COMP313-08A
Programming Languages
Notes on Standard Semantics

Steve Reeves, heavily based on Mike Gordon's book—these notes are brief, so you need to take further notes at the lectures and read the book.

May 13, 2008

1 Standard Semantics

For "real" languages we need more heavy-duty apparatus than that used for TINY. Standard semantics is intended for real languages and it brings in ideas like:

1. states are not directly transformed (by expressions and commands) but indirectly via $continuations$;
2. identifiers are not bound directly to values but via the intermediary idea of $locations$ or variables, and using this idea we can handle sharing and aliasing.

2 Continuations

Read section 5.1 of Gordon for a fuller, more detailed, account.

These model "the rest of the program".

We make the denotations of constructs depend on the rest of the program (the *continuation* of the program)—this is where each construct will pass its result.

Usually, where no error occurs, it is the code following the current construct that takes control. This is the *normal continuation*. If an error occurs (or a jump...) then control passes to some other continuation.

A continuation is a function which takes “the answer so far” and, by being “the rest of the program”, transforms this into the “final answer”.

For example, in TINY $I := E; C$ is such that the answer so far after doing $I := E$ (which is some state) is passed to the rest of the program, i.e. C, and the final answer is the state got from doing C on the intermediate state.

The answer so far after doing E, i.e. what is passed to $I := \ ; C$ is a value (E’s value) together with a state (the state after doing E).

The first kind of continuation, modelling the rest of the program after a command, are *command continuations* from the domain $Cont$ where

$$Cont = State \rightarrow [State + \{error\}]$$

The second kind of continuation, modelling the rest fo the program after an expression, are *expression continuations* from the domain $Econt$ where

$$Econt = Value \rightarrow State \rightarrow [State + \{error\}]$$

Note that

$$Econt = Value \rightarrow Cont$$

Now the semantic domains for TINY, which were

$$E : Exp \rightarrow State \rightarrow [[Value \times State] + \{error\}]$$

and

$$C : Com \rightarrow State \rightarrow [State + \{error\}]$$

become:

$$E' :: Exp \rightarrow Econt \rightarrow State \rightarrow [State + \{error\}]$$

which is

$$E' :: Exp \rightarrow Econt \rightarrow Cont$$

and

$$C' : Com \rightarrow Cont \rightarrow State \rightarrow [State + \{error\}]$$

which is

$$C' : Com \rightarrow Cont \rightarrow Cont$$

In general we have:

$$E'[\![E]\!]\ k\ s = \begin{cases} k\ v\ s', & \text{where } E \text{ has value } v \text{ and transforms } s \text{ to } s' \\ \\ error, & \text{otherwise} \end{cases} \tag{1}$$

and

$$C'[\![C]\!]\ c\ s = \begin{cases} c\ s', & \text{where } C \text{ transforms } s \text{ to } s' \\ \\ error, & \text{otherwise} \end{cases} \tag{2}$$

For example

$$C'[\![C_1; C_2]\!]\ c\ s = C'[\![C_1]\!](C'[\![C_2]\!]\ c)s$$

2.1 Things to do with output

Three problems regarding output with what we have said so far:

1. It is not natural to say that the result of running a program is the whole final state; we usually think of just the output component as being "the result";
2. by making output part of the state then we have made the output just as accessible to programs as the memory or the input are, but this is not natural either since once something is output it is no longer accessible to the program;
3. we have assumed that output is a finite sequence of values, but for a non-terminating program this is not true.

So, we remove output from the state and change the type of continuations. We get:

$$\begin{array}{ll} State & = Memory \times Input \\ Memory & = Ide \to [Value + \{unbound\}] \\ Input & = Value^* \\ Value & = Num + Bool \\ Cont & = State \to Ans \\ Econt & = Value \to Cont \\ Ans & = \{error, stop\} + [Value \times Ans] \end{array}$$

The domain Ans of final answers is, note, recursive, so can express the results of a non-terminating program. Also note that a final answer is either a pair consisting of either $error$ or $stop$ and a finite sequence of answers, or it is an infinite sequence of answers (with no $error$ or $stop$).

This change means that the clause giving the output command its meaning now has to be:

$$C[\![output\ E]\!]\ c = E[\![E]\!]\lambda v, s.(v, cs)$$

so E is evaluated to get a value v and a new state s and the final answer is v followed by whatever the rest of the program c gives starting with state s. This means that if the program crashes after outputting some values then those values are not lost (which they would be using the original way of doing things). To see this, try the command $output\ 0$ with the “always an error” continuation $\lambda s.error$ on the new and old clauses for output.

If the program is the (probably composite) command C then $C[\![C]\!](\lambda s.stop)\ s$ will eventually output $stop$ if the program terminates normally when started in state s. If it never terminates then $stop$ is never output.

