Semantics of Programming Languages

Denotational Semantics

Prof. Peter Müller

Software Component Technology

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

3. Denotational Semantics

Static Scope Rules

- We extend IMP by blocks with local variable and procedure declarations

```
begin
  var x:= 7;
  proc p begin x := 0 end;
  begin
    var x:= 5;
    call p
  end
end
```

- With **static scope rules**, p updates the **global** variable x
- A simple stack as state is not sufficient

Locations

- Without local variable declarations, we used states that associate a value to each variable:

  $\text{State} : \text{Var} \longrightarrow \text{Val}$

- For dynamic scope rules, we used stacks as states:

  $\text{State} : \textit{stack of}\left(\text{Var} \longrightarrow \text{Val}\right)$

- For static scope rules, we use **locations**
  - A location can be seen as a memory cell
  - A **store** maps locations to values
  - An **environment** maps variables to locations

Stores

- We use the sort $\mathsf{Loc} = \mathbb{Z}$ for locations
  - Think: addresses
- To obtain fresh locations, we need a function
  $new : \mathsf{Loc} \longrightarrow \mathsf{Loc}$
  - We could use the successor function on integers
- A store maps locations to values

$$\mathsf{Store} : \mathsf{Loc} \cup \{\mathsf{next}\} \longrightarrow \mathsf{Val}$$

  - We use the meta-variable $\$$ for stores
  - next is a special token for the **next free location**
  - Since $\mathsf{Val} = \mathsf{Loc} = \mathbb{Z}$, $\$(\mathsf{next})$ is a location

Variable Environments

- We used type environments to associate a type to each variable: Γ : Var $\longrightarrow$ Type
- Similarly a variable environment maps variables to locations

$$\text{Env}_V : \text{Var} \longrightarrow \text{Loc}$$

  - We use the meta-variable Φ_V for variable environments

- The lookup function combines environment and store

$$lookup : \text{Env}_V \longrightarrow \text{Store} \longrightarrow \text{State}$$
$$lookup(\Phi_V, \$) = \$ \circ \Phi_V$$

Extended Semantics of IMP

- With locations, we get a new semantic function
  $\mathcal{S}'_{DS}$: Stm $\rightarrow$ Env$_V$ $\rightarrow$ Env$_P$ $\rightarrow$ $\left(\text{Store} \hookrightarrow \text{Store}\right)$

- `skip`

$$\mathcal{S}'_{DS}[\![\texttt{skip}]\!](\Phi_V, \Phi_P) = id$$

- The clause for assignment uses the environment

$$\mathcal{S}'_{DS}[\![x\texttt{:=}e]\!](\Phi_V, \Phi_P, \$) = \$[\Phi_V(x) \mapsto \mathcal{A}[\![e]\!]lookup(\Phi_V, \$)]$$

- Sequential composition

$$\mathcal{S}'_{DS}[\![s_1\texttt{;}s_2]\!](\Phi_V, \Phi_P) = (\mathcal{S}'_{DS}[\![s_2]\!](\Phi_V, \Phi_P)) \circ (\mathcal{S}'_{DS}[\![s_1]\!](\Phi_V, \Phi_P))$$

Extended Semantics of IMP (cont'd)

- Conditional statement

$$\mathcal{S}'_{DS}[\![\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}]\!](\Phi_V, \Phi_P) =$$
$$cond(\mathcal{B}[\![b]\!] \circ lookup(\Phi_V), \mathcal{S}'_{DS}[\![s_1]\!](\Phi_V, \Phi_P), \mathcal{S}'_{DS}[\![s_2]\!](\Phi_V, \Phi_P))$$

where

$$cond : \quad (\text{Store} \to \text{Bool}) \times (\text{Store} \hookrightarrow \text{Store}) \times (\text{Store} \hookrightarrow \text{Store}) \to$$
$$(\text{Store} \hookrightarrow \text{Store})$$

- Loop

$$\mathcal{S}'_{DS}[\![\texttt{while } b \texttt{ do } s \texttt{ end}]\!](\Phi_V, \Phi_P) = FIX\ F$$

where $F(g) = cond(\mathcal{B}[\![b]\!] \circ lookup(\Phi_V), g \circ \mathcal{S}'_{DS}[\![s]\!](\Phi_V, \Phi_P), id)$

Syntax for Variable Declarations

$$
\begin{array}{lll}
\text{Stm} & = & \ldots \\
 & | & \texttt{'begin'}\ D_V\ s\ \texttt{'end'} \\
\text{Dec}_V & = & \texttt{'var'}\ x\ \texttt{':='}\ e\ \texttt{';'}\ D_V \mid \epsilon
\end{array}
$$

- D_V is a meta-variable for the syntactic category Dec_V of variable declarations

