x = 1

let x = 1 in ...

x(1).

!x(1)

x.set(1)

Programming Language Theory

Denotational Semantics

Ralf Lämmel

Recall: Big-step operational semantics of While language

[ass_{ns}] $\langle x := a,\ s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$

[skip_{ns}] $\langle \texttt{skip},\ s\rangle \to s$

[comp_{ns}] $$\frac{\langle S_1,\ s\rangle \to s',\ \langle S_2,\ s'\rangle \to s''}{\langle S_1;S_2,\ s\rangle \to s''}$$

[$\text{if}^{\text{tt}}_{\text{ns}}$] $$\frac{\langle S_1,\ s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

[$\text{if}^{\text{ff}}_{\text{ns}}$] $$\frac{\langle S_2,\ s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

[$\text{while}^{\text{tt}}_{\text{ns}}$] $$\frac{\langle S,\ s\rangle \to s',\ \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''}{\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

[$\text{while}^{\text{ff}}_{\text{ns}}$] $\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s \text{ if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$

Recall: Small-step operational semantics of While language

$[\text{ass}_{\text{sos}}]$	$\langle x := a,\, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$
$[\text{skip}_{\text{sos}}]$	$\langle \texttt{skip},\, s\rangle \Rightarrow s$
$[\text{comp}^1_{\text{sos}}]$	$\dfrac{\langle S_1,\, s\rangle \Rightarrow \langle S_1',\, s'\rangle}{\langle S_1; S_2,\, s\rangle \Rightarrow \langle S_1'; S_2,\, s'\rangle}$
$[\text{comp}^2_{\text{sos}}]$	$\dfrac{\langle S_1,\, s\rangle \Rightarrow s'}{\langle S_1; S_2,\, s\rangle \Rightarrow \langle S_2,\, s'\rangle}$
$[\text{if}^{\text{tt}}_{\text{sos}}]$	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\, s\rangle \Rightarrow \langle S_1,\, s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{if}^{\text{ff}}_{\text{sos}}]$	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\, s\rangle \Rightarrow \langle S_2,\, s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\text{while}_{\text{sos}}]$	$\langle \texttt{while}\ b\ \texttt{do}\ S,\, s\rangle \Rightarrow$ $\langle \texttt{if}\ b\ \texttt{then}\ (S; \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip},\, s\rangle$

Denotational semantics of While language

$\mathcal{S}_{ds}$: Stm $\rightarrow$ (State $\hookrightarrow$ State)

$\mathcal{S}_{ds}[x := a]s = s[x \mapsto \mathcal{A}[a]s]$

$\mathcal{S}_{ds}[\texttt{skip}] = \mathsf{id}$

$\mathcal{S}_{ds}[S_1; S_2] = \mathcal{S}_{ds}[S_2] \circ \mathcal{S}_{ds}[S_1]$

$\mathcal{S}_{ds}[\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2] =$

$\mathsf{cond}(\mathcal{B}[b], \mathcal{S}_{ds}[S_1], \mathcal{S}_{ds}[S_2])$

Fixed-point combinator

$\mathcal{S}_{ds}[\texttt{while}\ b\ \texttt{do}\ S] = \mathsf{FIX}\ F$

where

$F\ g = \mathsf{cond}(\mathcal{B}[b], g \circ \mathcal{S}_{ds}[S], \mathsf{id})$

More functional as opposed to operational style

Auxiliary operators

$$\text{id } s = s$$

$$(f \circ g)\ s = \begin{cases} f(g\ s) & \text{if } g\ s \neq \text{ undef} \\ & \text{and } f(g\ s) \neq \text{ undef} \\ \text{undef} & \text{otherwise} \end{cases}$$

Partial function composition

$$\text{cond}(p, g_1, g_2)\ s = \begin{cases} g_1\ s & \text{if } p\ s = \text{tt} \\ & \text{and } g_1\ s \neq \text{ undef} \\ g_2\ s & \text{if } p\ s = \text{ff} \\ & \text{and } g_2\ s \neq \text{ undef} \\ \text{undef} & \text{otherwise} \end{cases}$$

