Notes from Semantics and verification of programs

Jacek Olczyk

October 2018

Part I
Notes from tutorials by Lorenzo Clemente

1 Small step semantics - continuation

1.1 Recap

- Global environments $\rho \vdash e \to e'$
- $\frac{\rho[x \to n] \vdash e \to e'}{\rho \vdash \text{ let } x = \underline{n} \text{ in } e \to \text{ let } x = \underline{n} \text{ in } e'}$

1.2 Local environments

- How do we define the semantics for 'let $x = e$ in f' expressions using local environments? More precisely, we need e to have its own environment, so that its evaluation doesn't affect the environment of f, as is the case with global environments.
- We are given the following 2 rules:
- $\frac{}{(\rho, x) \to (\rho, \rho(x))}$
- $\frac{}{(\rho, \text{ let } x = \underline{n} \text{ in } e) \to (\rho[x \to n], e)}$
- Now we need to give a rule for evaluating let expressions where a non-numeric expression is assigned to x.
- $\frac{(\rho, e) \to (\rho', e')}{(\rho, \text{ let } x = e \text{ in } f) \to ((\rho\text{? or maybe } \rho'\text{?}), \text{ let } x = e' \text{ in } f)}$
- ρ doesn't work, because then a nested let in expression can't change the value of their variables.

- Neither does ρ', because then we don't get our original environment back at the end.
- Solution: new construct
- e then $x = n$
- Now we have:
- $\frac{(\rho,e)\to(\rho',e')}{(\rho,e \text{ then } x=\underline{n})\to(\rho',e' \text{ then } x=\underline{n})}$
- $\frac{}{(\rho,\underline{m} \text{ then } x=\underline{n})\to(\rho[x\to\underline{n}],\underline{m})}$
- $\frac{}{(\rho, \text{ let } x=\underline{n} \text{ in } e)\to(\rho[x\to\underline{n}],e \text{ then } x=\rho(x))}$

2 Imperative language

Syntax

$$C ::= \text{ Skip } |X := e|C;C|\text{if } b \text{ then } c \text{ else } c|\text{while } b \text{ do } c$$

$$e ::= n|x|e+e$$

$$b :: true|false|e \leq e|\neg b|b \wedge b$$

$$E[[e]]_s \in \mathbb{Q}, B[[b]]_s \in \{true, false\}$$

$$s \in State = Var \to \mathbb{Q}$$

Configurations

$$(c, s) \in C$$

$$s \in C \text{ (final)}$$

Small step rules for C - expressions

$$\frac{}{(Skip, s) \to s}$$

$$\frac{}{(x := e, s) \to s[x \to E[[e]]_s}$$

$$\frac{(c, s) \to s'}{(c; d, s) \to (d, s')}$$

$$\frac{(c, s) \to (c', s')}{(c; d, s) \to (c'; d, s')}$$

$$\frac{B[[b]]_s = true}{(\text{if } b \text{ then } c \text{ else } d, s) \to (c, s)}$$

$$\frac{B[[b]]_s = false}{(\text{if } b \text{ then } c \text{ else } d, s) \to (d, s)}$$

$$\frac{B[[b]]_s = true}{(\text{while } b \text{ do } c, s) \rightarrow (c; \text{while } b \text{ do } c, s)}$$

$$\frac{B[[b]]_s = false}{(\text{while } b \text{ do } c, s) \rightarrow s}$$

Adding "Repeat c until b"

$$\frac{}{(\text{Repeat } c \text{ until } b, s) \rightarrow (c; \text{if } b \text{ then } Skip \text{ else Repeat } c \text{ until } b, s)}$$

