Scribe: Adam Barth and Stan Dziobiak Lecturer: Radu Rugina

1 Denotational Semantics for $F_{\mathbf{ref}}$

We express the denotational semantics for statements that manipulate references (i.e. allocations $\mathsf{ref}\ e$, dereferences $!e$, and updates $e_0 := e_1$) and imperative-style statement sequences $e_0; e_1$ as follows:

$$\begin{aligned}
\mathcal{C}[\![\mathsf{ref}\ e]\!]\rho\sigma &= \mathit{ecase}\ \ \mathcal{C}[\![e]\!]\rho\sigma\ \ \mathit{of}\ \ (\mathit{Value}(v), \mathit{State}(\sigma')) \Rightarrow (l, \sigma'[l \to v]) \quad (\text{where } l \notin \mathit{dom}(\sigma')\ ) \\
\mathcal{C}[\![!e]\!]\rho\sigma &= \mathit{ecase}\ \ \mathcal{C}[\![e]\!]\rho\sigma\ \ \mathit{of}\ \ (\mathit{Location}(l), \mathit{State}(\sigma')) \Rightarrow (\sigma'(l), \sigma') \\
\mathcal{C}[\![e_0 := e_1]\!]\rho\sigma &= \mathit{ecase}\ \ \mathcal{C}[\![e_0]\!]\rho\sigma\ \ \mathit{of}\ \ (\mathit{Location}(l), \mathit{State}(\sigma')) \\
&\quad \Rightarrow \mathit{ecase}\ \ \mathcal{C}[\![e_1]\!]\rho\sigma'\ \ \mathit{of}\ \ (\mathit{Value}(v), \mathit{State}(\sigma'')) \Rightarrow (\mathit{unit}, \sigma''[l \to v]) \\
\mathcal{C}[\![e_0; e_1]\!]\rho\sigma &= \mathit{ecase}\ \ \mathcal{C}[\![e_0]\!]\rho\sigma\ \ \mathit{of}\ \ (\mathit{Value}(v_0), \mathit{State}(\sigma')) \\
&\quad \Rightarrow \mathit{ecase}\ \ \mathcal{C}[\![e_1]\!]\rho\sigma'\ \ \mathit{of}\ \ (\mathit{Value}(v_1), \mathit{State}(\sigma'')) \Rightarrow (\mathit{unit}, \sigma'')
\end{aligned}$$

2 Translation of $F_{\mathbf{ref}}$ to F

We have given formal semantics for different languages, but we have not seen how to translate a program written in one language into an equivalent program in another language. The translation of a language L into another language L' has two main benefits: 1) it automatically provides a semantics of L, given an existing semantics of L', and 2) it provides more insights about the differences between the two languages, showing how difficult it is to express the constructs in one language using constructs in the other language.

In this lecture, we discuss the translation from F_{ref} to F. We assume the following constructs in F:

- A representation for *locations*, which can be dynamically created;
- A data structure representing a program *state*, i.e. a mapping of locations to their values;
- A set of primitive operations on states and locations:
  - $\mathsf{alloc\ s = l}$, where $\mathsf{l} \notin \mathsf{s}$
  - $\mathsf{update\ s\ l\ v}$
  - $\mathsf{lookup\ s\ l = v}$
- A representation of the unit value.

These constructs must be such that locations and the unit value are different that any other value in F. We formally define the translation using a function $\mathcal{T}[\![\langle F_{\text{ref}}\ \mathsf{syntax}\rangle]\!]\ \mathsf{s} = \langle F\ \mathsf{syntax}\rangle$, which translates a piece of F_{ref} syntax into a piece of F syntax, using a variable s (representing the current state) as parameter. We define the translation function $\mathcal{T}[\![\cdot]\!]$ by induction on the structure of expressions in F. The idea is to translate each value in F_{ref} into an explicit pair in F, consisting of the value and the current state. For instance, the translation for numbers and variables is:

$$\begin{aligned}
\mathcal{T}[\![\mathsf{n}]\!]\mathsf{s} &= \langle \mathsf{n,s} \rangle \\
\mathcal{T}[\![\mathsf{x}]\!]\mathsf{s} &= \langle \mathsf{x,s} \rangle
\end{aligned}$$