"if-then-else" on functions parametrized by a state

Interesting semantics of loops

$\mathcal{S}_{ds}[\texttt{while}\ b\ \texttt{do}\ S] = \mathsf{FIX}\ F$

where

$F\ g = \mathsf{cond}(\mathcal{B}[b],\ g \circ \mathcal{S}_{ds}[S],\ \mathsf{id})$

Compositional definition

$\mathcal{S}_{ds}[\texttt{while}\ b\ \texttt{do}\ S]$

$= \mathcal{S}_{ds}[\texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip}]$

Expectation

$= \mathsf{cond}(\mathcal{B}[b],\ \mathcal{S}_{ds}[S;\ \texttt{while}\ b\ \texttt{do}\ S],\ \mathcal{S}_{ds}[\texttt{skip}])$

Apply semantics for ";"

$= \mathsf{cond}(\mathcal{B}[b],\ \mathcal{S}_{ds}[\texttt{while}\ b\ \texttt{do}\ S] \circ \mathcal{S}_{ds}[S],\ \mathsf{id})$

Match with definition of *F*

$= F(\mathcal{S}_{ds}[\texttt{while}\ b\ \texttt{do}\ S])$

$\mathcal{S}_{ds}[\texttt{while}\ b\ \texttt{do}\ S]$ is a fixed point of F!

Fixed points

$$\mathcal{S}_{ds}[\texttt{while}\ b\ \texttt{do}\ S] = \mathsf{FIX}\ F$$

$$\text{where } F\ g = \mathsf{cond}(\mathcal{B}[b],\ g \circ \mathcal{S}_{ds}[S],\ \mathsf{id})$$

- Type of FIX:

$$\mathsf{FIX}\colon ((\mathsf{State} \hookrightarrow \mathsf{State}) \to (\mathsf{State} \hookrightarrow \mathsf{State})) \to (\mathsf{State} \hookrightarrow \mathsf{State})$$

- Interesting questions:
  - ✦ Will *F* always have a fixed point?
  - ✦ If there are several, which one to choose?

Definition of fixed point

FYI only

Let $f : D \rightarrow D$ be a continuous function on the ccpo $(D, \sqsubseteq)$ with least element $\bot$. Then

$$\text{FIX } f = \bigsqcup\{f^n\bot \mid n \geq 0\}$$

defines an element of D and this element is the least fixed point of f.

?

Remember fixed-point property:
FIX f = f (FIX f)

Chain-complete partially ordered sets (ccpo)

A subset Y of D is called a chain if for any two elements d_1 and d_2 in Y either

$$d_1 \sqsubseteq d_2 \text{ or } d_2 \sqsubseteq d_1$$

$(D, \sqsubseteq)$ is a chain complete partially ordered set (ccpo) if every chain of D has a least upper bound.

?

FYI only

Partially ordered sets

FYI only

A set D with an ordering $\sqsubseteq$ that is

- reflexive
  $d \sqsubseteq d$
- transitive
  $d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_3$ imply $d_1 \sqsubseteq d_3$
- anti-symmetric
  $d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_1$ imply $d_1 = d_2$

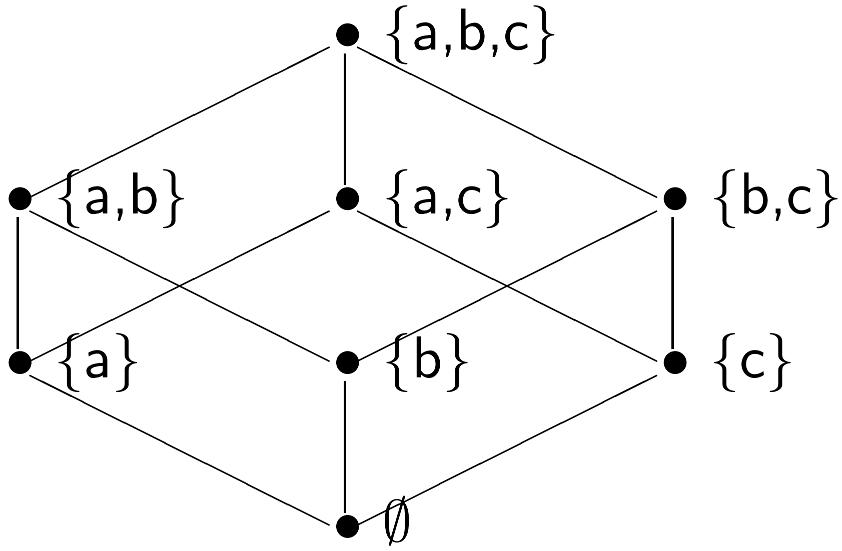
d is a least element of $(D, \sqsubseteq)$ if
$d \sqsubseteq d'$ for all d'.