3 Numbers as strings of bits

- Evaluate:
- $n ::= \$0 | \$1 | n0 | n1 | n + n$
- final configurations: numbers without "+", e.g. \$100101
- $n \rightarrow n'$
- $\frac{n \rightarrow n'}{n0 \rightarrow n'0}$
- $\frac{n \rightarrow n'}{n1 \rightarrow n'1}$
- $\frac{m \rightarrow m'}{m+n \rightarrow m'+n}$
- $\frac{n \rightarrow n'}{m+n \rightarrow m+n'}$
- $\frac{}{m0+n0 \rightarrow (m+n)0}$
- $\frac{}{m0+n1 \rightarrow (m+n)1}$
- $\frac{}{m1+n0 \rightarrow (m+n)1}$
- $\frac{}{m1+n1 \rightarrow (m+n+\$1)0}$
- Fill in the last 4
- I think we should add a rule to merge two doll

4 Next time

Add to the syntax:

- for x:=e to e do c
- do e times c
- do c while e

5 TODO przepisanie z zeszytu

6 Loop, continue and break

$$C ::= \ldots|loop\ c|continue|break$$

Last time we did small steps semantics using $c\ then\ d$ statements. Now we want big steps:

$$\frac{c, s \to \ldots}{loop\ c, s \to} \frac{}{continue, s \to} \frac{}{break, s \to \ldots}$$

We can change the set of configurations by adding to the existing set of final configurations pairs $(state, flag)$ where $flag \in \{CNT, BRK\}$, thus:

$$\frac{c, s \to s', loop\ c, s' \to s''}{loop\ c, s \to s''} \frac{c, s \to (s', CNT), loop\ c, s' \to s''}{loop\ c, s \to s''} \frac{c, s \to (s', BRK)}{loop\ c, s \to s'}$$

$$\frac{c, s \to s', (d, s') \to s''}{c; d, s \to s''}, \hat{s} \in \{s'', (s'', CNT), (s'', BRK)\} \frac{c, s \to (s', f)}{c; d, s \to (s', f)}$$

7 Expressions with side effects

The syntax is as follows:

$$C ::= Skip|x := e|c; c$$

$$e ::= x|n|e + e|c\ resultis\ e$$

Old rules:

$$\frac{}{n, s \to \underline{n}}$$

$$\frac{}{x, s \to s(x)}$$

$$\frac{e, s \to \underline{m} \quad f, s \to \underline{n}}{e + f, s \to \underline{\underline{m + n}}}$$

$$\frac{}{Skip, s \to s}$$

$$\frac{e, s \to \underline{n}}{x := e, s \to s[x \mapsto n]}$$

New rules:

$$\frac{c, s \to s' \quad e, s' \to \underline{n}}{c\ resultis\ e, s \to \underline{n}}$$

Buut, this doesn't propagate the state change from inside the expressions! To fix this, we change the meaning of $\to$ for expressions by making it go to a pair $(number, state)$. Here are the modified old rules for addition and *resultis*:

$$\frac{e, s \to \underline{m}, s' \quad f, s \to \underline{n}, s''}{e + f, s \to \underline{\underline{m + n}}, s''}$$

$$\frac{c, s \to s' \quad e, s' \to \underline{n}, s''}{c\ resultis\ e, s \to \underline{n}, s''}$$

8 Let in expressions with lazy evaluation

Previously we had 'call by value' semantics for let in expressions, now we want 'call by name' semantics, which evaluate the variable assignment only when its value is needed. In CBV, we had $\frac{}{s \vDash x \to s(x)}$. How do we write semantics for *let* in CBN?

$$\frac{s[x \mapsto e] \vDash f \to m}{s \vDash let\ x = e\ in\ f \to m}$$

$$\frac{s \vDash s(x) \to n}{s \vDash x \to \underline{n})}$$

But this is dynamic binding, the environment used is whatever was at the moment of evaluation. To get static binding, we need variables to record state alongside the expressions: $St = Var \to (Expr \times St) \cup \mathbb{Q}$. But this is not a definition, just a recursive equation! Thus, let $St_0 = \emptyset$ and $St_{i+1} = Var \to (E \times St_i \cup \mathbb{Q})$. And the whole state is defined like this: (A set of russian dolls with arbitrary nesting) $St = \bigcup_{i=0}^{\infty} St_i$. Now we can get to the rules with static binding:

$$\frac{s(x) = (e, s') \quad s' \vDash e \to n}{s \vDash x \to \underline{n}}$$

$$\frac{s[x \mapsto e, s] \vDash f \to m}{s \vDash let\ x = e\ in\ f \to m}$$

9 Tutorial 14/11

9.1 Eager vs. lazy, dynamic vs. static

9.1.1 Higher order expression

$$e ::= x|n|e + e|let\ x = e\ in\ e|\lambda x.e|e\ e$$

Where $\lambda x.e$ is λ abstraction - function definition, and $e\ e$ is function application. Now we find that the *let in* construct is redundant. How do we express its semantics using λ abstraction and application?

$$let\ x = e\ in\ f \equiv (\lambda x.f)e$$

We need to use parentheses because application has the highest priority of all expressions.

9.1.2 Call-by-value (eager) big step operational semantics.

Is there a difference between static and dynamic binding in this case? Without higher order expressions, we can't do dynamic binding, because we have no concept of expressions inside state.

Is static = dynamic in higher order?

$$let\ x = 7\ in\ let\ f = \lambda y.y + x\ in\ let\ x = 3\ in\ f\ 10$$

If we evaluate this expression with static binding, it evaluates to 17, as x gets mapped inside f to its value at the time of binding, and with dynamic binding it's 13, because x is bound to 3 at the time of application of f. To write the semantics, we introduce closure. For example, $\lambda y.y + x$ in state s evaluates to the triplet called closure $(y, y + x, s) \in Var \times Expr \times St$

Static binding with eager evaluation

	STATIC	DYNAMIC
EAGER	$Val = \mathbb{Z} \cup Var \times Expr \times St$ $St = Var \to Val$ mutually recursive	
LAZY		

Is it possible to construct sets that satisfy this recursive definition? We'll construct a family of sets for both Val and $State$ and define them as infinite unions of all sequences.

$$\begin{array}{rr} Val_0 = \emptyset & Val_{n+1} = \mathbb{Z} \cup Var \times Expr \times St_{n+1} \\ St_0 = \emptyset & St_{n+1} = Var \to Val_n \end{array}$$

Thus, $Val_1 = \mathbb{Z}, Val_2 = \mathbb{Z} \cup Var \times Expr \times (Var \to \mathbb{Z}) \dots$ Now the big step semantics:

$$\frac{}{n, s \to n} \quad \frac{}{x, s \to s(x) \in Val} \quad \frac{e, s \to \underline{m}, f, s \to \underline{n}}{e + f, s \to m + n}$$

And new ones:

$$\frac{}{\lambda x.e, s \to (x, e, s)}, \frac{(e, s) \to (x, e', s')\ (f, s) \to v\ (e', s'[x \mapsto v]) \to v'}{e\ f, s \to v'}$$

Important: since elements of Val can be either numbers or closures, then effects of our function applications can also be closures!

Dynamic binding with eager evaluation.

	STATIC	DYNAMIC
EAGER	$Val = \mathbb{Z} \cup (Var \times Expr \times St)$ $St = Var \to Val$ mutually recursive	$Val = \mathbb{Z} \cup (Var \times Expr)$ $St = Var \to Val$ not recursive anymore!
LAZY		

Now the rules:

$$\frac{}{\lambda x.e, s \to (x, e)}, \frac{(e, s) \to (x, e')\ (f, s) \to v\ (e', s[x \mapsto v]) \to v'}{e\ f, s \to v'}$$

Static binding with lazy evaluation Is lazy (call by name) even different than eager (call by value)? Suppose e is an expression that does not terminate. Find f that uses e such that its lazy semantics are different than eager.

$$(\lambda x.5)\ e$$

In lazy, the value is 5. In eager, it does not terminate. This is an example of a side effect: not pure function.