Most of the F_{ref} constructs are also constructs in F. Hence, their translation is straightforward. However, one must take into account that statements in F_{ref} may have side-effects and may modify the state. The translation must keep track of all possible state changes. We only present the rule for the if statement; the other rules are similar.

$$\begin{aligned}\mathcal{T}\llbracket \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \rrbracket \text{s} \quad = \quad & \text{let p} = \mathcal{T}\llbracket e_0 \rrbracket \text{s in} \\ & \quad \text{let b} = \text{fst p in} \\ & \quad\quad \text{let s'} = \text{snd p in} \\ & \quad\quad\quad \text{if b then } \mathcal{T}\llbracket e_1 \rrbracket \text{s' else } \mathcal{T}\llbracket e_2 \rrbracket \text{s'}\end{aligned}$$

The interesting part is the translation of the constructs that manipulate references, which are not available in F. We define the translation for these statements as follows:

$$\begin{aligned}\mathcal{T}\llbracket \text{ref } e \rrbracket \text{s} \quad = \quad & \text{let p} = \mathcal{T}\llbracket e \rrbracket \text{s in} \\ & \quad \text{let v} = \text{fst p in} \\ & \quad\quad \text{let s'} = \text{snd p in} \\ & \quad\quad\quad \text{let l} = \text{alloc s' in} \\ & \quad\quad\quad\quad \langle \text{l, update s' l v} \rangle\end{aligned}$$

$$\begin{aligned}\mathcal{T}\llbracket !e \rrbracket \text{s} \quad = \quad & \text{let p} = \mathcal{T}\llbracket e \rrbracket \text{s in} \\ & \quad \text{let l} = \text{fst p in} \\ & \quad\quad \text{let s'} = \text{snd p in} \\ & \quad\quad\quad \langle \text{lookup s' l, s'} \rangle\end{aligned}$$

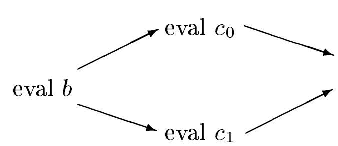
$$\begin{aligned}\mathcal{T}\llbracket e_0 := e_1 \rrbracket \text{s} \quad = \quad & \text{let p}_0 = \mathcal{T}\llbracket e_0 \rrbracket \text{s in} \\ & \quad \text{let l} = \text{fst p}_0 \text{ in} \\ & \quad\quad \text{let s'} = \text{snd p}_0 \text{ in} \\ & \quad\quad\quad \text{let p}_1 = \mathcal{T}\llbracket e_1 \rrbracket \text{s' in} \\ & \quad\quad\quad\quad \text{let v} = \text{fst p}_1 \text{ in} \\ & \quad\quad\quad\quad\quad \text{let s''} = \text{snd p}_1 \text{ in} \\ & \quad\quad\quad\quad\quad\quad \langle \text{unit, update s'' l v} \rangle\end{aligned}$$

$$\begin{aligned}\mathcal{T}\llbracket e_0 ; e_1 \rrbracket \text{s} \quad = \quad & \text{let p}_0 = \mathcal{T}\llbracket e_0 \rrbracket \text{s in} \\ & \quad \text{let s'} = \text{snd p}_0 \text{ in} \\ & \quad\quad \text{let p}_1 = \mathcal{T}\llbracket e_1 \rrbracket \text{s' in} \\ & \quad\quad\quad \langle \text{unit, snd p}_1 \rangle\end{aligned}$$

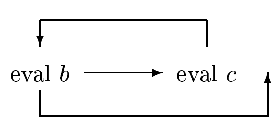
3 Continuation-Passing Style Semantics (CPS)

So far we have seen semantics for constructs with *structured control flow*. For each of these constructs, the control flow follows a regular pattern. Here are some examples of constructs with structured flow. The arrows in the figures represent the control flow.

