6110 Lecture 16 Compiling with continuations 27 February 2013

Lecturer: Andrew Myers

1 Compiling with continuations

Because continuations expose control explicitly, they make a good intermediate language for compilation, because control is exposed explicitly in machine language as well. We can show this by writing a translation from a stripped-down version of uML to a language similar to assembly.

The result of doing such a translation is that we will have a fairly complete recipe for compiling any of the languages we have talked about into the class down to the hardware.

2 Source language

Our source language looks like the lambda calculus with tuples and numbers, with the obvious (call-by-value) semantics:

$$e \;::=\; n \mid x \mid \lambda x.e \mid e_0\, e_1 \mid (e_0, e_1, \ldots, e_n) \mid (\#n\; e) \mid e_0 + e_1$$

The target language looks more like assembly language:

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

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 $\mathbf{malloc}$ instruction, which allocates n words of space and places the address of the space into a special register r_0. (This could be implemented as simply as $\mathbf{add}\ r_0,\ r_0,\ -n$ if we are not worried about garbage.)

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

3 Intermediate language 1

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

$$\begin{array}{rcl}
v & ::= & n \mid x \mid \lambda kx.c \mid \mathbf{halt} \\
& \mid & \underline{\lambda} x.c \\
e & ::= & v \mid v_0 + v_1 \mid (v_1, v_2, \ldots, v_n) \mid (\#n\; v) \\
c & ::= & \mathbf{let}\; x = e\; \mathbf{in}\; c \\
& \mid & v_0\; v_1\; v_2 \\
& \mid & v_0\; v_1
\end{array}$$

Some 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 a lot like basic blocks:

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

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

$$\begin{array}{rcl}
[\![x]\!]k & = & k\ x \\
[\![n]\!]k & = & k\ n \\
[\![\lambda x.\, e]\!]k & = & k\ (\lambda x k'.([\![e]\!]\ k')) \\
[\![e_0\ e_1]\!]k & = & [\![e_0]\!]\Big(\underline{\lambda} f.[\![e_1]\!]\big(\underline{\lambda} v.(f\ v\ k)\big)\Big) \\
[\![(e_1, e_2, \ldots, e_n)]\!]k & = & [\![e_1]\!]\Big(\underline{\lambda} x_1.\ldots[\![e_n]\!]\big(\underline{\lambda} x_n.\ \mathbf{let}\ t = (x_1, x_2, \ldots, x_n)\ \mathbf{in}\ (k\ t)\big)\Big) \\
[\![\#n\ e]\!]k & = & [\![e]\!](\underline{\lambda} t.\ \mathbf{let}\ y = \#n\ t\ \mathbf{in}\ (k\ y)) \\
[\![(e_1 + e_2)]\!]k & = & [\![e_1]\!]\big(\underline{\lambda} x_1.[\![e_2]\!](\underline{\lambda} x_2.\ \mathbf{let}\ z = x_1 + x_2\ \mathbf{in}\ (k\ z))\big)
\end{array}$$

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

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

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.

$\underline{\beta}$-Reduction

We can eliminate unnecessary application to a variable, by copy propagation:

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

Other unnecessary administrative lambdas can be converted into $\mathbf{let}$s:

$$(\underline{\lambda} x.c)v \longrightarrow \mathbf{let}\ x = v\ \mathbf{in}\ c$$

We can also perform administrative η-reductions:

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

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

```
let $f$ = $(\lambda k' a.\ \mathbf{let}\ y = \#1\ a\ \mathbf{in}\ k'\ y)$ in
 let $x_1$ = 3 in
  let $x_2$ = 4 in
   let $x_3$ = $(x_1,\ x_2)$ 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 *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 reduced using the rules above are converted into real lambdas.

3.1 Tail call optimization

A tail call is a function call that determines the result of another function. A tail-recursive function is one whose recursive calls are all tail calls. Continuations make tail calls easy to optimize. For example, the following program has a tail call from f to g:

```
let $g$ =$\lambda x.\ \#1\ x$ in
 let $f$ = $\lambda x.\ g\ x$
  in
   $f(2,3)$
```

The translation of the body of f is $(g\ (\underline{\lambda y}.k'\ y)\ x)$, which permits optimization by η-reduction to $(g\ k\ x)$. In this optimized code, g does not bother to return to f, but rather jumps directly back to f's caller. This is an important optimization for functional programming languages, where tail-recursive calls that take up linear stack space are converted into loops that take up constant stack space.

4 Intermediate Language 1 → Intermediate Language 2

The next step is the translation from Intermediate language 1 to Intermediate Language 2. In this next intermediate language, all lambdas are at the top level, with no nesting:

$$\begin{array}{rcl}
P & ::= & \mathbf{let}\ x_f = \lambda k x_1 \ldots x_n.\, c\ \mathbf{in}\ P \\
 & & \mid\ \mathbf{let}\ x_c = \lambda x_1 \ldots x_n.\, c\ \mathbf{in}\ P \\
 & & \mid\ c \\
c & ::= & \mathbf{let}\ x = e\ \mathbf{in}\ c\ \mid\ x_0\ x_1 \ldots\ x_n \\
e & ::= & n\ \mid\ x\ \mid\ \mathbf{halt}\ \mid\ x_1 + x_2\ \mid\ (x_1, x_2, \ldots, x_n)\ \mid\ \#n\ x
\end{array}$$

The translation requires the construction of closures that capture all the free variables of the lambda abstractions in intermediate language 1. Since these closures are built out of lambdas that are closed, those lambda values can be hoisted to top level. We have covered closure conversion earlier as a translation.

One way to set up this translation is to translate each expression and command to a new expression or command, plus a set of top-level lambda definitions that can be inserted at the top of the program. An expression will in general also need to rely on some local definitions inserted before the command or definition using that expression. The interesting case is the translation of lambda expressions, which looks like the following.

$$
\begin{aligned}
\mathcal{E}[\![\lambda\vec{x}.\,c]\!] = \langle (f = \lambda s\vec{x}.\,&\mathbf{let}\ y_1 = \#1\ s\ \mathbf{in}\\
&\mathbf{let}\ y_2 = \#2\ s\ \mathbf{in}\\
&\vdots\\
&\mathbf{let}\ y_n = \#n\ s\ \mathbf{in}\ c') :: f',\\
&\hspace{-8em}\langle \mathit{env} = (y_1, \ldots, y_n), \mathit{closure} = (\mathit{env}, f)\rangle,\\
&\hspace{-8em}\mathit{closure}\rangle\\
&\hspace{-11em}\text{where } \{y_1, \ldots, y_n\} = FV(\lambda\vec{x}.\,c)\\
&\hspace{-11em}\text{and } \mathcal{C}[\![c]\!] = \langle f', c'\rangle
\end{aligned}
$$

The first element in the result tuple is a list of top-level function definition including a definition using fresh name f. (Note that the translation of the function body c may also generate additional top-level definitions f'). The second element is a local definition of env, creating the environment to be used in constructing the closure. The third element is the expression that replaces the translated expression.

5 Intermediate Language 2 → Assembly

The translation is given below. Note: ra is the name of the dedicated register that holds the return address.

$$
\begin{aligned}
\mathcal{P}[\![p]\!] &= \ \mathbf{program\ \ for\ \ p}\\
\mathcal{C}[\![c]\!] &= \ \mathbf{sequence\ \ of\ \ commands}\ \ c_1; c_2; \ldots; c_n\\
\\
\mathcal{P}[\![s]\!] &= \ \mathbf{main} : \mathcal{C}[\![c]\!]; \mathbf{halt} :\\
\mathcal{P}[\![\mathbf{let}\ x_f = \lambda k x_1 \ldots x_n.\, c\ \mathbf{in}\ p]\!] &= \ x_f : \mathbf{mov}\ k, ra;\\
&\ \mathbf{mov}\ x_1, a_1;\\
&\ \vdots\\
&\ \mathbf{mov}\ x_n, a_n;\\
&\ \mathcal{C}[\![c]\!];\\
&\ \mathcal{P}[\![p]\!]\\
\mathcal{P}[\![\mathbf{let}\ x_c = \lambda x_1 \ldots x_n.\, c\ \mathbf{in}\ p]\!] &= \ x_c : \mathbf{mov}\ x_1, a_1;\\
&\ \vdots\\
&\ \mathbf{mov}\ x_n, a_n;\\
&\ \mathcal{C}[\![c]\!];\\
&\ \mathcal{P}[\![p]\!]\\
\mathcal{C}[\![\mathbf{let}\ x_1 = x_2\ \mathbf{in}\ c]\!] &= \ \mathbf{mov}\ x_1, x_2; \mathcal{C}[\![c]\!]\\
\mathcal{C}[\![\mathbf{let}\ x_1 = x_2 + x_3\ \mathbf{in}\ c]\!] &= \ \mathbf{add}\ x_1, x_2, x_3; \mathcal{C}[\![c]\!]
\end{aligned}
$$

$$\begin{aligned}
\mathcal{C}[\![\mathbf{let}\; x_0 = (x_1, x_2, \ldots, x_n)\; \mathbf{in}\; c]\!] = \;& \mathbf{malloc}\; n; \\
& \mathbf{mov}\; x_0, r_0; \\
& \mathbf{store}\; x_1, x_0[0]; \\
& \vdots \\
& \mathbf{store}\; x_n, x_0[n-1]; \\
& \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![\mathbf{let}\; x_1 = \#n\; x_2\; \mathbf{in}\; c]\!] = \;& \mathbf{load}\; x_1, x_2[n]; \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![x_0\; k\; x_1\; \ldots\; x_n]\!] = \;& \mathbf{mov}\; ra, k; \\
& \mathbf{mov}\; a_1, x_1; \\
& \;\;\vdots \\
& \mathbf{mov}\; a_n, x_n; \\
& \mathbf{jump}\; x_0
\end{aligned}$$

At this point, we are still assuming an infinite supply of registers. We need to do register allocation and possibly spill registers to a stack to obtain working code.

While this translation is very simple, it is possible to do a better job of generating calling code. For example, we are doing a lot of register moves when calling functions and when starting the function body. These could be optimized.