	STATIC	DYNAMIC
EAGER	$Val = \mathbb{Z} \cup (Var \times Expr \times St)$ $St = Var \to Val$ mutually recursive	$Val = \mathbb{Z} \cup (Var \times Expr)$ $St = Var \to Val$ not recursive anymore!
LAZY	$Val = \mathbb{Z} \cup (Var \times Expr \times St)$ $St = Var \to (Expr \times St)$ just state is recursive	

Now the big step semantics:

$$\frac{}{n, s \to n} \quad \frac{s(x) = (e, s')\ (e, s') \to v}{x, s \to v} \quad \frac{e, s \to \underline{m}, f, s \to \underline{n}}{e + f, s \to m + n}$$

$$\frac{}{\lambda x.e, s \to (x, e, s)}, \frac{e, s \to (x, e', s')\ , (e' s'[x \mapsto (f, s)]) \to v}{e\ f, s \to v}$$

We do not evaluate f anymore, we just pass it inside e'!

Dynamic binding with lazy evaluation Example for a difference between static and dynamic under lazy evaluation:

$$(\lambda x (\lambda y \lambda x\ y) x\ 3) 5$$

Under static we get 5, because y gets bound to expression x, with the environment where x was bound to 5, while in static the expression x gets evaluated in the internal environment where x is bound to 3.

	STATIC	DYNAMIC
EAGER	$Val = \mathbb{Z} \cup (Var \times Expr \times St)$ $St = Var \to Val$ mutually recursive	$Val = \mathbb{Z} \cup (Var \times Expr)$ $St = Var \to Val$ not recursive anymore!
LAZY	$Val = \mathbb{Z} \cup (Var \times Expr \times St)$ $St = Var \to (Expr \times St)$ just state is recursive	$Val = \mathbb{Z} \cup (Var \times Expr)$ $St = Var \to Expr$ again, not recursive!

And the rules:

$$\frac{s(x) = e\ (e, s) \to v}{x, s \to v} \quad \frac{}{\lambda x.e, s \to (x, e)}$$

$$\frac{(e, s) \to (x, e')\ (e', s[x \mapsto f]) \to v}{e\ f, s \to v}$$

10 Ćwiczenia n+1

$$c ::= \ldots | \texttt{for } x = e \texttt{ to } f \texttt{ try } c \texttt{ else } d | \texttt{fail}$$

How to interpret this?

1. If $e > f$ then do d.
2. Otherwise, $x := e$.
3. Do c.
4. If c succeeds, then succeed and restore x.
5. If c fails, then $x := x + 1$.
6. If $x \leq n$ go to step 3
7. Otherwise restore x and succeed

Big step semantics

$$C := c \times St \cup St \times \{\top, \bot\} \text{success - } \top, \text{ fail - } \bot$$

$$St = Var \to \mathbb{N}$$

$$\frac{}{\texttt{skip}, s \to s, \top}$$

$$\frac{}{\texttt{fail}, s \to s, \bot}$$

$$\frac{c, s \to s', \top \quad d, s' \to v}{c; d, s \to v}$$

$$\frac{c, s \to s', \bot}{c; d, s \to s', \bot}$$

$$\frac{e, s \to m \quad f, s \to n \quad m \leq n \quad c, s[x \mapsto m] \to s', \top}{\texttt{for } x = e \texttt{ to } f \texttt{ try } c \texttt{ else } d, s \to s'[x \mapsto s(x)], \top}$$

$$\frac{e, s \to m \quad f, s \to n \quad m > n \quad d, s \to v}{\texttt{for } x = e \texttt{ to } f \texttt{ try } c \texttt{ else } d, s \to v}$$

$$\frac{e, s \to m \quad f, s \to n \quad m \leq n \quad c, s[x \mapsto m] \to s', \bot \quad \texttt{for } x = m + 1 \texttt{ to } n \texttt{ try } c \texttt{ else skip}, s' \to s'', \_}{\texttt{for } x = e \texttt{ to } f \texttt{ try } c \texttt{ else } d, s \to s'[x \mapsto s(x)], \top}$$