If $(D, \sqsubseteq)$ has a least element then it is unique and is called $\bot$.

Example for cpo (ccpo, complete lattice)

FYI only

$(\mathcal{P}(\{a,b,c\}), \subseteq)$



Complete lattices

FYI only

Let $(D, \sqsubseteq)$ be a partially ordered set and let $Y \subseteq D$.

d is an upper bound on Y if

$d' \sqsubseteq d$ for all $d' \in Y$

d is a least upper bound on Y if

d is an upper bound on Y

if d' is an upper bound on Y then $d \sqsubseteq d'$.

Complete lattices are ccpos.

Continuos functions

FYI only

Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be ccpo's and consider a (total) function $f : D \to D'$. Then f is continuous if

- f is monotone
- $\bigsqcup'\{f\ d \mid d \in Y\} = f\ (\bigsqcup Y)$

for all non-empty chains Y of D.

?

Monotone functions

FYI only

Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be ccpo's and consider a (total) function

$$f : D \to D'$$

Then f is monotone if

whenever $d_1 \sqsubseteq d_2$ also $f\ d_1 \sqsubseteq' f\ d_2$

Monotone functions

FYI only

Examples

$f_1, f_2 : \mathcal{P}(\{a,b,c\}) \rightarrow \mathcal{P}(\{d,e\})$

X	$f_1\ X$	$f_2\ X$
{a,b,c}	{d,e}	{d}
{a,b}	{d}	{d}
{a,c}	{d,e}	{d}
{b,c}	{d,e}	{e}
{a}	{d}	{d}
{b}	{d}	{e}
{c}	{e}	{e}
$\emptyset$	$\emptyset$	{e}

Exercise: find a non-monotone function!

Definition of fixed point

FYI only

Let $f : D \to D$ be a continuous function on the ccpo $(D, \sqsubseteq)$ with least element $\bot$. Then

$$\text{FIX } f = \bigsqcup\{f^n\bot \mid n \geq 0\}$$

defines an element of D and this element is the least fixed point of f.

Hence, ***if*** the semantic equations construct continuous functions, then the semantics of while loops is well-defined.

What is the relationship between operational and denotational semantics?

FYI only

Recall: SOS

$$(x := a, s) \Rightarrow s[x \mapsto \mathcal{A}[a]s]$$

$$(\texttt{skip}, s) \Rightarrow s$$

$$\frac{(S_1, s) \Rightarrow (S_1', s')}{(S_1; S_2, s) \Rightarrow (S_1'; S_2, s')}$$

$$\frac{(S_1, s) \Rightarrow s'}{(S_1; S_2, s) \Rightarrow (S_2, s')}$$

$$(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s) \Rightarrow (S_1, s)$$

if $\mathcal{B}[b]s = \mathsf{tt}$

$$(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s) \Rightarrow (S_2, s)$$

if $\mathcal{B}[b]s = \mathsf{ff}$

$$(\texttt{while}\ b\ \texttt{do}\ S, s) \Rightarrow$$
$$(\texttt{if}\ b\ \texttt{then}\ (S; \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip}, s)$$

Theorem about equivalence

FYI only

For every statement S of While we have

$$\mathcal{S}_{sos}[S] = \mathcal{S}_{ds}[S]$$

where

$$\mathcal{S}_{sos}[S]\ s = \begin{cases} s' & \text{if } (S, s) \Rightarrow^* s' \\ \text{undefined} & \text{otherwise} \end{cases}$$

