Formal Techniques for Software Engineering: More on Denotational Semantics

Rocco De Nicola

IMT Institute for Advanced Studies, Lucca
rocco.denicola@imtlucca.it

May 2013

Lesson 5

Syntax and Semantics

Syntax

Set of rules for defining "well formed phrases"

Syntactic Domain

Set of well formed phrases

Semantic Domain

Set of known entities

Semantic Interpretation

Mapping from Syntactic Domain to Semantic Domain or Interpretation of well formed phrases in terms of known concepts

Tiny: A simple imperative language

Syntax

$$e \quad ::= \quad \textbf{true} \mid \textbf{false} \mid \textbf{not}\ e \mid n \mid e_1\ nop\ e_2 \mid e_1\ bop\ e_2 \mid \textbf{read} \mid x$$

$$c \quad ::= \quad \textbf{noaction} \mid x := e \mid c_1 ; c_2 \mid \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \mid \textbf{while}\ e\ \textbf{do}\ c \mid \textbf{output}\ e$$

- *Exp* denotes the set of expressions generable by the above grammar starting from *e*
- *Com* denotes the set of commands generable by the above grammar starting from *c*

Tiny: A simple imperative language

Syntax

$$e ::= \textbf{true} \mid \textbf{false} \mid \textbf{not}\ e \mid n \mid e_1\ nop\ e_2 \mid e_1\ bop\ e_2 \mid \textbf{read} \mid x$$

$$c ::= \textbf{noaction} \mid x := e \mid c_1 ; c_2 \mid \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \mid \textbf{while}\ e\ \textbf{do}\ c \mid \textbf{output}\ e$$

- *Exp* denotes the set of expressions generable by the above grammar starting from e
- *Com* denotes the set of commands generable by the above grammar starting from c

Transition System for Expressions

The transition system for expressions is: $(\Sigma, \Sigma_F, \longrightarrow\!\!\!\!\rightarrow)$

- $\Sigma = \{\langle e, \sigma\rangle \mid e \in Exp, \sigma : \mathrm{Id} \longrightarrow \mathrm{Val}\} \cup \{\sigma : \mathrm{Id} \longrightarrow \mathrm{Val}\}$
- $\Sigma_F = \{\sigma : \mathrm{Id} \longrightarrow \mathrm{Val}\}$

Tiny: A simple imperative language

Syntax

$$e \ ::= \ \textbf{true} \mid \textbf{false} \mid \textbf{not}\ e \mid n \mid e_1\ nop\ e_2 \mid e_1\ bop\ e_2 \mid \textbf{read} \mid x$$

$$c \ ::= \ \textbf{noaction} \mid x := e \mid c_1 ; c_2 \mid \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \ \mid \textbf{while}\ e\ \textbf{do}\ c \mid \textbf{output}\ e$$

- *Exp* denotes the set of expressions generable by the above grammar starting from e
- *Com* denotes the set of commands generable by the above grammar starting from c

Transition System for Commands

The transition system for commands is: $(K, K_F, \longrightarrow)$

- $K = \{\langle c, \sigma \rangle \mid c \in Com, \sigma : \mathrm{Id} \longrightarrow \mathrm{Val}\}$
- $K_F = \{\langle \textbf{noaction}, \sigma \rangle \mid \sigma : \mathrm{Id} \longrightarrow \mathrm{Val}\}$

Notation

State update

Given state σ, value $v \in \text{Val}$, identifier $id \in \text{Id}$,

$$\begin{array}{rcll} \sigma[v/id](id') & = & \sigma(id') & \textit{if } id' \neq id, \\ & = & v & \textit{otherwise.} \end{array}$$

Before it was $\sigma\{v \mapsto id\}$

Special identifiers

- `res`: will denote a basic (boolean or natural) value
- `in`: will denote a sequence of basic values - inputs
- `out`: will denote a sequence of basic values - outputs

  - Results of a function evaluations are associated to `res`.
  - **read** extracts a value from `in` and stores them in `res`
  - **output** adds values to `out`

Operational Semantics of Expressions

$$\langle \mathbf{true}, \sigma \rangle \twoheadrightarrow \sigma[\mathit{true}/\mathtt{res}] \qquad \text{(True)}$$

$$\langle \mathbf{false}, \sigma \rangle \twoheadrightarrow \sigma[\mathit{false}/\mathtt{res}] \qquad \text{(False)}$$

$$\langle n, \sigma \rangle \twoheadrightarrow \sigma[n/\mathtt{res}] \qquad \text{(Nat)}$$

$$\langle x, \sigma \rangle \twoheadrightarrow \sigma[\sigma(x)/\mathtt{res}] \qquad \text{(Ide)}$$

$$\frac{\langle e, \sigma \rangle \twoheadrightarrow \sigma' \qquad \sigma'(\mathtt{res}) = v}{\langle \mathbf{not}\ e, \sigma \rangle \twoheadrightarrow \sigma'[\neg v/\mathtt{res}]} \qquad \text{(Not)}$$

