CS611 Lecture 28 **Translational Semantics** 7 November 2003

Scribe: Maya Haridasan Lecturer: Radu Rugina

1 Denotational Semantics for FUN! (Cont.)

In the last lecture we extended our applicative language **FUN** to support dynamic allocation and imperative features. We did this by introducing statements that manipulate references, and by allowing statement sequences that just return the value returned by the last statement of the sequence. The modified syntax for our newly defined language **FUN!** is:

$$e \quad ::= \quad \ldots \ \mid \ \textsf{ref}\ e \ \mid \ !e \ \mid \ e_1 := e_2 \ \mid \ e_1; e_2 \ \mid \ \textsf{unit}$$

Recall that we defined a new value *unit* to be returned by expressions that are just evaluated for their side-effects (assignments and sequences).

Last time we defined the operational semantics for **FUN!** and part of its denotational semantics. Recall that the denotation function $\mathcal{C}[\![e]\!]$ takes an environment variable and a state variable as arguments and returns something of type *Result*, which is a pair of a value and a state.

$$\mathcal{C}[\![e]\!] : Env \rightarrow State \rightarrow Result$$

Today we will present the denotational semantics for the rest of the statements, namely statements that manipulate references, imperative-style statement sequences and unit:

$$\begin{aligned}
\mathcal{C}[\![\textsf{ref}\ e]\!]\rho\sigma &= \textsf{ecase}\ \mathcal{C}[\![e]\!]\rho\sigma\ \textsf{of}\ (Value(v), State(\sigma')) \\
&\quad \Rightarrow (l, \sigma' \cup \{l \rightarrow v\}), \quad l \notin dom(\sigma) \\
\\
\mathcal{C}[\![!e]\!]\rho\sigma &= \textsf{ecase}\ \mathcal{C}[\![e]\!]\rho\sigma\ \textsf{of}\ (Location(l), State(\sigma')) \\
&\quad \Rightarrow (\sigma'(l), \sigma') \\
\\
\mathcal{C}[\![e_1 := e_2]\!]\rho\sigma &= \textsf{ecase}\ \mathcal{C}[\![e_1]\!]\rho\sigma\ \textsf{of}\ (Location(l), State(\sigma')) \\
&\quad \Rightarrow \textsf{ecase}\ \mathcal{C}[\![e_2]\!]\rho\sigma'\ \textsf{of}\ (Value(v), State(\sigma'')) \\
&\quad\quad \Rightarrow (unit, \sigma''[l \mapsto v]) \\
\\
\mathcal{C}[\![e_1; e_2]\!]\rho\sigma &= \textsf{ecase}\ \mathcal{C}[\![e_1]\!]\rho\sigma\ \textsf{of}\ (Value(v_1), State(\sigma')) \\
&\quad \Rightarrow \textsf{ecase}\ \mathcal{C}[\![e_2]\!]\rho\sigma'\ \textsf{of}\ (Value(v_2), State(\sigma'')) \\
&\quad\quad \Rightarrow (v_2, \sigma'') \\
\\
\mathcal{C}[\![\textsf{unit}]\!]\rho\sigma &= (unit, \sigma)
\end{aligned}$$

Notice that in the denotational semantics of unit, on the left hand side we have unit being used as a piece of syntax, while in the right hand side we have *unit* being used as part of our domain. This is similar to what we had in the denotational semantics of truth values.

2 Translational Semantics

We will now see how to give a semantics for a new language *L'* using the semantics of a known language *L*. Translational semantics allows us to translate a program written in one language into an equivalent program in another language. Therefore, the translation plus the semantics of the underlying language *L'* provide a semantics for language *L*. This is an alternative way to give the semantics of a language based on the semantics of another, existing language. Translational semantics is also useful to show how much syntactic sugar a language adds relative to another language.

As an example, we will now present the semantics for **FUN!** using translational semantics relative to **FUN**. A translation will be formally defined as a function that maps a piece of **FUN!** syntax into a piece of **FUN** syntax. It will require an extra argument s which will be a variable representing the current state.

$$\mathcal{T}[\![\langle \textbf{FUN!}\ \mathsf{syntax}\rangle, s]\!] = \langle \textbf{FUN}\ \mathsf{syntax}, s'\rangle$$

Most of the **FUN!** constructs are also constructs in **FUN**, but it is important to take into account that statements in **FUN!** may have side-effects and therefore modify the state. We will present the translation for some of the constructs:

$$\begin{array}{rcl}
\mathcal{T}[\![n, s]\!] & = & \langle n, s\rangle \\[1em]
\mathcal{T}[\![x, s]\!] & = & \langle x, s\rangle \\[1em]
\mathcal{T}[\![\textbf{if}\, e_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2, s]\!] & = & \mathsf{let}\ p = \mathcal{T}[\![e_0, s]\!]\ \mathsf{in} \\
& & \quad \mathsf{let}\ b = \mathsf{fst}\ p\ \mathsf{in} \\
& & \quad\quad \mathsf{let}\ s' = \mathsf{snd}\ p\ \mathsf{in} \\
& & \quad\quad\quad \mathsf{if}\ b\ \mathsf{then}\ \mathcal{T}[\![e_1, s']\!] \\
& & \quad\quad\quad\quad\quad \mathsf{else}\ \mathcal{T}[\![e_2, s']\!]
\end{array}$$

One of the issues that arise when trying to translate language **FUN!** into **FUN** is: How do we represent locations using **FUN**? For now we will assume that we have a way to represent the value *unit* and that we have the following functions:

- $\mathsf{alloc}\ s = l$
- $\mathsf{update}\ s\ l\ v = s'$
- $\mathsf{lookup}\ s\ l = v$

These functions are responsible for dealing with all errors arising from erroneous arguments which should represent locations. We will come back to the implementation details of these functions later.

