CS611 Lecture 15 — Semantics via CPS translation — 10/3/2001

Scribe: Walter Chang, Dennis Yeh — Lecturer: Andrew Myers

1 Continuation Passing

Last time we introduced a version of λ calculus that forced us to write code in continuation-passing style (CPS).

$$\begin{array}{rcl} e & ::= & x \mid \lambda\ x\ s \mid \lambda(x,y)\ s \\ s & ::= & e_0\ e_1 \mid e_0\ (e_1, e_2) \end{array}$$

Note that in this language, functions can be applied to 1 or 2 arguments, which we didn't permit in the CBV λ calculus. However, we can also write continuation-passing style code even in the ordinary lambda calculus, which will allow simpler definitional translations of many language features.

To show that we can do CPS translation even into ordinary λ calculus, we define a translation function $\mathcal{CPS}[\![\cdot]\!]$. The contract that this translation must satisfy is that given an expression e and a CPS term k that represents a continuation, $\mathcal{CPS}[\![e]\!]k$ is a computation that sends the result of e to k. The continuation k has the form $(\lambda v\ \ ...)$, and the idea is that we will invoke it with $(k\ v)$ where v is the value of e.

Assuming that $\mathcal{CPS}$ operates on subterms according to this contract, we can write a new CPS translation, from the λ calculus into itself:

$$\begin{array}{rcll} \mathcal{CPS}[\![x]\!]k & = & k\ x & \\ \mathcal{CPS}[\![\lambda x\ e]\!]k & = & k(\lambda x\ (\lambda k'\ \mathcal{CPS}[\![e]\!]\ k')) & (\text{where } k' \notin FV[\![e]\!]) \\ \mathcal{CPS}[\![e_0\ e_1]\!]k & = & \mathcal{CPS}[\![e_0]\!](\lambda f\ \mathcal{CPS}[\![e_1]\!]\ (\lambda x\ (f\ x)\ k)) & (\text{where } f \notin FV[\![e_1]\!]) \end{array}$$

Note that in the second rule, we translate an abstraction to a term that might appear to violate the CPS rules, since the body of an abstraction is an abstraction instead of an application. However, these nested abstractions are simply acting as a two-argument function, because they are only invoked as the f in the third rule. The function values generated in the second rule are never curried. The translation of an abstraction has an η redex in it; however, reducing this redex doesn't really help make our translations any simpler because of the way that we've written our rules with the application to the continuation already folded in.

For program termination, a special continuation called *halt* will cause the termination of the program. One natural way to define *halt* is as the identity term:

$$\mathit{halt} \overset{def}{=} \lambda z\ z$$

Recalling our contract for $\mathcal{CPS}[\![\cdot]\!]$, this makes sense because the translation using the *halt* continuation will result in the value of the translated expression being "sent" to the identity function, and thus popping out immediately as the result of the whole computation:

$$e \longmapsto^* v \Rightarrow (\mathcal{CPS}[\![e]\!]\mathit{halt} \longmapsto^* ((\lambda z\ z)\ \mathcal{V}[\![v]\!]) \longmapsto^* \mathcal{V}[\![v]\!])$$

where $\mathcal{V}[\![v]\!]$ is the CPS value translation that is implicit in the CPS translation above:[1]

$$\begin{array}{rcll} \mathcal{CPS}[\![v]\!]k & = & k\ \mathcal{V}[\![v]\!] & \\ \mathcal{V}[\![x]\!] & = & x & \\ \mathcal{V}[\![\lambda x\ e]\!] & = & \lambda x\ (\lambda k'\ \mathcal{CPS}[\![e]\!]k') & (\text{where } k' \notin FV[\![e]\!]) \end{array}$$

[1]Actually, the CPS expression doesn't necessarily step to exactly $\mathcal{V}[\![v]\!]$; in general it steps to a term that is equivalent and can be determined to be so through a set of equational rules. This is the same situation that arose in our CBN$\rightarrow$CBV translation.