$$\frac{\begin{array}{cc}\langle e_1, \sigma \rangle \twoheadrightarrow \sigma' & \sigma'(\mathtt{res}) = v_1 \\ \langle e_2, \sigma' \rangle \twoheadrightarrow \sigma'' & \sigma''(\mathtt{res}) = v_2\end{array}}{\langle e_1\ \mathit{bop}\ e_2, \sigma \rangle \twoheadrightarrow \sigma''[v_1\ \mathit{bop}\ v_2/\mathtt{res}]} \qquad \text{(Bop)}$$

$$\frac{\begin{array}{cc}\langle e_1, \sigma \rangle \twoheadrightarrow \sigma' & \sigma'(\mathtt{res}) = v_1 \\ \langle e_2, \sigma' \rangle \twoheadrightarrow \sigma'' & \sigma''(\mathtt{res}) = v_2\end{array}}{\langle e_1\ \mathit{nop}\ e_2, \sigma \rangle \twoheadrightarrow \sigma''[v_1\ \mathit{nop}\ v_2/\mathtt{res}]} \qquad \text{(Nop)}$$

$$\langle \mathbf{read}, \sigma \rangle \twoheadrightarrow \sigma[\mathit{hd}(\sigma(\mathtt{in}))/\mathtt{res}, \mathit{tl}(\sigma(\mathtt{in}))/\mathtt{in}] \qquad \text{(Read)}$$

Operational Semantics of Commands

$$\frac{\langle e, \sigma \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma' \qquad \sigma'(\mathtt{res}) = v}{\langle x := e, \sigma \rangle \longrightarrow \langle \textbf{noaction}, \sigma'[v/x] \rangle} \quad (\text{Ass})$$

$$\langle \textbf{noaction}; c_2, \sigma \rangle \longrightarrow \langle c_2, \sigma \rangle \quad (\text{Seq}_1)$$

$$\frac{\langle c_1, \sigma \rangle \longrightarrow \langle c_1', \sigma' \rangle}{\langle c_1; c_2, \sigma \rangle \longrightarrow \langle c_1'; c_2, \sigma' \rangle} \quad (\text{Seq}_2)$$

$$\frac{\langle e, \sigma \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma' \qquad \sigma'(\mathtt{res}) = \mathit{true}}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \sigma \rangle \longrightarrow \langle c_1, \sigma' \rangle} \quad (\text{Cond}_1)$$

$$\frac{\langle e, \sigma \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma' \qquad \sigma'(\mathtt{res}) = \mathit{false}}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \sigma \rangle \longrightarrow \langle c_2, \sigma' \rangle} \quad (\text{Cond}_2)$$

$$\frac{\langle e, \sigma \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma' \qquad \sigma'(\mathtt{res}) = \mathit{true}}{\langle \textbf{while } e \textbf{ do } c, \sigma \rangle \longrightarrow \langle c; \textbf{while } e \textbf{ do } c, \sigma' \rangle} \quad (\text{While}_1)$$

$$\frac{\langle e, \sigma \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma' \qquad \sigma'(\mathtt{res}) = \mathit{false}}{\langle \textbf{while } e \textbf{ do } c, \sigma \rangle \longrightarrow \langle \textbf{noaction}, \sigma' \rangle} \quad (\text{While}_2)$$

$$\frac{\langle e, \sigma \rangle \longrightarrow\!\!\!\!\!\rightarrow \sigma' \qquad \sigma'(\mathtt{res}) = v}{\langle \textbf{output } e, \sigma \rangle \longrightarrow \langle \textbf{noaction}, \sigma'[v :: (\sigma(\mathtt{out}))/\mathtt{out}] \rangle} \quad (\text{Out})$$

Domains for Denotational Semantics

Semantic Domains

To specify the interpretation functions $\mathcal{E}$ and $\mathcal{C}$ for *Exp* and *Com* we need to specify their semantic domain $\mathbb{D}_1$ e $\mathbb{D}_2$:

$$\mathcal{E} : \mathit{Exp} \longrightarrow \mathbb{D}_1$$
$$\mathcal{C} : \mathit{Com} \longrightarrow \mathbb{D}_2.$$

We make use of

$$\begin{array}{rcl} \mathbb{VAL} & = & \mathbb{NAT} + \mathbb{BOOL} \\ \mathbb{MEM} & = & \mathbb{ID} \longrightarrow (\mathbb{VAL} + \{\mathit{unbound}\}) \\ \mathbb{STATE} & = & \mathbb{VAL}^* \times \mathbb{VAL}^* \times \mathbb{MEM}. \end{array}$$

Semantic Domains

$$\mathcal{E} : \mathit{Exp} \longrightarrow \mathbb{STATE} \longrightarrow ((\mathbb{VAL} \times \mathbb{STATE}) + \{\mathit{error}\})$$

$$\mathcal{C} : \mathit{Com} \longrightarrow \mathbb{STATE} \longrightarrow (\mathbb{STATE} + \{\mathit{error}\})$$

