CS611 Lecture 15 — Compiling with Continuations — 01 October, 2004

Scribe: Shafquat Rahman, Yogeshwer Sharma — Lecturer: Andrew Myers

1 Introduction

Continuations can be used for two main purposes.

- One is to understand the meaning and semantics of the programs.
- Other is to compile the programs to the machine code. They can be used to explain the compilation process itself. CPS is more close to machine-language and makes control explicit, e.g. order of evaluation and procedure call/return control flow.

In this lecture we will talk about the later application of the continuations. For this we will define a source language and a target language and then carry out the translation from former to later with the help of continuations. In the process, we will also define two other languages called intermediate 1 and intermediate 2 and use them for the translation.

The result of doing such a translation is that we will have a fairly complete recipe for compiling any of the language we have talked about into the class to assembly code or maching language.

Section 2 defines the source and target languages and points some of the features of both the languages. Section 3 gives the translation from the source language to first intermediate languages. The transformation from first intermediate language to second one is not here. The final translation from the second intermediate language to the target language is explained in the last section.

2 Language Definition

Our source language looks like lambda calculus with tuples added in it. We also provide number for something to compute on and addition operation to compute the addition in convenient way.

$$\begin{array}{rcl} e & ::= & n \\ & | & x \\ & | & \lambda x.e \\ & | & e_0\ e_1 \\ & | & (e_0, e_1, \ldots, e_n) \\ & | & (\#n\ e) \\ & | & e_0 + e_1 \end{array}$$

The target language looks more like assembly language where we have low-level instructions like add and load. The precise definition follow:

$$\begin{array}{rcl} p & ::= & bb_1; bb_2; \ldots; bb_n \\ bb & ::= & lb : S_1; S_2; \ldots; S_n; \mathsf{jump}\ x \\ S & ::= & \mathsf{mov}\ x_1, x_2 \\ & | & \mathsf{mov}\ x, n \\ & | & \mathsf{mov}\ x, lb \\ & | & \mathsf{add}\ x_1, x_2, x_3 \\ & | & \mathsf{load}\ x_1, x_2(n) \\ & | & \mathsf{cons}\ x_0, x_1, \ldots, x_n \end{array}$$

One important thing to notice in this language is that we do not have the store instruction in the target language. This is because we want it to be functional language (and there are no side-effects in functional

$$
\begin{array}{rcl}
v & ::= & x \\
 & | & \lambda k x.S \\
 & | & \mathsf{halt} \\
 & | & \overline{\lambda} x.S \\
e & ::= & v \\
 & | & v_0 + v_1 \\
 & | & (v_1, v_2, \ldots, v_n) \\
 & | & (\#n\ v) \\
S & ::= & \mathsf{let}\ x = e\ \mathsf{in}\ S \\
 & | & v_0\ v_1\ v_2 \\
 & | & v_0\ v_1
\end{array}
$$

Figure 1: Definition of the first intermediate language

language). cons operation above allocates n "locations" and then puts the "address" of the first one in x_0. jump instruction is the indirect jump which makes the "program counter" to become the value of the following register. The names of other instruction are instructive in themselves.

3 The First Translation

In this section, we will describe the translation from the source language to intermediate language 1 which is defined in figure 1.

Some of the things to note about the intermediate language:

- We introduce $\overline{\lambda}$ in the language (called "administrative" lambda) which is used during the translation and which will not appear in the translated version.
- There are no subexpressions in the language (e does not occur in its definition itself).
- The lambda's are not closed and are not lifted either.
- Statements look like basic blocks.

A typical statement in the statement in the language can be written as follows:

```
let x_1 = e_1 in
  let x_2 = e_2 in
    ·
     ·
      ·
      let x_n = e_n in
        v_0 v_1 v_2
```

One more point to note: the lambdas we introduce during the translation are all administrative lambdas, they will be removed at the end. The real lambdas (from lambda expressions) transform to the ordinary lambda.

The contract of the translation is that $[\![e]\!]k$ will evaluate e and pass its result to the continuation k. Here is the translation from the source to the first intermediate language.

$$
\begin{array}{rcl}
[\![x]\!]k & = & k\ x \\
[\![\lambda x.e]\!]k & = & k\ (\lambda x k'.([\![e]\!]\ k')) \\
[\![e_0\ e_1]\!]k & = & [\![e_0]\!]\Big(\overline{\lambda} f.[\![e_1]\!]\big(\overline{\lambda} v.(f\ v\ k)\big)\Big)
\end{array}
$$

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

Let us make that all concrete by going through an example. We evaluate the expression $[\![(\lambda a.(\#1\ a))\ (3,4)]\!]k$ and in the last we will put $k = \mathsf{halt}$.