The domain Ans is language dependent.

3 Environments, locations and stores and scope

Often in a language identifiers are not bound to values but to $variables$ or $locations$.

This allows *sharing* or *aliasing*.

Sharing happens if we declare a procedure (method) by

$$procedure\ P(var\ x, y\ :\ int) < statements >$$

and then call $P(z, z)$ then inside the body (inside $< statements >$) both x and y will share the variable denoted by z.

So, if sharing can happen we have to model it. First we introduce $locations$ and then $stores$. A store associates locations with values:

$$Store = Loc \rightarrow [Sv + \{unused\}]$$

and here Sv is a domain of storable values. These are language dependent. What can be stored in a language is one dimension used to classify languages. It is conventional to use ss to range over stores, ιs to range over locations and vs to range over storable values.

To model identifiers and their denotations we have the idea of $environment$. These associate identifiers with locations:

$$Env = Ide \rightarrow [Dv + \{unbound\}]$$

Again Dv, the denotable values, are language dependent. They form another dimension for classifying languages. It is conventional to use rs to range over environments and ds to range over denotable values. usually $Dv = Loc$ but other things may be denotable too—constants, arrays, records, procedures etc.

We also use the domain of expressible values Ev and use es to range over this.

Next, scope. Commands change the $contents$ of locations, but not the way identifiers bind to locations. Consider:

```
begin  integer x;
       .
       .
       x := 1;
       .
       .
       x := 2;
       .
       .
end
```

where x denotes a fixed $location$, and then consider:

```
begin  integer x;
       integer y;
       .
       .
       x := 1;
       .
       begin    integer x;
                .
                .
                x := 2;
                .
                .
       end
       .
       .
       .
end
```

The x occurrences in the inner and outer blocks denote different locations. Since y is not declared in the inner block it denotes the same location throughout. The *scope* of a declaration is where it holds. The inner block is a *hole* in the scope of the outer declaration of x, but not of y.

In standard semantics:

- commands change the store but *not* the environment;
- declarations change the environment (and perhaps the store—new store may be used in, e.g., $var\ I = E$).

We introduce a new syntactic category Dec of declarations, ranged over by D as:

$$D ::= const\ I = E \mid var\ I = E \mid proc\ I(I_1); C \mid fun\ I(I_1); E \mid D_1; D_2$$

and each declaration generates a new little piece of environment: so $const\ I = E$ generates e/I where e is E's value, and $var\ I = E$ generates ι/I where ι is a new location updated with E's value.

4 Continuation machinery and useful functions

Command continuations

$$Cc = Store \to Ans$$

Ans is the domain of final answers and is language dependent but always contains a special $error$ element. We use cs to range over Cc.

Expression continuations

$$Ec = Ev \to Store \to Ans, \text{ so } Ec = Ev \to Cc$$

We use ks to range over Ec.

Declaration continuations

$$Dc = Env \to Store \to Ans, \text{ so } Dc = Env \to Cc$$

We use us to range over Dc.

In standard semantics we have the following semantic functions:

$$\begin{array}{l} E : Exp \to Env \to Ec \to Store \to Ans \\ C : Com \to Env \to Cc \to Store \to Ans \\ D : Dec \to Env \to Dc \to Store \to Ans \end{array}$$

If there are no errors, jump etc. (i.e. normal flow of control is not disturbed) then:

$E[\![E]\!]\ r\ k\ s$	$= k\ e\ s'$	where e is E's value in environment r and store s and s' is the store after e's evaluation
$C[\![C]\!]\ r\ c\ s$	$= c\ s'$	where s' is the store resulting from executing C in environment r and store s
$D[\![D]\!]\ r\ u\ s$	$= u\ r'\ s'$	where r' is the environment consisting of the bindings specified in D (when evaluated with respect to r and s) and s' is the store resulting from D's evaluation

For example, on our (coming soon) new language SMALL we have:

$$E[\![0]\!]\ r\ k\ s\ = k\ 0\ s$$

$$C[\![C_1; C_2]\!]\ r\ c\ s\ = C[\![C_1]\!] r(\lambda s'.C[\![C_2]\!]\ r\ c\ s')\ s$$

C_1 is executed in environment r and store s to get a store s' which is passed to the continuation $\lambda s'.C[\![C_2]\!]\ r\ c\ s'$ which executes C_2 in the same environment but in store s' and then finally sends the resulting store on to c. Note that since commands do not change environments, the same environment r is passed to both commands.

$$D[\![const\ I = E]\!]\ r\ u\ s = E[\![E]\!]\ r\ (\lambda e, s'.u(e/I)\ s')\ s$$

Here e's value e is bound to I to form the little environment e/I which is passed on to u together with store s' resulting from E's evaluation.