Auxiliary notation

let and *cond* construct

We shall use

- $\mathit{let}\ x\ \mathit{be}\ e_1\ \mathit{in}\ e_2$ instead of $(\lambda x.e_2)e_1$.
- $e \rightarrow e_1, e_2$ instead of $\mathit{cond}(e, e_1, e_2)$

cases construct

If $p_i(\cdot)$ is a predicate selecting e_i according to the properties (type, value, structure) of e, we use

$$\begin{array}{ll} \mathit{cases}\ e\ \ \mathit{of} & p_1(e) : e_1 \\ & p_2(e) : e_2 \\ & \ldots \\ & p_n(e) : e_n \\ \mathit{endcases} & \end{array}$$

instead of

$$\mathit{let}\ x\ \mathit{be}\ e\ \mathit{in}$$

$$p_1(x) \rightarrow e'_1, (p_2(x) \rightarrow e'_2, (\ldots(p_n(x) \rightarrow e'_n)\ldots))$$

Denotational Semantics of TINY Expressions

$$\mathcal{E}[\![\textbf{true}]\!] = \lambda\sigma.\ < true, \sigma >$$

$$\mathcal{E}[\![\textbf{false}]\!] = \lambda\sigma.\ < false, \sigma >$$

$$\mathcal{E}[\![n]\!] = \lambda\sigma.\ < n, \sigma >$$

$$\mathcal{E}[\![\textbf{read}]\!] = \lambda(in, out, mem).\ < hd(in), < tl(in), out, mem >>$$

$$\mathcal{E}[\![\textbf{not}\ e]\!] = \lambda\sigma.\ < \neg\pi_1(\mathcal{E}[\![e]\!]\sigma), \pi_2(\mathcal{E}[\![e]\!]\sigma) >$$

$$\mathcal{E}[\![e_1\ nop\ e_2]\!] = \lambda\sigma.\left\langle \pi_1(\mathcal{E}[\![e_1]\!]\,\sigma)\ nop\ \pi_1\Big(\mathcal{E}[\![e_2]\!](\pi_2(\mathcal{E}[\![e_1]\!]\,\sigma))\Big), \pi_2\Big(\mathcal{E}[\![e_2]\!](\pi_2(\mathcal{E}[\![e_1]\!]\,\sigma))\Big)\right\rangle$$

$$\mathcal{E}[\![e_1\ bop\ e_2]\!] = \lambda\sigma.\ let\ (\mathcal{E}[\![e_1]\!]\,\sigma)\ be\ < v_1, \sigma_1 >\ in\ let\ (\mathcal{E}[\![e_2]\!]\,\sigma_1)\ be\ < v_2, \sigma_2 >\ in\ < v_1\ bop\ v_2, \sigma_2 >$$

$$\mathcal{E}[\![x]\!] = \lambda(in, out, mem).\ mem(x) = unbound \rightarrow error, < mem(x), < in, out, mem >>$$

Denotational Semantics of TINY Commands

$\mathcal{C}[\![\textbf{noaction}]\!] = \lambda\sigma.\,\sigma$

$\mathcal{C}[\![x := e]\!] = \lambda\sigma.\; < \pi_1\big(\pi_2(\mathcal{E}[\![e]\!]\sigma)\big), \pi_2(\sigma), \pi_3(\sigma)[\pi_1(\mathcal{E}[\![e]\!]\sigma)/x] >$

$\mathcal{C}[\![c_1\,;\,c_2]\!] = \lambda\sigma.\,\mathcal{C}[\![c_2]\!]\,(\mathcal{C}[\![c_1]\!]\sigma);$

$$\mathcal{C}[\![\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2]\!] = \lambda\sigma.\,let\,(\mathcal{E}[\![e]\!]\,\sigma)\;be\; < v, \sigma' > \;in \\ v \rightarrow \mathcal{C}[\![c_1]\!]\,\sigma', \mathcal{C}[\![c_2]\!]\,\sigma'$$

$$\mathcal{C}[\![\textbf{output } e]\!] = \lambda\sigma.\,let\,(\mathcal{E}[\![e]\!]\,\sigma)\;be\; < v, < in, out, mem >> \;in \\ < in, v :: out, mem >$$

$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] = fix\Big(\lambda\Theta_w.\,\lambda\sigma.\,let\,(\mathcal{E}[\![e]\!]\,\sigma)\;be\; < v, \sigma' > \;in \\ v \rightarrow \Theta_w(\mathcal{C}[\![c]\!]\,\sigma'), \sigma'\Big)$$

Why the latter?

Denotational Semantics of **while**

$$\textbf{while } e \textbf{ do } c \quad \approx \quad \textbf{if } e \textbf{ then } (\ c; \textbf{while } e \textbf{ do } c\ ) \textbf{ else noaction}$$

By considering the semantics of **if**, we have:

$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] = \lambda\sigma.\ let\ (\mathcal{E}[\![e]\!]\,\sigma)\ be\ < v, \sigma' >\ in$$
$$v \rightarrow \mathcal{C}[\![c; \textbf{while } e \textbf{ do } c]\!]\,\sigma', \mathcal{C}[\![\textbf{noaction}]\!]\,\sigma'$$

By considering the semantics of ; and **noaction**, we have:

$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] = \lambda\sigma.\ let\ (\mathcal{E}[\![e]\!]\,\sigma)\ be\ < v, \sigma' >\ in$$
$$v \rightarrow \mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!]\,(\mathcal{C}[\![c]\!]\,\sigma'), \mathcal{C}[\![\textbf{noaction}]\!]\,\sigma'$$

