

## 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:

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

The target language looks more like assembly language:

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

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**  $pc, x$ . We can use this instruction to implement other kinds of branch instructions such as absolute jumps and conditional branches.

## 3 Intermediate language 1

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

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

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:

```

let  $x_1 = e_1$  in
  let  $x_2 = e_2$  in
     $\ddots$ 
    let  $x_n = e_n$  in
       $v_0 \ v_1 \ v_2$ 

```

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

The contract of the translation is that  $\llbracket e \rrbracket k$  is an IL1 command that evaluates  $e$  and passes its result to the continuation  $k$ . To translate an entire program, we use  $k = \mathbf{halt}$ , 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:

$$\begin{aligned}
\llbracket x \rrbracket k &= k \ x \\
\llbracket n \rrbracket k &= k \ n \\
\llbracket \lambda x. e \rrbracket k &= k \ (\lambda k' x. (\llbracket e \rrbracket k')) \\
\llbracket e_0 \ e_1 \rrbracket k &= \llbracket e_0 \rrbracket (\lambda f. \llbracket e_1 \rrbracket (\lambda v. (f \ k \ v))) \\
\llbracket (e_1, e_2, \dots, e_n) \rrbracket k &= \llbracket e_1 \rrbracket (\lambda x_1. \dots \llbracket e_n \rrbracket (\lambda x_n. \mathbf{let} \ t = (x_1, x_2, \dots, x_n) \mathbf{in} \ (k \ t))) \\
\llbracket \#n \ e \rrbracket k &= \llbracket e \rrbracket (\lambda t. \mathbf{let} \ y = \#n \ t \mathbf{in} \ (k \ y)) \\
\llbracket (e_1 + e_2) \rrbracket k &= \llbracket e_1 \rrbracket (\lambda x_1. \llbracket e_2 \rrbracket (\lambda x_2. \mathbf{let} \ z = x_1 + x_2 \mathbf{in} \ (k \ z)))
\end{aligned}$$

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

$$\begin{aligned}
&\llbracket (\lambda a. (\#1 \ a)) \ (3, 4) \rrbracket k \\
&= \llbracket \lambda a. (\#1 \ a) \rrbracket (\lambda f. \llbracket (3, 4) \rrbracket (\lambda v. (f \ k \ v))) \\
&= (\lambda f. \llbracket (3, 4) \rrbracket (\lambda v. (f \ k \ v))) (\lambda a k'. \llbracket \#1 \ a \rrbracket k') \\
&= (\lambda f. \llbracket 3 \rrbracket (\lambda x_1. \llbracket 4 \rrbracket (\lambda x_2. \mathbf{let} \ b = (x_1, x_2) \mathbf{in} \ (\lambda v. (f \ k \ v)) \ b))) (\lambda a k'. \llbracket \#1 \ a \rrbracket k') \\
&= (\lambda f. (\lambda x_1. (\lambda x_2. \mathbf{let} \ b = (x_1, x_2) \mathbf{in} \ (\lambda v. (f \ k \ v)) \ b) \ 4) \ 3) (\lambda a k'. \llbracket \#1 \ a \rrbracket k') \\
&= (\lambda f. (\lambda x_1. (\lambda x_2. \mathbf{let} \ b = (x_1, x_2) \mathbf{in} \ (\lambda v. (f \ k \ v)) \ b) \ 4) \ 3) (\lambda a k'. \llbracket a \rrbracket (\lambda t. \mathbf{let} \ y = \#1 \ t \mathbf{in} \ k' \ y)) \\
&= (\lambda f. (\lambda x_1. (\lambda x_2. \mathbf{let} \ b = (x_1, x_2) \mathbf{in} \ (\lambda v. (f \ k \ v)) \ b) \ 4) \ 3) (\lambda a k'. (\lambda t. \mathbf{let} \ y = \#1 \ t \mathbf{in} \ k' \ y) \ a)
\end{aligned}$$

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.

### $\beta$ -Reduction

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

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

We can do the same with constants  $n$ , giving us constant propagation. Other unnecessary administrative lambdas can be converted into **lets**. This is useful when  $v$  is a lambda expression.

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

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

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

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

```
let f = (λk'a. let y = #1 a in k' y) in
let x1 = 3 in
let x2 = 4 in
let x3 = (x1, x2) in
f k x3
```

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 = λx. #1 x in
let f = λx. 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  $\eta$ -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. The term  $(\underline{\lambda}y.k' y)$  represents the “rest of  $f$ ” after the call to  $g$ . Clearly this is a useless computation; tail-call elimination gets rid of this lambda and the stack frame that will be needed to apply it. This is an important optimization for functional programming languages, where tail-recursive calls requiring a linear amount of stack space are optimized into loops that take up constant stack space.

## 4 Intermediate Language 1 $\rightarrow$ Intermediate Language 2

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

$$\begin{aligned} P &::= \text{let } x_f = \lambda k x_1 \dots x_n. c \text{ in } P \\ &\quad | \text{let } x_c = \lambda x_1 \dots x_n. c \text{ in } P \\ &\quad | c \\ c &::= \text{let } x = e \text{ in } c \mid x_0 x_1 \dots x_n \\ e &::= n \mid x \mid \text{halt} \mid x_1 + x_2 \mid (x_1, x_2, \dots, x_n) \mid \#n x \end{aligned}$$

The translation requires the construction of closures that capture all the free variables of the lambda 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 Intermediate Language 2 $\rightarrow$ Assembly

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

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

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.