Extended While language (While with ***exceptions***)

$$
\begin{array}{rcl}
S & ::= & x := a \mid \texttt{skip} \mid S_1; S_2 \\
 & \mid & \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\
 & \mid & \texttt{while}\ b\ \texttt{do}\ S \\
 & \mid & \texttt{begin}\ S_1\ \texttt{handle}\ e\texttt{:}\ S_2\ \texttt{end} \\
 & \mid & \texttt{raise}\ e
\end{array}
$$

Example

```
begin while true do
      if x < 0
      then  raise  exit
      else  x :=  x - 1
handle  exit:  y :=  7
end
```

How is the semantics modified?

Continuations

- The continuation *c* of a program fragment *S* is the effect of executing the remainder of the program.

$$c \in \mathsf{Cont} = \mathsf{State} \hookrightarrow \mathsf{State}$$

- The continuation for the complete program is the identity function: the remainder of the program is "empty" so the state will not be changed.

Calculating Continuations

Given

$$\cdots \quad ; \quad S \quad ; \underbrace{\quad \cdots \quad}_{c \in \text{ Cont } = \text{ State} \hookrightarrow \text{State}}$$

we want to obtain

$$\cdots \quad ; \underbrace{\quad S \quad ; \quad \cdots \quad}_{c' \in \text{ Cont } = \text{ State} \hookrightarrow \text{State}}$$

Semantic function:

$$\mathcal{S}_{cs}\text{: Stm} \to (\text{Cont} \to \text{Cont})$$

Continuation style

$\mathcal{S}_{cs}$: Stm → (Cont → Cont)

$$\mathcal{S}_{cs}[x := a]\ c\ s = c(s[x \mapsto \mathcal{A}[a]s])$$

$$\mathcal{S}_{cs}[\texttt{skip}] = \mathsf{id}$$

$$\mathcal{S}_{cs}[S_1; S_2] = \mathcal{S}_{cs}[S_1] \circ \mathcal{S}_{cs}[S_2]$$

$$\mathcal{S}_{cs}[\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\ c = \mathsf{cond}(\mathcal{B}[b], \mathcal{S}_{cs}[S_1]c, \mathcal{S}_{cs}[S_2]c)$$

$$\mathcal{S}_{cs}[\texttt{while}\ b\ \texttt{do}\ S] = \mathsf{FIX}\ G$$

where

$$(G\ g)\ c = \mathsf{cond}(\mathcal{B}[b], \mathcal{S}_{cs}[S](g\ c), c)$$

Direct style again (for comparison)

$\mathcal{S}_{ds}$: Stm $\rightarrow$ (State $\hookrightarrow$ State)

$\mathcal{S}_{ds}[x := a]s = s[x \mapsto \mathcal{A}[a]s]$

$\mathcal{S}_{ds}[\texttt{skip}] =$ id

$\mathcal{S}_{ds}[S_1; S_2] = \mathcal{S}_{ds}[S_2] \circ \mathcal{S}_{ds}[S_1]$

$\mathcal{S}_{ds}[\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2] =$
cond($\mathcal{B}[b]$, $\mathcal{S}_{ds}[S_1]$, $\mathcal{S}_{ds}[S_2]$)

$\mathcal{S}_{ds}[\texttt{while } b \texttt{ do } S] =$ FIX F
where
$F\ g =$ cond($\mathcal{B}[b]$, $g \circ \mathcal{S}_{ds}[S]$, id)

Meaning of ";"

$$\mathcal{S}_{ds}[S_1; S_2] = \mathcal{S}_{ds}[S_2] \circ \mathcal{S}_{ds}[S_1]$$

$$\mathcal{S}_{cs}[S_1; S_2] = \mathcal{S}_{cs}[S_1] \circ \mathcal{S}_{cs}[S_2]$$

In direct style, the state transformer of S_1 must be applied first and the one of S_2 second. In continuation style, the meaning of the second statement is the continuation of the first, and hence order is inverted.

Consolidation

How do the two semantics relate to each other?