$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] = \Theta_w = \lambda\sigma.\ let\ (\mathcal{E}[\![e]\!]\,\sigma)\ be\ < v, \sigma' >\ in$$
$$v \rightarrow \Theta_w(\mathcal{C}[\![c]\!]\,\sigma'), \sigma'$$

By abstracting on Θ_w we have a recursive function for which we can calculate the fixed point:

$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] =$$
$$\lambda\Theta_w.\ \lambda\sigma.\ let\ (\mathcal{E}[\![e]\!]\,\sigma)\ be\ < v, \sigma' >\ in\ v \rightarrow \Theta_w(\mathcal{C}[\![c]\!]\,\sigma'), \sigma'$$

Denotational Semantics of **while**

while e **do** c $\approx$ **if** e **then** (c; **while** e **do** c) **else noaction**

By considering the semantics of **if**, we have:

$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] = \lambda\sigma.\, let\, (\mathcal{E}[\![e]\!]\,\sigma)\ be\ < v, \sigma' >\ in$$
$$v \rightarrow \mathcal{C}[\![c; \textbf{while } e \textbf{ do } c]\!]\,\sigma', \mathcal{C}[\![\textbf{noaction}]\!]\,\sigma'$$

By considering the semantics of ; and **noaction**, we have:

$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] = \lambda\sigma.\, let\, (\mathcal{E}[\![e]\!]\,\sigma)\ be\ < v, \sigma' >\ in$$
$$v \rightarrow \mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!]\,(\mathcal{C}[\![c]\!]\,\sigma'), \mathcal{C}[\![\textbf{noaction}]\!]\,\sigma'$$
$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] = \Theta_w = \lambda\sigma.\, let\, (\mathcal{E}[\![e]\!]\,\sigma)\ be\ < v, \sigma' >\ in$$
$$v \rightarrow \Theta_w(\mathcal{C}[\![c]\!]\,\sigma'), \sigma'$$

By abstracting on Θ_w we have a recursive function for which we can calculate the fixed point:

$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] =$$
$$\lambda\Theta_w.\, \lambda\sigma.\, let\, (\mathcal{E}[\![e]\!]\,\sigma)\ be\ < v, \sigma' >\ in\ v \rightarrow \Theta_w(\mathcal{C}[\![c]\!]\,\sigma'), \sigma'$$

Denotational Semantics of **while**

while e **do** c $\approx$ **if** e **then** (c; **while** e **do** c) **else noaction**

By considering the semantics of **if**, we have:

$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] = \lambda\sigma.\, let\, (\mathcal{E}[\![e]\!]\,\sigma)\ be\ < v, \sigma' > in$$
$$v \rightarrow \mathcal{C}[\![c; \textbf{while } e \textbf{ do } c]\!]\,\sigma', \mathcal{C}[\![\textbf{noaction}]\!]\,\sigma'$$

By considering the semantics of ; and **noaction**, we have:

$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] = \lambda\sigma.\, let\, (\mathcal{E}[\![e]\!]\,\sigma)\ be\ < v, \sigma' > in$$
$$v \rightarrow \mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!]\,(\mathcal{C}[\![c]\!]\,\sigma'), \mathcal{C}[\![\textbf{noaction}]\!]\,\sigma'$$
$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] = \Theta_w = \lambda\sigma.\, let\, (\mathcal{E}[\![e]\!]\,\sigma)\ be\ < v, \sigma' > in$$
$$v \rightarrow \Theta_w(\mathcal{C}[\![c]\!]\,\sigma'), \sigma'$$

By abstracting on Θ_w we have a recursive function for which we can calculate the fixed point:

$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] =$$
$$\lambda\Theta_w.\, \lambda\sigma.\, let\, (\mathcal{E}[\![e]\!]\,\sigma)\ be\ < v, \sigma' > in\ v \rightarrow \Theta_w(\mathcal{C}[\![c]\!]\,\sigma'), \sigma'$$

A richer language

Syntax for SMALL

$$
\begin{array}{lcl}
prog & ::= & \textbf{program}\ c \\
d & ::= & \textbf{const}\ x = e \mid \textbf{var}\ x = e \mid \textbf{proc}\ p(x); c \mid \textbf{fun}\ f(x); e \mid d_1 ; d_2 \\
e & ::= & b \mid n \mid \textbf{not}\ e \mid e_1\ nop\ e_2 \mid e_1\ bop\ e_2 \\
 & & \mid \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid x \mid e(e_1) \mid \textbf{read} \\
c & ::= & e := e_1 \mid c_1 ; c_2 \mid \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ e\ \textbf{do}\ c \mid \\
 & & \mid \textbf{output}\ e \mid \textbf{begin}\ d; c\ \textbf{end} \mid e(e_1)
\end{array}
$$

New Ingredients

- Blocks for variable scoping
- Procedure Calls
- Function Calls
- No **noaction**

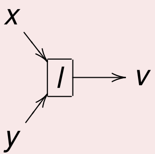
Blocks and Variables in SMALL

An Example SMALL Program

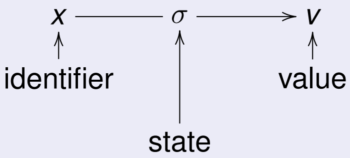
```
program
    begin
        var x = 100;
        var y = 0;
            y := x;
            begin
                var x = 1;
                    y := x
            end;
            y := x
        end
```

Scopes of variables

- A variable can have different values depending on the block in which it is declared
- Two variables can refer to the same location (aliasing) and take always the same value.