CPS makes control transfer explicit—continuations are explicitly represented in the language as first-class values (functions) that serve as control contexts. CPS is really a low-level language; we can think of the function invocations as being the same thing as indirect jumps in assembly: they don't need to return because there is never anything to return to. This corresponds exactly to the fact that in the operational semantics for CPS (seen last time), we don't need evaluation contexts. Because CPS is low-level, the intermediate languages used in modern compilers are often CPS, especially compilers for functional languages [2]. Language semantics written in CPS are sometimes called *continuation semantics* or *standard semantics* (as opposed to the *direct* semantics we have been using so far).

2 uF → uF with error checking

Let's now see how we can map uF to uF with error checking, but using a CPS translation. We'll also explicitly keep track of the naming environment ρ. Our translation function $\mathcal{D}[\![e]\!]\rho k$ sends e in environment ρ to continuation k.

First, we tag values to make it possible to distinguish different types of values:

$$\begin{array}{rcl}
\mathcal{V}[\![\mathit{error}]\!] &=& \langle 0,\ \#\mathsf{u}\rangle \\
\mathcal{V}[\![\#\mathsf{u}]\!] &=& \langle 1,\ \#\mathsf{u}\rangle \\
\mathcal{V}[\![\#\mathsf{t}]\!] &=& \langle 2,\ \#\mathsf{t}\rangle \\
\mathcal{V}[\![\#\mathsf{f}]\!] &=& \langle 2,\ \#\mathsf{f}\rangle \\
\mathcal{V}[\![n]\!] &=& \langle 3,\ n\rangle \\
\mathcal{V}[\![\langle v_1,\ v_2\rangle]\!] &=& \langle 4,\ \langle \mathcal{V}[\![v_1]\!],\ \mathcal{V}[\![v_2]\!]\rangle\rangle \\
\mathcal{V}[\![\lambda x\ e]\!] &=& \langle 5,\ \mathcal{V}[\![\lambda x\ e]\!]\rangle
\end{array}$$

We also define various type-checking abstractions. These are functions that given a continuation, will produce a continuation that filters the values passed to it. Values of the appropriate type will be passed through to the underlying continuation; other values will cause an immediate halt with error. For example, we can use *check-fn* if we want to check for function values:

$\mathit{check\text{-}fn} = (\lambda k \lambda p\ \mathsf{if}\ (\mathsf{left}\ p) == 5\ \mathsf{then}\ k\ (\mathsf{right}\ p)\ \mathsf{else}\ \mathit{halt}\ \langle 0,\ \#\mathsf{n}\rangle)$,
Other checking functions, such as *check-bool* and *check-pair*, can be defined similarly.

$$\begin{array}{rcl}
\mathcal{D}[\![x]\!]\rho k &=& k\ (\rho\ \text{“}x\text{”}) \\
\mathcal{D}[\![\#\mathsf{u}]\!]\rho k &=& k\ \langle 1,\ \#\mathsf{u}\rangle \\
\mathcal{D}[\![\#\mathsf{t}]\!]\rho k &=& k\ \langle 2,\ \#\mathsf{t}\rangle \\
\mathcal{D}[\![\#\mathsf{f}]\!]\rho k &=& k\ \langle 2,\ \#\mathsf{f}\rangle \\
\mathcal{D}[\![n]\!]\rho k &=& k\ \langle 3,\ n\rangle \\
\mathcal{D}[\![\lambda x\ e]\!]\rho k &=& k\ \langle 5, \lambda x' \lambda k'\ \mathcal{D}[\![e]\!](\mathit{extend\text{-}env}\ \rho\ \text{“}x\text{”}\ x')\ k'\rangle \\
\mathcal{D}[\![e_0\ e_1]\!]\rho k &=& \mathcal{D}[\![e_0]\!]\rho(\mathit{check\text{-}fn}\ (\lambda f\ \mathcal{D}[\![e_1]\!]\rho(\lambda v\ f\ v\ k))) \\
\mathcal{D}[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!] &=& \mathcal{D}[\![e_0]\!]\rho(\mathit{check\text{-}bool}(\lambda b\ \mathsf{if}\ b\ \mathsf{then}\ \mathcal{D}[\![e_1]\!]\rho k\ \mathsf{else}\ \mathcal{D}[\![e_2]\!]\rho k)) \\
\mathcal{D}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho k &=& \mathcal{D}[\![e_1]\!]\rho(\lambda v\ \mathcal{D}[\![e_2]\!](\mathit{extend\text{-}env}\ \rho\ \text{“}x\text{”}\ v)\ k) \\
\mathcal{D}[\![\langle e_1,\ e_2\rangle]\!] &=& \mathcal{D}[\![e_1]\!]\rho(\lambda v_1\ \mathcal{D}[\![e_2]\!]\rho\ (\lambda v_2\ k\langle 4,\ \langle v_1,\ v_2\rangle\rangle))) \\
\mathcal{D}[\![\mathsf{left}\ e]\!]\rho k &=& \mathcal{D}[\![e]\!]\rho\ (\mathit{check\text{-}pair}\ (\lambda p\ k\ (\mathsf{left}\ p)))
\end{array}$$