For all statements *S* of While and all continuations *c* of *Cont*:

$$\mathcal{S}_{cs}[S]c = c \circ \mathcal{S}_{ds}[S]$$

Exceptions

$$\begin{array}{rcl} S & ::= & \cdots \\ & | & \texttt{begin}\ S_1\ \texttt{handle}\ e\texttt{:}\ S_2\ \texttt{end} \\ & | & \texttt{raise}\ e \end{array}$$

- Exception environments
  - ✦ map exception names to their meanings.
  - ✦ the handle statement updates the environment.
  - ✦ the raise statement inspects the environment.
- Semantic function for statements:

$$\mathcal{S}_{cs}\text{: Stm} \rightarrow \text{EEnv} \rightarrow \text{Cont} \rightarrow \text{Cont}$$

Meaning of exceptions

What is the meaning of an exception:

> the effect of executing the rest of the program from the definition point of the exception

i.e.: a continuation!

Exception environment

$$\text{EEnv} = \text{Ename} \to \text{Cont}$$

$$\mathcal{S}_{cs}[x := a]\ eenv\ c\ s = c(s[x \mapsto \mathcal{A}[a]s])$$

$$\mathcal{S}_{cs}[\texttt{skip}]\ eenv = \mathsf{id}$$

$$\mathcal{S}_{cs}[S_1; S_2]\ eenv = (\mathcal{S}_{cs}[S_1]\ eenv) \circ (\mathcal{S}_{cs}[S_2]\ eenv)$$

$$\mathcal{S}_{cs}[\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\ \ eenv\ c = \mathsf{cond}(\mathcal{B}[b],\ \mathcal{S}_{cs}[S_1]\ eenv\ c,\ \mathcal{S}_{cs}[S_2]\ eenv\ c)$$

$$\mathcal{S}_{cs}[\texttt{while}\ b\ \texttt{do}\ S]\ eenv = \mathsf{FIX}\ G$$

where

$$(G\ g)\ c = \mathsf{cond}(\mathcal{B}[b],\ \mathcal{S}_{cs}[S]\ eenv\ (g\ c),\ c)$$

$$\mathcal{S}_{cs}[\texttt{begin}\ S_1\ \texttt{handle}\ e :\ \ S_2\ \texttt{end}]\ eenv\ c = \mathcal{S}_{cs}[S_1]\ (eenv[e \mapsto (\mathcal{S}_{cs}[S_2]\ eenv\ c)])\ c$$

$$\mathcal{S}_{cs}[\texttt{raise}\ e]\ eenv\ c = eenv\ e$$

This slide is derived from the book & slides by Nielson & Nielson: "Semantics with applications" (1991 & 1999+).

Extended While language (While with ***declarations***)

$$
\begin{array}{lcl}
S & ::= & x := a \mid \texttt{skip} \mid S_1; S_2 \\
& \mid & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\
& \mid & \texttt{while } b \texttt{ do } S \\
& \mid & \texttt{begin } D_V\ D_P\ S \texttt{ end} \\
& \mid & \texttt{call } p
\end{array}
$$

$$D_V \quad ::= \quad \texttt{var } x := a;\ D_V \mid \epsilon$$

$$D_P \quad ::= \quad \texttt{proc } p \texttt{ is } S;\ D_P \mid \epsilon$$

Time may be insufficient to deal with this part in detail in the lecture.

How is the semantics modified?

- static scope?
- dynamic scope?

A revised semantic function

$$\mathcal{S}'_{ds}\colon \mathsf{Stm} \rightarrow (\mathsf{Env} \rightarrow (\mathsf{Store} \hookrightarrow \mathsf{Store}))$$

Bindings of variable ids to locations

Bindings of locations to values

Refinement of states to deal with ***scopes***

variables $\longrightarrow$ locations $\longrightarrow$ values

environment store

Environments are ***constructed*** from declarations.

Stores are ***transformed*** by statements.

Locations corresponds to addresses so e.g.
Loc = N

Technically:

$$s \in \text{State} = \text{Var} \rightarrow \text{Z}$$

is replaced by

$$env \in \text{Env} = \text{Var} \rightarrow \text{Loc and}$$

$$sto \in \text{Store} = \text{Loc} \rightarrow \text{Z}$$

All meanings transform the ***same store*** but each meaning be bound to a ***specific environment***.