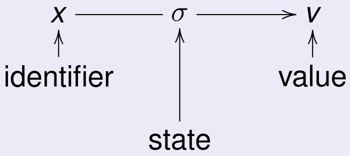
Binding in TINY



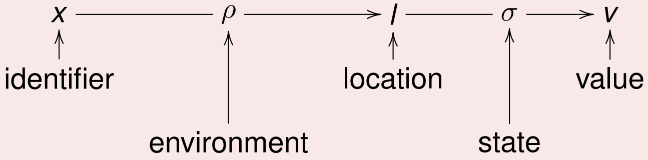
Binding in SMALL

Variables binding

Binding in TINY



Binding in SMALL



Semantic Domains

Values

$\mathbb{BVAL}$ Basic Values: can be input or output of Programs (e.g. naturals and booleans).

$\mathbb{NVAL}$ Nameable Values can be denoted by an identifier (e.g. locations or procedures).

$\mathbb{EVAL}$ Expressible Values: the set of values that expression can take (e.g. functions or basic values)

$\mathbb{SVAL}$ Storable Values: the set of values associated to memory locations (e.g. basic values or sequences thereof)

Semantic Domains for SMALL

$$\begin{aligned}
\mathbb{BVAL} &= \mathbb{NAT} + \mathbb{BOOL} \\
\mathbb{NVAL} &= \mathbb{BVAL} + \mathbb{LOC} + \mathbb{FUN} + \mathbb{PROC} \\
\mathbb{EVAL} &= \mathbb{NVAL} \\
\mathbb{SVAL} &= \mathbb{BVAL} + \mathbb{BVAL}^*
\end{aligned}$$

Semantic Domains

Values

- $\mathbb{BVAL}$ Basic Values: can be input or output of Programs (e.g. naturals and booleans).
- $\mathbb{NVAL}$ Nameable Values can be denoted by an identifier (e.g. locations or procedures).
- $\mathbb{EVAL}$ Expressible Values: the set of values that expression can take (e.g. functions or basic values)
- $\mathbb{SVAL}$ Storable Values: the set of values associated to memory locations (e.g. basic values or sequences thereof)

Semantic Domains for SMALL

$$\begin{aligned}
\mathbb{BVAL} &= \mathbb{NAT} + \mathbb{BOOL} \\
\mathbb{NVAL} &= \mathbb{BVAL} + \mathbb{LOC} + \mathbb{FUN} + \mathbb{PROC} \\
\mathbb{EVAL} &= \mathbb{NVAL} \\
\mathbb{SVAL} &= \mathbb{BVAL} + \mathbb{BVAL}^*
\end{aligned}$$

Strores and Environments

Domains for Environment and Stores

$$\begin{array}{rcl} \mathbb{ENV} & = & \mathbb{ID} \longrightarrow (\mathbb{NVAL} + \{\mathit{unbound}\}) \\ \mathbb{STORE} & = & \mathbb{LOC} \longrightarrow (\mathbb{SVAL} + \{\mathit{unused}\}) \end{array}$$

Updates for Stores and Environment

1. $\rho[loc/id]$ stands for $\lambda x.\,(x = id) \rightarrow loc, \rho(x)$;
2. $\rho[\rho']$ stands for $\lambda x.\,(\rho'(x) = unbound) \rightarrow \rho(x), \rho'(x)$.
3. $\sigma[val/loc]$ stands for $\lambda x.\,(x = loc) \rightarrow val, \sigma(x)$.

Generating new locations

To refer to a new location we use

$$new : \mathbb{STORE} \longrightarrow \mathbb{LOC}$$

that applied to state σ returns the smallest n that has never been used, e.g, n such that $\sigma(n) = unused$ and $\sigma(m) \neq unused, \forall m < n$.

Semantic Interpretation Function

Programs

$\mathcal{P} : \mathit{Prog} \longrightarrow \mathbb{BVAL}^* \longrightarrow (\mathbb{BVAL}^* + \{\mathit{error}\})$

Declarations

$\mathcal{D} : \mathit{Dec} \longrightarrow \mathbb{ENV} \longrightarrow \mathbb{STORE} \longrightarrow ((\mathbb{ENV} \times \mathbb{STORE}) + \{\mathit{error}\})$

Expressions

$\mathcal{E} : \mathit{Exp} \longrightarrow \mathbb{ENV} \longrightarrow \mathbb{STORE} \longrightarrow ((\mathbb{EVAL} \times \mathbb{STORE}) + \{\mathit{error}\})$

$\mathcal{R} : \mathit{Exp} \longrightarrow \mathbb{ENV} \longrightarrow \mathbb{STORE} \longrightarrow ((\mathbb{BVAL} \times \mathbb{STORE}) + \{\mathit{error}\})$

