Scribe: Ian Kash, Yee Jiun Song Lecturer: Andrew Myers

Today's notes cover:

- Continuation-passing style (CPS)
- CPS semantics
- CPS conversion
- uML error checking

1 Continuation-passing style

Direct semantics have a number of problems including:

- The control structure is implicitly preserved.
- Constructions like goto and exceptions are very hard to describe exactly.
- They don't really model low-level code and miss some important insights about compilation.

We want a semantics with no evaluation context and no implicit control structures. The solution is continuation-passing style.

Consider the statement $\mathsf{if}\ x < 0\ \mathsf{then}\ x\ \mathsf{else}\ x+1$. We can think of this as $(\mathsf{if}\ [\cdot]\ \mathsf{then}\ x\ \mathsf{else}\ x+1)[x < 0]$, which we can represent very naturally as $(\lambda y.\ \mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ x+1)(x < 0)$. We call this function a *continuation*. Intuitively it is the place to send the result of the current computation $(x < 0)$ in order to continue the computation.

2 CPS Semantics

Our grammar for the lambda calculus was:

$$e \ ::= \ x \ \mid \ \lambda x.\, e_0 \ \mid \ e_0\ e_1$$

Our grammar for the CPS lambda calculus will be:

$$\begin{aligned} e &::= x \mid \lambda x_1 \ldots x_n.\, s \mid \mathsf{halt} \\ s &::= e_0\ e_1 \ldots e_n \end{aligned}$$

Some things to note:

- We will only actually need $n = 1$ and $n = 2$
- All expressions are values: there are no subexpressions
- An application is not an expression; if it terminates, it evaluates to $\mathsf{halt}\ v$ rather than to a value v.

The small step semantics for this has a single rule: $\overline{(\lambda x_i.\, s)e_i \longrightarrow s\{e_i/x_i\}}$. The final configuration has the form $\mathsf{halt}\ e$. We do not need any evaluation contexts because all expressions are values.

The big step semantics is also quite simple, with only two rules: $\dfrac{s\{e_i/x_i\} \Downarrow v}{(\lambda x_i.\, s)e_i \Downarrow v}$ and $\overline{\mathsf{halt}\ v \Downarrow v}$. The resulting proof tree will not be very tree-like. In fact since each rule has 0 or 1 arguments, it will basically be a stack in which each level of the stack is a step in the small step semantics. This allows for a much simpler interpreter because it can work in a straight line rather than having to make multiple recursive calls. Thus this provides us with a lower-level model of computation.

3 CPS Conversion

To show that we haven't lost any expressive power in going to this simpler calculus, we convert the lambda calculus to the CPS lambda calculus. Our conversions will look like $[\![e]\!]k$ where e is an expresion in the lambda calculus and k is a continuation of the form $\lambda v. .s$. We want this to satisfy the contract that $[\![e]\!]k$ should send the result of e to k. Then our translation of a program e will just be $[\![e]\!]\mathsf{halt}$. We want our translation to satisfy $e \longrightarrow_\lambda^* v. \Leftrightarrow [\![e]\!]\mathsf{halt} \longrightarrow_{CPS}^* v'$ such that if v is a primitive value, $v = v'$. For this purpose divergence is a primitive value, so if one diverges the other should too (i.e $e \Uparrow_\lambda \Leftrightarrow . [\![e]\!]\mathsf{halt} \Uparrow_{CPS}$). Our translation will be (adding numbers as a primitive value):

$$\begin{aligned}
[\![n]\!]k &= kn \\
[\![x]\!]k &= kx \\
[\![\lambda x.\, e]\!]k &= k(\lambda x.\, \lambda k'.\, [\![e]\!]k') \\
[\![e_0\ e_1]\!]k &= [\![e_0]\!](\lambda f.\, [\![e_1]\!](\lambda v.\, fvk))
\end{aligned}$$

Basically a function is translated into a new function that also takes a "return address" and an application supplies that function with both the value and the return address. Now, let's see an example of how this works.