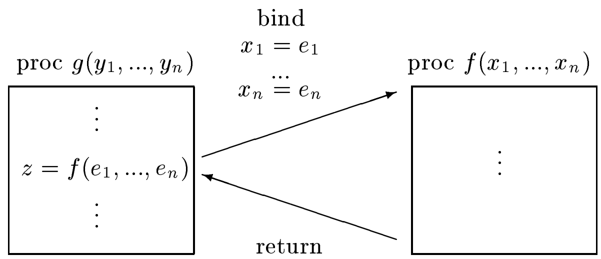
- Conditional statement: if b then e_0 else e_1. After the evaluation of the condition, the control flows to one of the branches, depending on the test condition. Regardless of the branch being executed, the control after the branch is transferred to the program point after the if statement:



- While statement: while b do c. After evaluating the test condition, the control may flow either to the loop body, or to the point after the loop. If the loop body is evaluated, the control flows back to the evaluation of the test condition.



- Function calls. At each call site, formal parameters of the invoked procedure are bound to the actual values at the call site. The the control is transferred at the beginning of the callee's body. After the evaluation of the invoked procedure, the control is transferred back to the caller procedure, at the point after the call statement.



Nesting statements with structured control flow gives rise to a hierarchy of control flow. Therefore, we also refer to structured control flow as *hierarchical control flow*.

Let's now look at constructs with unstructured forms of control flow. These include the following:

- Unstructured jumps: goto
- Exceptions: throw, catch
- Loop exits: break, continue

For each of these constructs, the control may flow outside the construct, to a point not immediately enclosing the construct. Therefore, we also refer to unstructured forms of control flow as *non-local control flow*. We would like to write the semantics of constructs with non-local flow of control. It turns out that it is difficult to express their semantics with our current formal tools. We therefore introduce a new kind of semantics called Continuation Passing Style (CPS) semantics.

The main idea in CPS is that of a *continuation*. A continuation is a piece of computation which has not yet been evaluated. The evaluation of an expression e takes as arguments an environment (as before), but also a continuation, which represents the computation that the program performs after evaluating e. Therefore the CPS denotation of an expression e has the following signature:

$$\mathcal{C}[\![e]\!] : \mathit{Env} \to \mathit{Cont} \to \mathit{Result}$$

The denotation $\mathcal{C}[\![e]\!]\rho\kappa$ first evaluates e in the environment ρ, then calls the continuation κ (which is an argument of $\mathcal{C}[\![e]\!]$) with the result of the evaluation of e as the argument of κ.

Example

We next present a simple example which gives more intuition behind continuations. Consider the following conditional statement:

$$\mathsf{if}\ (x < 0)\ \mathsf{then}\ 0\ \mathsf{else}\ f(x-1)$$

We can regard the execution of this statement as consisting of two steps: first evaluate the test condition $x < 0$; then execute one of the branches depending on the result of the test condition. To emphasize these two steps, we rewrite this program as follows:

$$\underbrace{\lambda b.\,\mathsf{if}\ b\ \mathsf{then}\ 0\ \mathsf{else}\ f(x-1)}_{\text{Continuation}}\ \underbrace{(x < 0)}_{\text{Expr}}$$

This form of the program explicitly shows that we first evaluate the test $x < 0$, then we pass the result of this evaluation to a continuation κ representing "the rest of the computation": $\kappa = \lambda b.\,\mathsf{if}\ b\ \mathsf{then}\ 0\ \mathsf{else}\ f(x-1)$. Therefore, we can express the CPS semantics of this statement as follows:

$$\begin{aligned} & \mathcal{C}[\![x < 0]\!]\,\rho\,\kappa \\ = \ & \mathcal{C}[\![x < 0]\!]\,\rho\,(\lambda b.\,\mathsf{if}\ b\ \mathsf{then}\ 0\ \mathsf{else}\ f(x-1)) \end{aligned}$$