Commands

$\mathcal{C} : \mathit{Com} \longrightarrow \mathbb{ENV} \longrightarrow \mathbb{STORE} \longrightarrow (\mathbb{STORE} + \{\mathit{error}\})$

Auxiliary Operators

An operator for error handling

1. If $f : D_1 \longrightarrow (D_2 + \{error\})$ and $g : D_2 \longrightarrow (D_3 + \{error\})$, then

$$f \star g : D_1 \longrightarrow (D_3 + \{error\})$$

$$f \star g = \lambda x.\, f\, x = error \rightarrow error, g(fx).$$

2. If $f : D_1 \longrightarrow ((D_2 \times D_3) + \{error\})$ and
   $g : D_2 \longrightarrow D_3 \longrightarrow (D_4 + \{error\})$ then

$$f \star g : D_1 \longrightarrow (D_4 + \{error\})$$

$$\begin{aligned} f \star g = \lambda x.\, & cases\ f\ x\ of \\ & \quad < d_1, d_2 >: g\ d_1\ d_2; \\ & \quad error : error \\ & endcases \end{aligned}$$

Auxiliary Operators

Using $\star$

If we would have considered errors, the full semantics of ; in TINY would have been:

$$\begin{aligned}\mathcal{C}[\![c_1 ; c_2]\!] = \lambda\sigma.\ \textit{cases}\ \mathcal{C}[\![c_1]\!]\,\sigma\ \textit{of}\\ \sigma' : \mathcal{C}[\![c_2]\!]\,\sigma';\\ \textit{error} : \textit{error}\\ \textit{endcases}\end{aligned}$$

with the $\star$ operator, it becomes:

$$\mathcal{C}[\![c_1 ; c_2]\!] = \mathcal{C}[\![c_1]\!] \star \mathcal{C}[\![c_2]\!].$$

Auxiliary Operators

Checking types of results

$$checkD = \lambda v.\, \lambda\sigma.\, isD(v) \rightarrow < v, \sigma >, error.$$

checkD acts as a filter between two functions. It that transmits only the result of the first function to the second one only if it is of type *D*.

By using $\star$ and *checkD* we have that the semantics of **not** in TINY, that when taking errors into account would have been:

$$\begin{aligned} \mathcal{E}[\![\textbf{not } e]\!] = \lambda\sigma.\, cases\, \mathcal{E}[\![e]\!]\sigma \text{ of} & \\ & < v, \sigma' >: isbool(v) \rightarrow < \neg v, \sigma' >, error; \\ & error : error \\ & endcases \end{aligned}$$

becomes:

$$\mathcal{E}[\![\textbf{not } e]\!] = \mathcal{E}[\![e]\!] \star checkBOOL \star \lambda v\, \sigma. < \neg v, \sigma >$$

Denotational Semantics of SMALL

Semantics of Programs

$$\mathcal{P} : \mathit{Prog} \longrightarrow \mathbb{BVAL}^* \longrightarrow (\mathbb{BVAL}^* + \{\mathit{error}\})$$

$$\begin{aligned}\mathcal{P}[\![\textbf{program } c]\!]\mathit{in} = \mathit{cases}\ & \mathcal{C}[\![c]\!]\rho_0(\lambda x.\mathit{unused})[\mathit{in}/\mathit{lin}][\mathit{nil}/\mathit{lout}]\ \mathit{of} \\ & \sigma : \sigma(\mathit{lout}); \\ & \mathit{error} : \mathit{error} \\ & \mathit{endcases}\end{aligned}$$

Semantics of Declarations

$$\mathcal{D} : \mathit{Dec} \longrightarrow \mathbb{ENV} \longrightarrow \mathbb{STORE} \longrightarrow ((\mathbb{ENV} \times \mathbb{STORE}) + \{\mathit{error}\})$$

$$\mathcal{D}[\![\textbf{const } x = e]\!]\,\rho = \mathcal{R}[\![e]\!]\,\rho \star \lambda v\,\sigma.\ < \rho_0[v/x], \sigma >$$

$$\mathcal{D}[\![\textbf{var } x = e]\!]\,\rho = \mathcal{R}[\![e]\!]\,\rho \star \lambda v\,\sigma.\ < \rho_0[\mathit{new}\,\sigma/x], \sigma[v/\mathit{new}\,\sigma] >$$

$$\mathcal{D}[\![\textbf{proc } p(x); c]\!]\,\rho = \lambda\sigma.\ < \rho_0[(\lambda d.\mathcal{C}[\![c]\!]\,\rho[d/x])/p], \sigma >$$

$$\mathcal{D}[\![\textbf{fun } f(x); e]\!]\,\rho = \lambda\sigma.\ < \rho_0[(\lambda d.\mathcal{E}[\![e]\!]\,\rho[d/x])/f], \sigma >$$

Denotational Semantics of SMALL

Semantics of Programs

$$\mathcal{P} : Prog \longrightarrow \mathbb{BVAL}^* \longrightarrow (\mathbb{BVAL}^* + \{error\})$$

$$\mathcal{P}[\![\textbf{program } c]\!] in = cases\, \mathcal{C}[\![c]\!]\rho_0(\lambda x.unused)[in/lin][nil/lout] \text{ of}$$
$$\sigma : \sigma(lout);$$
$$error : error$$
$$endcases$$