In plain lambda calculus, we have:

$$(\lambda x.\, (\lambda y.\, x))1 \rightarrow \lambda y.\, 1$$

The same example, in CPS:

$$\begin{aligned}
& [\![(\lambda x.\, (\lambda y.\, x))1]\!]\ \mathsf{halt} \\
\Downarrow\ & [\![\lambda x.\, (\lambda y.\, x)]\!](\lambda f.\, [\![1]\!](\lambda v.\, (f v\ \mathsf{halt}))) \\
\Downarrow\ & (\lambda f.\, (\lambda v.\, f v\ \mathsf{halt})1)(\lambda x k'.\, [\![\lambda y.\, x]\!]k') \\
\Downarrow\ & (\lambda f.\, (\lambda v.\, f v\ \mathsf{halt})1)(\lambda x k'.\, k'(\lambda y k''.\, k'' x)) \\
\Downarrow\ & (\lambda v.\, (\lambda x k'.\, k'(\lambda y k''.\, k'' x)v\ \mathsf{halt}))1 \\
\Downarrow\ & (\lambda x k'.\, k'(\lambda y k''.\, k'' x))1\ \mathsf{halt} \\
\Downarrow\ & \mathsf{halt}\ (\lambda y k''.\, k''1) \\
=\ & [\![\lambda y.\, 1]\!]\mathsf{halt}
\end{aligned}$$

Notice that because CPS terms evaluate (when they terminate) to $\mathsf{halt}\ v$ for some value v, if we replace halt with the identity function then CPS translations evaluate to a value corresponding to the value of the source lambda calculus term.

Continuation semantics is also known as CPS semantics or *standard* semantics.

4 Error Checking

Now let us use CPS semantics to augment our previously defined uML language translation so that it supports error checking.

$[\![e]\!]\rho k$: The contract is that we will send the result of evaluating e in the naming environment ρ to the continuation k. In addition, we want to be able to catch errors that may occur during the evaluation. We create the following type tags to keep track of types:

$$
\begin{aligned}
boolean &: 0 \\
integers &: 1 \\
tuples &: 2 \\
functions &: 3
\end{aligned}
$$

Using these type tags, we can now define rules for error checking. We begin with rules for translating values:

$$
\begin{aligned}
\mathcal{V}[\![b]\!]\rho &= (0, b) \\
\mathcal{V}[\![n]\!]\rho &= (1, n) \\
\mathcal{V}[\![(v_1 ... v_n)]\!]\rho &= (2, (\mathcal{V}[\![v_1]\!], ... \mathcal{V}[\![v_n]\!])) \\
\mathcal{V}[\![\lambda x.\, e]\!]\rho &= (3, \lambda x k.\, [\![e]\!]\rho k) \\
[\![x]\!]\rho k &= k(\rho \text{“}x'') \\
[\![v]\!]\rho k &= k(\mathcal{V}[\![v]\!]\rho) \\
[\![e_0 e_1]\!]\rho k &= [\![e_0]\!](\lambda p.\, \mathsf{let}(tag, f) = p \text{ in if } t! = 3 \text{ then error else } [\![e_1]\!]\rho(\lambda v.\, f v k))
\end{aligned}
$$

We can simplify this by defining a helper function check-fn:

$$
\text{check-fn} = \lambda p k.\, \mathsf{let}(t, f) = p \text{ in if } t! = 3 \text{ then error else } k f
$$

Now,

$$
[\![e_0\ e_1]\!]\rho k = [\![e_0]\!]\rho(\lambda p.\, \text{check-fn } p(\lambda f.\, [\![e_1]\!]\rho(\lambda v.\, f v k)))
$$

Similarly,

$$
\begin{aligned}
[\![\text{if } e_0 \text{ then } e_1 \text{ else } e_2]\!]\rho k &= [\![e_0]\!]\rho(\lambda p.\, \text{check-bool } p(\lambda b.\, \text{if } b \text{ then } [\![e_1]\!]\rho k \text{ else } [\![e_2]\!]\rho k)) \\
[\![\text{let } x = e_1 \ in \ e_2]\!]\rho k &= [\![e_1]\!]\rho(\lambda p.\, [\![e_2]\!] \text{ extend}(\rho, \text{“}x'', p) k) \\
[\![(e_0,\ ...\ ,\ e_n)]\!]\rho k &= [\![e_0]\!]\rho(\lambda x_0.\, [\![e_1]\!]\rho(\lambda x_1. ... [\![e_n]\!]\rho(\lambda x_n.\, k(2, (x_0,\ ...\ ,\ x_n))))) \\
[\![\#n\ e]\!]\rho k &= [\![e]\!]\rho(\lambda p.\, \text{check-tup } p(\lambda t.\, k(\#n\ t)))
\end{aligned}
$$