Denotational semantics with locations

$$\mathcal{S}'_{ds}[x := a]env\ sto = sto[l \mapsto \mathcal{A}[a]\ (\text{lookup}\ env\ sto)]) \quad \text{where } l = env\ x$$

$$\mathcal{S}'_{ds}[\texttt{skip}]\ env = \text{id}$$

$$\mathcal{S}'_{ds}[S_1; S_2]env = (\mathcal{S}'_{ds}[S_2]\ env) \circ (\mathcal{S}'_{ds}[S_1]\ env)$$

$$\mathcal{S}'_{ds}[\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\ env = \text{cond}(\mathcal{B}[b] \circ (\text{lookup}\ env), \mathcal{S}'_{ds}[S_1]env, \mathcal{S}'_{ds}[S_2]env)$$

$$\mathcal{S}'_{ds}[\texttt{while}\ b\ \texttt{do}\ S]\ env = \text{FIX}\ F$$

where

$$F\ g = \text{cond}(\mathcal{B}[b] \circ (\text{lookup}\ env), g \circ (\mathcal{S}'_{ds}[S]env), \text{id})$$

To find a value of a variable:

$$\text{lookup: Env} \rightarrow (\text{Store} \rightarrow \underbrace{(\text{Var} \rightarrow \text{Z})}_{\text{State}})$$

$$\text{lookup}\ env\ sto\ x = sto\ l \quad \text{where } l = env\ x$$

Variable declarations

$D_V ::= \texttt{var}\ x := a;\ D_V \ \mid\ \epsilon$

- updates the environment:
  x is given a new location l
- updates the store:
  l is given the value of a

Two ways to get new locations

1. from the environment:
   - Env = (Var $\rightarrow$ Loc) $\times$ Loc
   - Env = (Var $\cup$ {next}) $\rightarrow$ Loc
2. from the store:
   - Store = (Loc $\rightarrow$ Z) $\times$ Loc
   - Store = (Loc $\cup$ {next}) $\rightarrow$ (Z $\cup$ Loc)

But the semantics are different!

Variable declarations

$\mathcal{D}_V$: $\mathsf{Dec}_V \rightarrow \mathsf{Env} \times \mathsf{Store} \rightarrow \mathsf{Env} \times \mathsf{Store}$

$$\begin{aligned}
&\mathcal{D}_V[\texttt{var}\ x := a;\ D_V](env, sto) = \\
&\quad \mathcal{D}_V[D_V](env[x \mapsto l], \\
&\quad\quad sto[l \mapsto v][\mathsf{next} \mapsto \mathsf{new}\ sto]) \\
&\quad \text{where } l = sto\ (\mathsf{next}) \\
&\quad \text{and } v = \mathcal{A}[a]\ (\mathsf{lookup}\ env\ sto)
\end{aligned}$$

BTW, we abstract from "garbage collection".

$\mathcal{D}_V[\epsilon] = \mathsf{id}$

$\mathcal{S}'_{ds}$: $\mathsf{Stm} \rightarrow \mathsf{Env} \rightarrow \mathsf{Store} \hookrightarrow \mathsf{Store}$

$$\begin{aligned}
&\mathcal{S}'_{ds}[\texttt{begin}\ D_V\ S\ \texttt{end}]\ env\ sto = \\
&\quad\quad \mathcal{S}'_{ds}[S] env'\ sto' \\
&\text{where } (env', sto') = \mathcal{D}_V[D_V](env, sto)
\end{aligned}$$

Procedure declarations

$$D_P ::= \texttt{proc}\ p\ \texttt{is}\ S;\ D_P \ \mid\ \epsilon$$

$$S ::= \cdots \ \mid \texttt{call}\ p$$

Procedure environments:

- map procedure names to their meanings
- are updated by procedure declarations
- are inspected by procedure calls

Semantic function for statements:

$$\mathcal{S}'_{ds}\text{: Stm} \rightarrow \text{Env} \rightarrow \text{PEnv} \rightarrow \text{Store} \hookrightarrow \text{Store}$$