Semantics of Declarations

$$\mathcal{D} : Dec \longrightarrow \mathbb{ENV} \longrightarrow \mathbb{STORE} \longrightarrow ((\mathbb{ENV} \times \mathbb{STORE}) + \{error\})$$

$$\mathcal{D}[\![\textbf{const } x = e]\!]\,\rho = \mathcal{R}[\![e]\!]\,\rho \star \lambda v\,\sigma.\ < \rho_0[v/x], \sigma >$$

$$\mathcal{D}[\![\textbf{var } x = e]\!]\,\rho = \mathcal{R}[\![e]\!]\,\rho \star \lambda v\,\sigma.\ < \rho_0[new\,\sigma/x], \sigma[v/new\,\sigma] >$$

$$\mathcal{D}[\![\textbf{proc } p(x); c]\!]\,\rho = \lambda\sigma.\ < \rho_0[(\lambda d.\,\mathcal{C}[\![c]\!]\,\rho[d/x])/p], \sigma >$$

$$\mathcal{D}[\![\textbf{fun } f(x); e]\!]\,\rho = \lambda\sigma.\ < \rho_0[(\lambda d.\mathcal{E}[\![e]\!]\,\rho[d/x])/f], \sigma >$$

Functions and Procedures

Types

$$\mathbb{FUN} = \mathbb{NVAL} \longrightarrow \mathbb{STORE} \longrightarrow ((\mathbb{EVAL} \times \mathbb{STORE}) + \{error\})$$

$$\mathbb{PROC} = \mathbb{NVAL} \longrightarrow \mathbb{STORE} \longrightarrow (\mathbb{STORE} + \{error\})$$

Declarations

$$\mathcal{D}[\![\textbf{proc } p(x); c]\!]\,\rho = \lambda\sigma.\ < \rho_0[(\lambda d.\mathcal{C}[\![c]\!]\,\rho[d/x])/p], \sigma >$$

$$\mathcal{D}[\![\textbf{fun } f(x); e]\!]\,\rho = \lambda\sigma.\ < \rho_0[(\lambda d.\mathcal{E}[\![e]\!]\,\rho[d/x])/f], \sigma >$$

Invocations

$$\mathcal{E}[\![e(e')]\!]\,\rho = \mathcal{E}[\![e]\!]\,\rho \star checkFUN \star \lambda f.\mathcal{E}[\![e']\!]\,\rho \star \lambda v.\lambda\sigma.f\,v\,\sigma$$

$$\mathcal{C}[\![e(e')]\!]\,\rho = \mathcal{E}[\![e]\!]\,\rho \star checkPROC \star \lambda p.\mathcal{E}[\![e']\!]\,\rho \star \lambda v\,\sigma.p\,v\,\sigma$$

Denotational Semantics of Expressions

Another auxiliary operator

To avoid explicitly dereferencing the result of the evaluation of an expression when this yields a location - $\sigma(\mathcal{E}[\![e]\!])$-, a new valuation function for expressions $\mathcal{R}$ is introduced that is similar to $\mathcal{E}$, but yields *error* when $\mathcal{E}[\![e]\!]$ is not a basic value or a location.

$$\mathcal{R} : Exp \longrightarrow \mathbb{ENV} \longrightarrow \mathbb{STORE} \longrightarrow (\mathbb{BVAL} \times \mathbb{STORE}) + \{error\}$$

$$\begin{aligned}
\mathcal{R}[\![e]\!]\,\rho = \mathcal{E}[\![e]\!]\,\rho \star \lambda v \sigma.\ & cases\ v\ of \\
& \quad isbval(v) : < v, \sigma >; \\
& \quad isloc(v) : \sigma(v) = unused \rightarrow error, \\
& \qquad\qquad\qquad\qquad < \sigma(v), \sigma >; \\
& \quad isfun(v) : error; \\
& \quad isproc(v) : error \\
& endcases
\end{aligned}$$

Semantics of Expressions

$$\mathcal{E} : \textit{Exp} \longrightarrow \mathbb{ENV} \longrightarrow \mathbb{STORE} \longrightarrow ((\mathbb{EVAL} \times \mathbb{STORE}) + \{\textit{error}\})$$

Constants

$$\mathcal{E}[\![\textbf{true}]\!]\,\rho = \lambda\sigma.\ <\textit{true}, \sigma>$$

$$\mathcal{E}[\![\textbf{false}]\!]\,\rho = \lambda\sigma.\ <\textit{false}, \sigma>$$

$$\mathcal{E}[\![n]\!]\,\rho = \lambda\sigma.\ <n, \sigma>$$

Basic Operations

$$\mathcal{E}[\![\textbf{not}\ e]\!]\,\rho = \mathcal{R}[\![e]\!]\,\rho \star \textit{checkBOOL} \star \lambda b\,\sigma.\ <\neg b, \sigma>$$

$$\mathcal{E}[\![e_1\ \textit{nop}\ e_2]\!]\,\rho = \mathcal{R}[\![e_1]\!]\,\rho \star \textit{checkNAT} \star \lambda n_1 . \mathcal{R}[\![e_2]\!]\,\rho \star \textit{checkNAT} \star \lambda n_2\,\sigma.\ <n_1\ \textit{nop}\ n_2, \sigma>$$

$$\mathcal{E}[\![e_1\ \textit{bop}\ e_2]\!]\,\rho = \mathcal{R}[\![e_1]\!]\,\rho \star \textit{checkBOOL} \star \lambda b_1 . \mathcal{R}[\![e_2]\!]\,\rho \star \textit{checkBOOL} \star \lambda b_2\,\sigma.\ <b_1\ \textit{bop}\ b_2, \sigma>$$