We can now define the translation for the constructs that manipulate references and for the sequencing construct:

$$\begin{array}{rcl}
\mathcal{T}[\![\mathsf{ref}\ e, s]\!] & = & \mathsf{let}\ p = \mathcal{T}[\![e, s]\!]\ \mathsf{in} \\
& & \quad \mathsf{let}\ v = \mathsf{fst}\ p\ \mathsf{in} \\
& & \quad\quad \mathsf{let}\ s' = \mathsf{snd}\ p\ \mathsf{in} \\
& & \quad\quad\quad \mathsf{let}\ l = \mathsf{alloc}\ s'\ \mathsf{in} \\
& & \quad\quad\quad\quad \langle l,\ \mathsf{update}\ s'\ l\ v\rangle
\end{array}$$

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

$$
\begin{aligned}
\mathcal{T}[\![e_1 := e_2, s]\!] \quad = \quad & \mathsf{let}\ p_1 = \mathcal{T}[\![e_1, s]\!]\ \mathsf{in} \\
& \quad \mathsf{let}\ l = \mathsf{fst}\ p_1\ \mathsf{in} \\
& \quad\quad \mathsf{let}\ s' = \mathsf{snd}\ p_1\ \mathsf{in} \\
& \quad\quad\quad \mathsf{let}\ p_2 = \mathcal{T}[\![e_2, s]\!]\ \mathsf{in} \\
& \quad\quad\quad\quad \mathsf{let}\ v = \mathsf{fst}\ p_2\ \mathsf{in} \\
& \quad\quad\quad\quad\quad \langle \mathsf{unit},\ \mathsf{update}\ (\mathsf{snd}\ p_2)\ l\ v \rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}[\![e_1; e_2, s]\!] \quad = \quad & \mathsf{let}\ p = \mathcal{T}[\![e_1, s]\!]\ \mathsf{in} \\
& \quad \mathsf{let}\ s' = \mathsf{snd}\ p\ \mathsf{in} \\
& \quad\quad \mathcal{T}[\![e_2, s']\!]
\end{aligned}
$$

This is an alternative way of expressing the semantics of a language. We should now go back and talk about the implementation of functions alloc, update and lookup previously defined. These functions act upon a heap, and therefore we need to be able to represent a heap using **FUN**.

A heap is just a collection of locations, so we can represent one using a list. We can represent a list in **FUN** using nested pairs:

$$
heap = \langle v_0, \langle v_1, ..., \langle v_n, \langle \mathsf{true}, \mathsf{true} \rangle \rangle \rangle \rangle
$$

Notice that the pair $\langle \mathsf{true}, \mathsf{true} \rangle$ is used as a terminator and represents an empty heap. Addresses in this representation of a heap are just integers starting from zero, and v_i contains the value at address i. We can then easily implement functions alloc, update, and lookup to perform the desired operations over this representation of heap structures.

One problem with this approach is that we are using the same domain for integers and heap locations. We have two domains that we want to distinguish. What we want is to take the disjoint sum of these two domains with common values. We could represent such a disjoint sum using pairs, where the first value is a flag that is either true or false, and the second is an integer. If the flag is true, the integer indeed represents an integer value; if it is false, it represents a heap location. But this changes the representation of integers. Hence, operations such as addition or comparison of integer must account for the new representation. We would need to define a translation for all constructs that manipulate integers, to extract their values from the pairs that they are wrapped in.

3 Control Flow Constructs

All constructs we have seen so far have structured control flow (also known as hierarchical control flow or local control flow). For each of these constructs, the control flow follows a regular pattern.

Let us consider the while loops, for example. In while loops, after evaluating the test condition, control flows to the loop body or to the point after the loop. In conditional statements the control flows to one of the two branches, and after the branch it is transferred to the program point after the conditional statement.

Another special case is the control flow pattern of function calls. In this case the control is transferred from the caller procedure to the beginning of the called procedure body. After the evaluation of the invoked procedure, control is transferred back to the point after the call statement in the caller procedure.

We will now start to think about unstructured control flow constructs, which include the following:

- break, continue (loop exits - occur in loop constructs)
- goto, jump (control flow can go to any label or instruction in the program)
- throw, catch (exception handling - control is passed to a completely different function)

In all these constructs, control is transferred to somewhere outside the construct. Therefore, we refer to this kind of control flow as non-local control flow.