The meaning of procedures

Four choices for meanings of procedures:

- $\mathsf{Env} \to \mathsf{PEnv} \to \mathsf{Store} \hookrightarrow \mathsf{Store}$
- $\mathsf{PEnv} \to \mathsf{Store} \hookrightarrow \mathsf{Store}$
- $\mathsf{Env} \to \mathsf{Store} \hookrightarrow \mathsf{Store}$
- $\mathsf{Store} \hookrightarrow \mathsf{Store}$

"static scope"

But the semantics are different!

$$\mathsf{PEnv} = \mathsf{Pname} \to (\mathsf{Store} \hookrightarrow \mathsf{Store})$$

Procedure declarations

$\mathcal{D}_P$: $\mathsf{Dec}_P \rightarrow \mathsf{Env} \rightarrow \mathsf{PEnv} \rightarrow \mathsf{PEnv}$

$\mathcal{D}_P[\texttt{proc}\ p\ \texttt{is}\ S;\ D_P]\ env\ penv =$

$\quad \mathcal{D}_P[D_P]\ env\ penv[p \mapsto FIX\ F]$

where $F g = \mathcal{S}'_{ds}[S] env\ penv[p \mapsto g]$

$\mathcal{D}_P[\epsilon] env = \mathsf{id}$

$\mathcal{S}'_{ds}$: $\mathsf{Stm} \rightarrow \mathsf{Env} \rightarrow \mathsf{PEnv} \rightarrow \mathsf{Store} \hookrightarrow \mathsf{Store}$

$\mathcal{S}'_{ds}[\texttt{begin}\ D_V\ D_P\ S\ \texttt{end}]\ env\ penv\ sto =$

$\quad \mathcal{S}'_{ds}[S] env'\ penv'\ sto'$

where $(env', sto') = \mathcal{D}_V[D_V](env, sto)$

and $penv' = \mathcal{D}_P[D_P] env'\ penv)$

$\mathcal{S}'_{ds}[\texttt{call}\ p]\ env\ penv = penv\ p$

Scope rules

- Dynamic scope for variables and procedures
- Dynamic scope for variables but static for procedures
- Static scope for variables as well as procedures

```
begin var x := 0;
      proc p is x := x * 2;
      proc q is call p;
      begin var x := 5;
            proc p is x := x + 1;
            call q; y := x
      end
end
```

recapitulation

Option: dynamic scope for variables and procedures

```
begin var x := 0;
      proc p is x := x * 2;
      proc q is call p;
      begin var x := 5;
            proc p is x := x + 1;
            call q; y := x
      end
end
```

recapitulation

- Execution
  - call q
  - call p (calls inner, say local p)
  - x := x + 1 (affects inner, say local x)
  - y := x (obviously accesses local x)
- Final value of y = 6

This slide is derived from the book & slides by Nielson & Nielson: "Semantics with applications" (1991 & 1999+).

Option: dynamic scope for variables static scope for procedures

```
begin var x := 0;
      proc p is x := x * 2;
      proc q is call p;
      begin var x := 5;
            proc p is x := x + 1;
            call q; y := x
      end
end
```

recapitulation

- Execution
  - call q
  - **call p (calls outer, say global p)**
  - x := x * 2 (affects inner, say local x)
  - y := x (obviously accesses local x)
- Final value of y = 10

Option: static scope for variables and procedures

```
begin var x := 0;
      proc p is x := x * 2;
      proc q is call p;
      begin var x := 5;
            proc p is x := x + 1;
            call q; y := x
      end
end
```

recapitulation

- Execution
  - call q
  - call p (calls outer, say global p)
  - **x := x * 2 (affects outer, say global x)**
  - y := x (obviously accesses local x)
- Final value of y = 5

- **Summary**: *Denotational semantics*
  - *Direct style: meanings are state transformers.*
  - *Continuation style: meanings take "rest of program".*
  - *States can be split into environments & locations.*
  - *Denotational semantics are easily written in Haskell.*
- **Prepping**: *"Semantics with applications"*
  - *Chapter on denotational semantics*
- **Outlook**:
  - Program analysis