Semantics of Expressions

Conditional expressions

$$\mathcal{E}[\![\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2]\!]\,\rho = \mathcal{R}[\![e]\!]\,\rho \star checkBOOL$$
$$\star\,\lambda b.\, b \to \mathcal{E}[\![e_1]\!]\,\rho, \mathcal{E}[\![e_2]\!]\,\rho$$

Input expressions

$$\begin{aligned}\mathcal{E}[\![\textbf{read}]\!]\,\rho = \lambda\sigma.\, & cases\ \sigma(lin)\ of \\ & \quad v :: in : < v, \sigma[in/lin] >; \\ & \quad nil : error \\ & endcases\end{aligned}$$

Variables interpretation

$$\mathcal{E}[\![x]\!]\,\rho = \lambda\sigma.\, \rho(x) = unbound \to error, < \rho(x), \sigma >$$

Function calls

$$\mathcal{E}[\![e(e')]\!]\,\rho = \mathcal{E}[\![e]\!]\,\rho \star checkFUN \star \lambda f.\, \mathcal{E}[\![e']\!]\,\rho \star \lambda v.\, \lambda\sigma.\, f\ v\ \sigma$$

Semantics of expressions

$$\mathbb{FUN} = \mathbb{NVAL} \longrightarrow \mathbb{STORE} \longrightarrow ((\mathbb{EVAL} \times \mathbb{STORE}) + \{error\})$$

Function Calls

$$\mathcal{E}[\![e(e')]\!]\,\rho = \mathcal{E}[\![e]\!]\,\rho \star checkFUN \star \lambda f.\,\mathcal{E}[\![e']\!]\,\rho \star \lambda v.\,\lambda\sigma.\,f\;v\;\sigma$$

1. To evaluate the argument we use $\mathcal{E}$ and not $\mathcal{R}$, thus we can pass as argument any expressible value, i.e. not only basic values but also locations, procedures or one functions.
2. The environment, ρ, used when calling the function is not used during the actual evaluation of the function. Here only the argument and the state are used. The environment that is used is the one active when the function was defined (static scoping)

Function Declaration

$$\mathcal{D}[\![\textbf{fun}\ f(x); e]\!]\,\rho = \lambda\sigma.\ < \rho_0[(\lambda d.\mathcal{E}[\![e]\!]\,\rho[d/x])/f], \sigma >$$

Semantics of Commands

$$\mathcal{C} : Com \longrightarrow \mathbb{ENV} \longrightarrow \mathbb{STORE} \longrightarrow (\mathbb{STORE} + \{error\})$$

$$\mathcal{C}[\![e := e']\!]\,\rho = \mathcal{E}[\![e]\!]\,\rho \star checkLOC \star \lambda l.\, \mathcal{R}[\![e']\!]\,\rho \star \lambda v\,\sigma.\, \sigma[v/l]$$

$$\mathcal{C}[\![c_1 ; c_2]\!]\,\rho = \mathcal{C}[\![c_1]\!]\,\rho \star \mathcal{C}[\![c_2]\!]\,\rho$$

$$\mathcal{C}[\![\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2]\!]\,\rho = \mathcal{R}[\![e]\!]\,\rho \star checkBOOL \star \lambda b.\, b \rightarrow \mathcal{C}[\![c_1]\!]\,\rho, \mathcal{C}[\![c_2]\!]\,\rho$$

$$\mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!]\,\rho = fix(\lambda \Theta.\, \mathcal{R}[\![e]\!]\,\rho \star checkBOOL \star \lambda b.\, b \rightarrow \mathcal{C}[\![c]\!]\,\rho \star \Theta, \lambda\sigma.\,\sigma$$

$$\mathcal{C}[\![\textbf{output } e]\!]\,\rho = \mathcal{R}[\![e]\!]\,\rho \star \lambda b\,\sigma.\, \sigma[b :: \sigma(lout)/lout]$$

$$\mathcal{C}[\![\textbf{begin } d; c \textbf{ end}]\!]\,\rho = \mathcal{D}[\![d]\!]\,\rho \star \lambda\rho'.\, \mathcal{C}[\![c]\!]\,\rho[\rho']$$

$$\mathcal{C}[\![e(e')]\!]\,\rho = \mathcal{E}[\![e]\!]\,\rho \star checkPROC \star \lambda p.\, \mathcal{E}[\![e']\!]\,\rho \star \lambda v\,\sigma.\, p\,v\,\sigma$$