10.1 Exceptions

$$c ::= \ldots | \texttt{throw}(e) | \texttt{try } c \texttt{ catch } (e) d$$

Configurations

$$C := c \times St \cup St \times (\mathbb{N} \cup \{\top\})$$

$$\frac{}{\texttt{skip}, s \to s, \top}$$

$$\frac{e \to k}{\texttt{throw } (e), s \to s, k}$$

$$\frac{c, s \to s', \top \quad d, s' \to v}{c; d, s \to v}$$

$$\frac{c, s \to (s', k), k \neq \top}{c; d, s \to s', k}$$

$$\frac{c \to s', \top}{\texttt{try } c \texttt{ catch } (e)\ d, s \to s', \top}$$

$$\frac{c \to (s', k), k \neq \top \quad e, s' \to n \quad n = k \quad d, s' \to v}{\texttt{try } c \texttt{ catch } (e)\ d, s \to v}$$

$$\frac{c \to (s', k), k \neq \top \quad e, s' \to n \quad n \neq k \quad d, s' \to v}{\texttt{try } c \texttt{ catch } (e)\ d, s \to s', k}$$

10.2 Nondeterministic programming language

$$c ::= 1 | c + c | c - c | x := c | c; c | c \texttt{ or } c | c? | c*$$

- assignment returns c
- semicolon returns second command
- `or` executes non-deterministically and returns what it executed
- question mark returns 1 if $c = 1$ and has no semantics otherwise
- star - non-deterministically choose a natural number and execute c that many times

Execute b, if it's 1, calculate c and repeat.

$$\texttt{while } b \texttt{ do } c \equiv (b?; c)*; (1 - b)?$$

We forced the program to non-deterministically choose the right amount of iterations in *.

$$\texttt{if } b \texttt{ then } c \texttt{ else } d \equiv (b?; c) \texttt{ or } ((1 - b)?; d)$$

Big step semantics

$$C := c \times St \cup St \times \mathbb{Z}$$

$$\frac{}{1 \to s, 1}$$

$$\frac{c, s \to s', m \quad d, s' \to s'', n}{c + d \to s'', m + n}$$

$$\frac{c, s \to s', n}{x := c \to s'[x \mapsto n], n}$$

$$\frac{c, s \to s', n \quad d, s' \to s'', m}{c; d \to s'', m}$$

$$\frac{c, s \to s', n}{c \texttt{ or } d \to s', m}$$

$$\frac{d, s \to s', n}{c \texttt{ or } d \to s', n}$$

$$\frac{c, s \to s', 1}{c? \to s', 1}$$

$$c* \equiv (c*; c)or1$$

11 Tutorial 28/11

11.1 Local blocks and variables

$$C \ni c ::= \ldots \mid \texttt{begin } \{d\} \texttt{ in } c$$

$$D \ni d ::= \texttt{Var } x | d; d$$

We don't have state anymore, instead we get:

$$s \in Store = Loc \to \mathbb{N}$$

$$\rho \in Env = Var \to Loc$$

Loc is a countable set of memory locations. We need not To get environments, we need a function $newloc : Env \to Loc$ that satisfies $newloc(\rho) \notin dom(\rho)$. Configurations:

$$C_1 = C \times Store \times Env \cup Store \text{ for statements}$$

$$C_2 = D \times Env \cup Env \text{ for declarations}$$

Expressions are just like before, except:

$$\frac{}{x, s, \rho \to s(\rho(x))}$$

Statements:

$$\frac{c_1, s, \rho \to s' \quad c_2, s', \rho \to s''}{c_1; c_2, s, \rho \to s''}$$

$$\frac{e, s, \rho \to n}{x := e, s, p \to s[\rho(x) \mapsto n]}$$

$$\frac{d, \rho \to \rho' \quad c, s, \rho' \to s'}{\texttt{begin } \{d\} \texttt{ in } c, s, p \to s'}$$

Declarations:

$$\frac{}{\texttt{Var } x, \rho \to \rho[x \mapsto \mathit{newloc}(\rho)]}$$

$$\frac{d_1, \rho \to \rho' \quad d_2, \rho' \to \rho''}{d_1; d_2, \rho \to \rho''}$$

11.2 Procedures with one integer variable parameter

We extend the declaration syntax:

$$d ::= \texttt{Var } x \mid d; d \mid \texttt{proc } x(y) := c$$

Where x is the procedure name and y is a formal parameter. We also extend the statement syntax:

$$C \ni c ::= \ldots \mid \texttt{begin } \{d\} \texttt{ in } c \mid \texttt{call } x(y)$$

where x is again the procedure name and y is the actual parameter.