plus rules for right, rec, etc.

[2]For example, see *Compiling with Continuations*, by Andrew Appel

Recall that ($\textit{extend-env}\ \rho$ "x" x') means the environment ρ is extended with a new variable "x" bound to x'. Notice that this gives us a more compact translation of uF.

3 $\mathsf{uF}_{\mathsf{CBN}} \rightarrow \mathsf{uF}$

With only a couple of small changes, we can turn the translation above into a definitional translation for a call-by-name version of uF. If evaluation is lazy, variables can't simply return their values. Instead, a variable is now bound to a function that expects a continuation and sends a value to it : $\rho x = (\lambda k\ \ldots)$. Argument passing behaves similarly. We only need to change the following rules from above:

$$\begin{aligned}
\mathcal{D}[\![x]\!]\rho k &= \rho\ (\text{“}x\text{”})\ k \\
\mathcal{D}[\![e_0\ e_1]\!]\rho k &= \mathcal{D}[\![e_0]\!]\rho(\textit{check-fn}\ (\lambda f\ f\ (\lambda k'\ \mathcal{D}[\![e_1]\!]\rho k')\ k)) \\
\mathcal{D}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] &= \mathcal{D}[\![e_2]\!](\textit{extend-env}\ \rho\ \text{“}x\text{”}\ (\lambda k'\ \mathcal{D}[\![e_1]\!]\rho k'))\ k
\end{aligned}$$

4 $\mathsf{uF}_{!}$ to uF with error checking

Now, the translation $\mathcal{D}[\![e]\!]\rho k\sigma$ sends the result of e in environment ρ with store σ to continuation k. The continuation must also accept a new store σ', so it has the form $k = \lambda v \lambda \sigma\ \ldots$. Recall that σ maps locations to values. Whenever σ is unmodified, as in the first two rules, the translation is identical to uF $\rightarrow$ uF after performing η-reduction on σ.

$$\begin{aligned}
\mathcal{D}[\![x]\!]\rho k\sigma &= k\ (\rho\ \text{“}x\text{”})\ \sigma \\
\mathcal{D}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho k &= \mathcal{D}[\![e_1]\!]\rho(\lambda v \lambda \sigma'\ \mathcal{D}[\![e_2]\!](\textit{extend-env}\ \rho\ \text{“}x\text{”}\ v)\ k) \\
\mathcal{D}[\![!e]\!]\rho k &= \mathcal{D}[\![e]\!]\rho(\textit{check-location}\ (\lambda l \lambda \sigma\ k\ (\sigma(l))\ \sigma)) \\
\mathcal{D}[\![\mathsf{ref}\ e]\!]\rho k &= \mathcal{D}[\![e]\!]\rho(\lambda v \lambda \sigma\ \mathsf{let}\ l = (\textit{malloc}\ \sigma)\ \mathsf{in}\ k\ \langle 6,\ l\rangle\ (\textit{update-store}\ \sigma\ l\ v)) \\
\mathcal{D}[\![e_1 ::= e_2]\!]\rho k &= \mathcal{D}[\![e_1]\!]\rho(\textit{check-location}\ (\lambda l\ \mathcal{D}[\![e_2]\!]\rho\ (\lambda v \lambda \sigma\ k\ \langle 1,\ \#\mathsf{u}\rangle\ (\textit{update-store}\ \sigma\ l\ v))))
\end{aligned}$$

Through the magic of η-reduction, we only need to add new rules for the new constructs added in $\mathrm{uF}_!$!