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

But this translation seems to be quite cumbersome. In particular, it generates a lot of administrative lambdas that 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.

$\overline{\beta}$-Reduction

$$(\overline{\lambda} x.e)\ v \xrightarrow{\beta} e\{v/x\}$$

The left-hand side is a $\overline{\beta}$-redex and can be recognized at compile time. In compiler terms, this beta reduction is a simple form of copy propagation.

Another $\overline{\beta}$-Reduction

$$(\overline{\lambda} x.e)e' \longrightarrow \mathsf{let}\ x = e'\ \mathsf{in}\ e$$

These transformation rule is needed because we have to remove administrative lambdas ($\overline{\lambda}$) if they are not really needed.

Reduction ($\overline{\eta}$)

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

This one is eta-reduction for an administrative lambda.

If we apply these two to the expressions above, we get

```
let f = λk'a.  (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.

4 Intermediate Language 1 $\longrightarrow$ Intermediate Language 2

The next step is the translation from Intermediate language 1 to Intermediate Language 2. The grammar for this intermediate language is

$$
\begin{aligned}
P &::= \mathsf{let}\ x_f = \lambda k x_1 \ldots \lambda k x_n . S\ \mathsf{in}\ P \mid S \\
v &::= x \mid \mathsf{halt} \mid n \\
S &::= \mathsf{let}\ x = e\ \mathsf{in}\ S \mid v_0\ v_1 \ldots\ v_n \\
e &::= v \mid v_0 + v_1 \mid (v_1, v_2, \ldots, v_n) \mid (\#n\ v)
\end{aligned}
$$

We will not elaborate on the actual mechanics of this translation. A couple of key issues are listed below

- Requires closure conversion and lambda lifting - all lambdas are at the top level in p.
- *key idea*: explicit abstraction over free variables
- Partial evaluation is an integral part in some lambda lifting schemes

5 Intermediate Language 2 $\longrightarrow$ Assembly

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

$$
\begin{aligned}
&P[\![p]\!] : \mathsf{program\ for\ p} && (1)\\
&S[\![S]\!] : \mathsf{sequence\ of\ statements}\ s_1; s_2; \ldots; s_n && (2)\\
&P[\![S]\!] = \mathsf{main} : S[\![S]\!] && (3)\\
&P[\![\mathsf{let}\ x_f = \lambda k x_1 \ldots \lambda k x_n . S\ \mathsf{in}\ p]\!] = x_f\ : \mathsf{mov}\ k, ra; && (4)\\
&\mathsf{mov}\ x_1, a_1; \ldots\ ; x_n, a_n;\ S[\![S]\!]; P[\![p]\!] && (5)\\
&S[\![\mathsf{let}\ x_1 = x_2 \mathsf{in} S]\!] = \mathsf{mov}\ x1, x2; S[\![S]\!] && (6)\\
&S[\![\mathsf{let}\ x_1 = x_2 + x_3\ \mathsf{in}\ S]\!] = \mathsf{add}\ x_1, x_2, x_3; S[\![S]\!] && (7)\\
&S[\![\mathsf{let}\ x_0 = (x_1, x_2, \ldots, x_n)\ \mathsf{in}\ S]\!] = && (8)\\
&\mathsf{Cons}\ x_0, x_1, x_2, \ldots, x_n;\ S[\![S]\!] && (9)\\
&S[\![\mathsf{let}\ x_1 = \#n\ x_2\ \mathsf{in}\ S]\!] =\ \mathsf{load}\ x_1, x_2(n) && (10)\\
&S[\![x_0\ k\ x_1\ \ldots\ x_n]\!] = \mathsf{mov}\ ra, k; && (11)\\
&\qquad\qquad\qquad \mathsf{mov}\ a_1, x_1; && (12)\\
&\qquad\qquad\qquad \vdots && (13)\\
&\qquad\qquad\qquad \mathsf{mov}\ a_n, x_n; && (14)\\
&\qquad\qquad\qquad \mathsf{jump}\ x_0 && (15)
\end{aligned}
$$

This translation is mostly quite direct, though 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. Note that there are no administrative lambdas at this stage. Also the rules do not specify register allocation.

6 Tail recursion optimizaiton

A function is called tail recursive when the result of the function in the non-base case is determined solely by the result of calling the function with a simpler argument. More generally, a tail call is a call that provides the result of a containing function body.
For example the following program has a tail call from f to g that allows an optimization:

```
let g =λx.#1 x in
  let f = λx.g x
    in
      f(2)
```

The translation of the body of f is $g(\overline{\lambda y}.k'y)x$, which permits optimization by eta reduction to gkx. 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 by this optimization into loops that take up constant stack space. This is one example of a compile-time optimization that follows from the translation rules stated.