11.3 Static binding with eager evaluation

We need to extend the environment to be able to store procedures. We'll use closures consisting of argument, command and environment.

$$Cl = Var \times C \times Env$$

The environment will also be different:

$$\rho \in Env = Var \rightharpoonup (Loc \cup Cl)$$

Big steps:

$$\frac{\rho(x) = (z, c, \rho') \quad l = \mathit{newloc}(\rho') \quad c, s[l \mapsto s(\rho(y))], \rho'[z \mapsto l] \to s'}{\texttt{call } x(y), s, \rho \to s'}$$

$$\frac{cl = (y, c, \rho)}{\texttt{proc } x(y) := c, \rho \to \rho[x \mapsto cl]}$$

11.3.1 Is this ~~loss~~ recursion?

$\texttt{begin } \{\texttt{Var } z; \texttt{proc } x(y) := (z := y); \texttt{proc } x(y) := \texttt{call } x(y)\}\texttt{call } x(y)$ The second procedure calls the first one!

11.3.2 Are we recursion yet?

Rewrite the procedure decl semantics:

$$\frac{cl = (y, c, \rho[x \mapsto cl])}{\texttt{proc}\ \ x(y) := c, \rho \to \rho[x \mapsto cl]}$$

This is an infinite closure!!!1!
How do we patch this? Either allow infinite closures (XD) or patch this at calltime.

$$\frac{\rho(x) = (z, c, \rho') \quad l = newloc(\rho') \quad c, s[l \mapsto s(\rho(y))], \rho'[z \mapsto l][x \mapsto (z, c, \rho')] \to s'}{\texttt{call}\ \ x(y), s, \rho \to s'}$$

There we update ρ' by mapping to ρ'! My lord, is this legal? We will make it legal! This is just a definition of $\rho'' = \rho'[z \mapsto l][x \mapsto (z, c, \rho')]$ which happens to rely on ρ' twice! Everything is cool.

11.4 Dynamic binding with eager evaluation

Now closures don't record environment!

$$Cl = Var \times C$$

$$\frac{\rho(x) = (z, c) \quad l = newloc(\rho) \quad c, s[l \mapsto s(\rho(y))], \rho[z \mapsto l] \to s'}{\texttt{call}\ \ x(y), s, \rho \to s'}$$

$$\frac{cl = (y, c)}{\texttt{proc}\ \ x(y) := c, \rho \to \rho[x \mapsto cl]}$$

Because we do the `call` with the environment local to the caller, the caller already has defined the function that they call, so we get recursion for freeeee!

11.5 Call by need

This tries to combine call by name and call by value. Call by name:

- terminates more often - good
- evaluates programs only if they are needed - good
- but if we need an argument many times, it gets evaluated again! - bad
- this is C macros in a nutshell

Call by value:

- We evaluate every argument **exactly** once - good and bad
- good bc we don't reevaluate
- bad bc we can evaluate unnecessarily

Call by need combines them into the ultimate Haskell experience:

$$Store = Loc \to (\mathbb{N} \cup Expr)$$

$$Env = Var \to (Loc \times Cl)$$
$$Cl = Var \times C$$
$$Conf = C_0 \cup C_1 \cup C_2$$

C_0 is for expressions, C_1 and C_2 are the same as before.

$$C_0 = Expr \times Store \times Env \cup \mathbb{N} \times Store$$

$$\frac{s(\rho(x)) = n}{x, s, p \to n, s}$$
$$\frac{s(\rho(x)) = e \quad e, s, \rho \to n}{x, s, p \to n, s[\rho(x) \mapsto n]}$$
$$\frac{\rho(x) = (y, c) \quad l = newloc(\rho) \quad c, s[l \mapsto e], \rho[y \mapsto l] \to s'}{\texttt{call } x(e), s, p \to n, s[\rho(x) \mapsto n]}$$