We'll see the rest of the clauses in chapter 6 (handed out). First, some useful continuation transforming functions:

1. $cont : Ec \rightarrow Ec$

   $cont\ k\ e\ s$ checks that e is a location and then looks up its contents in the store s and passed the results, together with s, to k. If e is not a location or unused then $cont\ k\ e\ s = error$

   $cont\ k\ e\ s = isLoc\ e \rightarrow (s\ e = unused \rightarrow error, k(e\ s)s),\ error$

   which we can also (as is conventional) write:

   $cont = \lambda\ k, e, s.isLoc\ e \rightarrow (s\ e = unused \rightarrow error, k(e\ s)s),\ error$

2. $update : Loc \rightarrow Cc \rightarrow Ec$

   $update\ \iota\ c\ e\ s$ stores e at location ι in store s and passes the resulting store to c. If e is not storable then we get an error.

   $update = \lambda\ \iota, c, e, s.isSv\ e \rightarrow c(s[e/\iota]), error$

3. $ref : Ec \rightarrow Ec$

   $ref\ k\ e\ s$ gets un unused location form s, updates it with e and passes it, and the updated store, to c. If there are no unused locations available (the store is full) then we get an error.

   $ref = \lambda\ k, e, s.new\ s = error \rightarrow error, update\ (new\ s)(k(new\ s))e\ s$

4. $deref : Ec \rightarrow Ec$

   $deref\ k\ e\ s$ tests to see if e is a location and if it is it passes the contents according to s, together with s, to k, If e is not a location then e and s are passed to k.

   $deref = \lambda k, e, s.isLoc\ e \rightarrow cont\ k\ e\ s, k\ e\ s$

5. $err : Cc$

   err is the error continuation, i.e. it always says that the rest of the program is ignored and we have an error.

   $err = \lambda\ s.error$

5 Assignment and L and R values

In $I_1 := I_2$ what happens, since both identifiers denote locations?

Either (1) the location referred to by I_2 is stored in the location referred to by I_1 or (2) the $contents$ of the location referred to by I_2 is stored in the location referred to by I_1.

(2) is the usual meaning for most programming languages. We say that the right-hand sides of assignments are $dereferenced$, i.e. have their values looked up in a store if they are a location.

This gives us the usual semantics for assignment. Formally we have:

$$\begin{aligned} C[\![I := E]\!]\, r\, c\, s \;=\;& E[\![I]\!]\, r\, k_1\, s \\ & where\ k_1 = \lambda e_1, s_1. isLoc\, e_1 \to E[\![E]\!]\, r\, k_2\, s_1, error \\ & \qquad where\ k_2 = \lambda e_2, s_2. deref\ k_3\, e_2\, s_2 \\ & \qquad\qquad where\ k_3 = \lambda e_3, s_3. update\ e_1\, c\, e_3\, s_3 \end{aligned}$$

Here I is evaluated and the result is passed to k_1, which checks that it has been passed a location and then evaluates E, passing on its result to k_2. k_2 dereferences the value of E if necessary, getting a storable value, and passes the result to k_3. This then updates the location with the value, and passes control on to c.

(Note...A piece of λ-calculus machinery: the expression $\lambda x.y\ x$ can be simplified to just y because $\lambda x.y\ x$ applied to e is $(\lambda x.y\ x)\ e$ which is $y\ e$.)

Using the simplification in the Note, we have:

$$k_3 = update\ e_1\ c$$

and

$$k_2 = deref\ k_3 = deref\ (update\ e_1\ c) = deref\ (update\ (e_1\ c))$$

and if we write an explicit function application as ; we have

$$k_2 = deref\ ;\ update\ ;\ e_1\ ;\ c$$

so

$$\begin{aligned} k_1 &= \lambda e_1. isLoc\, e_1 \to E[\![E]\!]\, r\ ;\ deref\ ;\ update\ ;\ e_1\ ;\ c, err \\ &= \lambda e_1. Loc?(\lambda\iota. E[\![E]\!]\, r\ ;\ deref\ ;\ update\ \iota\ ;\ c)\ e_1 \\ &= Loc?\ \lambda\iota. E[\![E]\!]\, r\ ;\ deref\ ;\ update\ \iota\ ;\ c \end{aligned}$$

and so finally

$$C[\![I := E]\!]\, r\, c\ = E[\![I]\!]\, r\ ;\ Loc?\ \lambda\iota. E[\![E]\!]\, r\ ;\ deref\ ;\ update\ \iota\ ;\ c$$

where $Loc?\ k\ e\ s$ passes e and s to k if e is a location, and is $error$ otherwise.