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 FL 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 λ-calculus with tuples and numbers, with the obvious (call-by-value) semantics:

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

The target language looks more like assembly language:

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

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 could 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.

3 Intermediate Language 1 (IL1)

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

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

Some things to note about IL1:

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

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

- Functions are not closed and can occur in the bodies of other functions.

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 IL1:

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

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

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

Clearly, the translation generates a lot of administrative functions, 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 functions.

$\underline{\beta}$-Reduction

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

$$(\underline{\lambda} x.e)\ y \xrightarrow{1} e\{y/x\}$$

Other unnecessary administrative functions can be converted into lets:

$$(\underline{\lambda} x.c)v \xrightarrow{1} \mathsf{let}\ x = v\ \mathsf{in}\ c$$

We can also perform administrative η-reductions:

$$\underline{\lambda} x.k\ x \xrightarrow{1} k$$

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

```
let f = (λk'a. let y = #1a in k'y) in
  let x1 = 3 in
    let x2 = 4 in
      let x3 = (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 *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 functions. Any that cannot be reduced using the rules above are converted into real functions.

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 =λx. #1 x in
  let f = λx. gx
    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 Translating IL1 to IL2

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

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

The translation requires the construction of closures that capture all the free variables of the λ-abstractions in intermediate language 1. We have covered closure conversion earlier; it too can be phrased as a translation that exploits compile-time partial evaluation.

5 Translating IL2 to Assembly

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

$$
\begin{aligned}
\mathcal{P}[\![p]\!] &= \textsf{program for p} \\
\mathcal{C}[\![c]\!] &= \textsf{sequence of commands } c_1; c_2; \ldots; c_n \\
\\
\mathcal{P}[\![s]\!] &= \textsf{main} : \mathcal{C}[\![c]\!]; \textsf{halt} : \\
\mathcal{P}[\![\textsf{let } x_f = \lambda k x_1 \ldots x_n . c \textsf{ in } p]\!] &= x_f : \textsf{mov } k, ra; \\
&\quad \textsf{mov } x_1, a_1; \\
&\quad \vdots \\
&\quad \textsf{mov } x_n, a_n; \\
&\quad \mathcal{C}[\![c]\!]; \\
&\quad \mathcal{P}[\![p]\!] \\
\mathcal{P}[\![\textsf{let } x_c = \lambda x_1 \ldots x_n . c \textsf{ in } p]\!] &= x_c : \textsf{mov } x_1, a_1; \\
&\quad \vdots \\
&\quad \textsf{mov } x_n, a_n; \\
&\quad \mathcal{C}[\![c]\!]; \\
&\quad \mathcal{P}[\![p]\!] \\
\mathcal{C}[\![\textsf{let } x_1 = x_2 \textsf{ in } c]\!] &= \textsf{mov } x_1, x_2; \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![\textsf{let } x_1 = x_2 + x_3 \textsf{ in } c]\!] &= \textsf{add } x_1, x_2, x_3; \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![\textsf{let } x_0 = (x_1, x_2, \ldots, x_n) \textsf{ in } c]\!] &= \textsf{malloc } n; \\
&\quad \textsf{mov } x_0, r_0; \\
&\quad \textsf{store } x_1, x_0[0]; \\
&\quad \vdots \\
&\quad \textsf{store } x_n, x_0[n-1]; \\
&\quad \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![\textsf{let } x_1 = \#n\ x_2 \textsf{ in } c]\!] &= \textsf{load } x_1, x_2[n]; \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![x_0\ k\ x_1\ \ldots\ x_n]\!] &= \textsf{mov } ra, k; \\
&\quad \textsf{mov } a_1, x_1; \\
&\quad \vdots \\
&\quad \textsf{mov } a_n, x_n; \\
&\quad \textsf{jump } x_0
\end{aligned}
$$

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.

Continuations are great for talking about semantics, but they are also useful more concretely, for programming and for compilation. We look at some of these uses here.

6 setjmp and longjmp

setjmp is a C function which takes a pointer to a buffer. Its operation is to save the state of all registers (including the program counter) into the specified buffer and return 0. longjmp, also a C function, takes as an argument a pointer to a buffer (which is presumed to be a buffer which has already been filled with a previous call to setjmp) and a value. When invoked, it restores all of the registers to the values saved in the buffer and returns the value passed in to the point in the program where setjmp was called (in effect, the program resumes executing right where setjmp was called, except the call will return the value passed in to longjmp). These functions can be used for error handling. setjmp is called before code that may result in an error. If an error occurs in computation, longjmp is called in order to restore initial state and handle the error. For instance:

```
if (setjmp(&jmpbuf))
        // error handling code goes here
else
        do_computation();
        // if error occurs in compute(), call
        // longjmp(jmpbuf, e), where e is the error code
```

The functions setjmp and longjmp can be translated using continuations as follows:

$$
\begin{aligned}
[\![\mathsf{setjmp}\ e]\!]\rho k =& [\![e]\!]\rho(\mathit{CHECK\text{-}LOC}\ (\lambda l\sigma.\, k(\mathit{INT}\ 0)(\mathit{UPDATE}\ \sigma\ l\ (\mathit{CONT}\ k))))\\
[\![\mathsf{longjmp}\ e\ e']\!]\rho k =& [\![e]\!]\rho(\mathit{CHECK\text{-}LOC}\ (\lambda l.\, [\![e']\!]\rho\\
& \qquad (\lambda v.\, \mathit{CHECK\text{-}CONT}(\mathit{LOOKUP}\sigma\ l)(\lambda k'.\, k'\ v\ \sigma))))
\end{aligned}
$$

The translation of setjmp stores a continuation at a new location, while the translation of longjmp restores a continuation from a program location. This is roughly equivalent to restoring the registers and program state of the executing program.