12 Tutorial 5/12

12.1 Denotational semantics

12.1.1 Arithmetic expressions

$$[[e]]_s \in \mathbb{N}, [[\_]]_\_ : Expr \to State \to \mathbb{N}, State : Var \to \mathbb{N}$$
$$[[n]]s = n, \text{ sometimes we also write } [[n]] = \lambda s.n$$

This is the same, but not in set theory without extentionality axiom.

$$[[x]]s = s\ x \quad [[x]] = \lambda s.s\ x \text{ - apply}$$

$$[[e + f]]s = [[e]]_s + [[f]]s$$
$$[[\texttt{let } x = e \texttt{ in } f]]s = [[f]]s[x \mapsto [[e]]s]$$

12.1.2 Programs

$$[[c]]s \in State_\perp, [[\_]]_\_ : C \to States \to States_\perp, States_\perp = States \cup \{\perp\}$$

$[[\texttt{skip}]]s = s$ - pointful notation, $[[\texttt{skip}]] = \lambda s.s = id$ - point-free (point-less) notation

$$[[x := e]]s = s[x \mapsto [[e]]s]$$
$$[[c; d]]s = [[d]]([[c]]s)$$

Introducing strictness in the semantic function: $[[c]]\perp = \perp$ regardless of what c is.

$$[[c; d]] = \lambda s.[[d]]([[c]]s) = [[d]] \circ [[c]]$$

For `if then else` we'll use a helper function $Cond_D(x,d,e) = \begin{cases} d & \texttt{if } x = \texttt{true} \\ e & \texttt{if } x = \texttt{false} \end{cases}$ where $d, e \in D$ and $x \in \{\texttt{true, false}\}$

$$[[\texttt{if } b \texttt{ then } c \texttt{ else } d]]s = Cond_{State}([[b]]s, [[c]]s, [[d]]s)$$

For our denotational semantics, if we want it to uniquely define a semantic we need compositionality.

$$[[\texttt{while } b \texttt{ do } c]]s = [[\texttt{if } b \texttt{ then } (c;\texttt{while } b \texttt{ do } c) \texttt{ else skip}]]s =$$

$$= Cond([[b]]s[[\texttt{while } b \texttt{ do } c]]([[c]]s), s)$$

This is, again, not a definition! If $w = \texttt{while } b \texttt{ do } c$, we have the following:

$$[[w]]s = [[w]]([[\texttt{skip}]]s) = [[w]]s$$

This means that our 'rule' didn't give us anything! How to do it? Let's define a partial order on states; $\forall_{s \in States_\perp} \perp \sqsubseteq S, S \sqsubseteq S$. We'll extend that to functions:

$$f, g : States_\perp \to States_\perp f \sqsubseteq g \text{ iff } \forall s \; f \; s \sqsubseteq g \; s$$

Note that $f, g \sim [[c]]$.
If an operator $\Phi : (States \to States_\perp) \to (States \to States_\perp)$ that applies the $Cond$ function ($\Phi(f) = \lambda s.Cond([[b]]s, f([[c]]s)), s)$ has fixed points, that is $f = \Phi \; f$, then we define the semantics of w as the least fixed point: $[[w]] = \mu f.\Phi(f)$

13 Tutorial 12/12

13.1 Denotational semantics of local variables and procedures

13.1.1 Call-by-name with static binding

Syntax:

$$c ::= \dots | \texttt{begin } \{d\} \texttt{ in } c | \texttt{ call } x(y)$$

$$d ::= \texttt{var } x := e | \texttt{proc x(}y\texttt{) := } c | d, d$$

Semantic domains:

$$Loc = \{0, 1, \dots\}$$

$$s \in Store = Loc \to \mathbb{Z}$$

$$\rho \in Env = Var \to Loc$$

$$\pi \in PEnv = Var \to Proc$$

In operational we used syntax and semantics to create closures. Here we can do better. In call by value, we'd use

$$Proc = \mathbb{Z} \to Store_\perp \to Store_\perp$$

Let's try that first.

$$C[[c]] : (Env \times PEnv \times Store_\perp) \to Store_\perp$$

$$D[[d]] : (Env \times PEnv \times Store) \to (Env \times PEnv \times Store)$$

Rules:

$$C[[\texttt{skip}]](\rho, \pi, s) = s$$

$$C[[x := e]](\rho, \pi, s) = s[\rho\ x \mapsto E[[e]]\rho s]$$

$$C[[C_1; C_2]](\rho, \pi, s) = C[[C_2]](\rho, \pi, C[[C_1]](\rho, \pi, s))$$

$$C[[\texttt{while}\ b\ \texttt{do}\ c]](\rho, \pi, s) =$$

We need to find a fixed point of $Cond : Bool \to Store_\perp \to Store_\perp \to Store_\perp$ again!

$$C[[\texttt{while}\ b\ \texttt{do}\ c]](\rho, \pi, s) = Cond(B[[b]]\rho s, C[[w]](\rho, \pi, C[[c]]\rho\pi s), s)$$

This is just an equation, not a definition!

$$\Phi(F) = \lambda s'.Cond(B[[b]]\rho s', F(\rho, \pi, C[[c]]\rho\pi s'), s')$$

$$C[[\texttt{while}\ b\ \texttt{do}\ c]](\rho, \pi, s) = (\mu F.\Phi)s$$

($\mu x.y$ means lowest x that is a fixed point of y)

$$C[[\texttt{begin}\ \{d\}\ \texttt{in}\ c]](\rho, \pi, s) = C[[c]](D[[d]](\rho, \pi, s)) = C[[c]]$$

$$C[[\texttt{call x(}y\texttt{)}]](\rho, \pi, s) = \pi(x)s(\rho(y))s$$

$$D[[\texttt{var}\ x := e]](\rho, \pi, s) = (\rho[x \mapsto e], \pi, s[l \mapsto [[e]]\rho s])$$

$$D[[\texttt{proc}\ x(y) := c]](\rho, \pi, s) = (\rho, \pi[x \mapsto \lambda n.\lambda s'.C[[c]](\rho[y \mapsto l], s'[l \mapsto n])], s)$$

Where $l = newloc(\rho)$.

Now call by name

$$Proc = Loc \to Store_\perp \to Store_\perp$$

$$D[[\texttt{proc}\ x(y) := c]](\rho, \pi, s) = (\rho, \pi[x \mapsto \lambda v.\lambda s'.C[[c]](\rho[y \mapsto v], \pi, s')], s)$$

$$C[[\texttt{call x(}y\texttt{)}]](\rho, \pi, s) = \pi(x)(\rho\ y)s$$

13.2 Continuations

Let's define a function:

$$f\ 0 = 1$$

$$f\ n = nf(n-1)$$

This is a classic style factorial $f : \mathbb{N} \to \mathbb{N}^+$. The continuation style factorial is $\hat{f} : \mathbb{N} \to (\mathbb{N}^+ \to \mathbb{N}^+) \to \mathbb{N}^+$

$$\hat{f}\ 0\ k = k\ 1$$

$$\hat{f}\ n\ k = \hat{f}(n-1)(\lambda m.k(nm))$$

Let's prove that the produce the same results, i.e. $f\ n = \hat{f}\ n\ id$: First, let $n * \_ = \lambda m.n * m$, so that

$$\hat{f}\ n\ k = \hat{f}(n-1)(\lambda m.k(nm)) = \hat{f}(n-1)(k \circ (n * \_))$$

Easier to see if defined as:

$$\hat{f}\ 0 = \lambda k.k\ 1$$

$$\hat{f}\ n = \lambda k.\hat{f}(n-1)(k \circ (n * \_))\ 1$$