Local Variable Declarations

- The variable environment has to be updated whenever a block is entered

$$\mathcal{D}^V_{DS} : \text{Dec}_V \to \text{Env}_V \times \text{Store} \to \text{Env}_V \times \text{Store}$$

$$\mathcal{D}^V_{DS}[\![\texttt{var}\ x\texttt{:=}e\ \texttt{;}\ D_V]\!](\Phi_V, \$) =$$

$$\mathcal{D}^V_{DS}[\![D_V]\!](\Phi_V[x \mapsto n], \$[n \mapsto v][\text{next} \mapsto new(n)])$$

where $n = \$(\text{next})$ and $v = \mathcal{A}[\![e]\!](lookup(\Phi_V, \$))$

$$\mathcal{D}^V_{DS}[\![\epsilon]\!] = id$$

- For the semantics of blocks, we get

$$\mathcal{S}'_{DS}[\![\texttt{begin}\ D_V\ s\ \texttt{end}]\!](\Phi_V, \$) = \mathcal{S}'_{DS}[\![s]\!](\mathcal{D}^V_{DS}[\![D_V]\!](\Phi_V, \$))$$

Syntax with Procedure Declarations

$$
\begin{array}{lll}
\text{Stm} & = & \texttt{...} \\
 & | & \texttt{'begin'}\ D_V\ D_P\ s\ \texttt{'end'} \\
 & | & \texttt{'call'}\ p \\
\text{Dec}_V & = & \texttt{'var'}\ x\ \texttt{':='}\ e\ \texttt{';'}\ D_V \mid \epsilon \\
\text{Dec}_P & = & \texttt{'proc'}\ p\ \texttt{'begin'}\ s\ \texttt{'end'}\ \texttt{';'}\ D_P \mid \epsilon
\end{array}
$$

- Meta-variables
  - D_P for the syntactic category Dec_P of procedure declarations
  - p for the syntactic category Pname of procedure names

Procedure Environments

- A procedure environment maps procedure names to the effect of executing its body

$$\text{Env}_P : \text{Pname} \longrightarrow \big(\text{Store} \hookrightarrow \text{Store}\big)$$

  - We use the meta-variable Φ_P for procedure environments

Updating Procedure Environments

$$\mathcal{D}^P_{DS} : \mathsf{Dec}_P \to \mathsf{Env}_V \to \mathsf{Env}_P \to \mathsf{Env}_P$$

$$\mathcal{D}^P_{DS}[\![\texttt{proc}\ p\ \texttt{begin}\ s\ \texttt{end;}\ D_P]\!](\Phi_V, \Phi_P) = \mathcal{D}^P_{DS}[\![D_P]\!](\Phi_V, \Phi_P[p \mapsto \mathcal{S}'_{DS}[\![s]\!](\Phi_V, \Phi_P)])$$

$$\mathcal{D}^P_{DS}[\![\epsilon]\!] = id$$

- This function works for **non-recursive** procedures
- The semantics of the procedure body is determined by using the variable and procedure environment of the procedure declaration (static scope rules)

Block Declarations and Calls

- Blocks

$$\mathcal{S}'_{DS}[\![\texttt{begin } D_V\ D_P\ s\ \texttt{end}]\!](\Phi_V, \Phi_P, \$) = \mathcal{S}'_{DS}[\![s]\!](\Phi'_V, \Phi'_P, \$')$$

where $(\Phi'_V, \$') = \mathcal{D}^V_{DS}[\![D_V]\!](\Phi_V, \$)$
and $\Phi'_P = \mathcal{D}^P_{DS}[\![D_P]\!](\Phi'_V, \Phi_P)$

- The semantics of a procedure call is defined by consulting the procedure environment

$$\mathcal{S}'_{DS}[\![\texttt{call } p]\!](\Phi_V, \Phi_P) = \Phi_P(p)$$

Example

- In the final store, the local variable x has value 5 and the global variable x has value 0

```
begin
  var x:= 7;
  proc p begin x := 0 end;
  begin
    var x:= 5;  call p
  end
end
```

Dealing with Recursion

- Semantics for non-recursive procedures

$$\mathcal{D}^P_{DS}[\![\texttt{proc}\ p\ \texttt{begin}\ s\ \texttt{end;}\ D_P]\!](\Phi_V, \Phi_P) = \mathcal{D}^P_{DS}[\![D_P]\!](\Phi_V, \Phi_P[p \mapsto \mathcal{S}'_{DS}[\![s]\!](\Phi_V, \Phi_P)])$$

- We have to ensure that the meaning of all recursive calls in s is the same as that of the procedure p being defined
- For recursive procedures, we need a function g that satisfies $g = \mathcal{S}'_{DS}[\![s]\!](\Phi_V, \Phi_P[p \mapsto g])$
- Again, we need to use fixed points

Semantics of Recursive Procedures

- Declaration of recursive procedures

$$\mathcal{D}^P_{DS}[\![\texttt{proc}\ p\ \texttt{begin}\ s\ \texttt{end;}\ D_P]\!](\Phi_V, \Phi_P) =$$
$$\mathcal{D}^P_{DS}[\![D_P]\!](\Phi_V, \Phi_P[p \mapsto FIXF])$$
$$\text{where } F(g) = \mathcal{S}'_{DS}[\![s]\!](\Phi_V, \Phi_P[p \mapsto g])])$$

- Functions for empty procedure declaration (ϵ) and procedure call stay unchanged

- To show well-definedness of $\mathcal{S}'_{DS}$, we have to prove
  1. that $\mathcal{D}^V_{DS}$ is a well-defined function
  2. that $(\text{Env}_P, \sqsubseteq')$ is a ccpo
  3. that the semantic clauses define continuous functions

3. Denotational Semantics

Exceptions

- We extend IMP by **exceptions**

$$\begin{aligned} \text{Stm} \;=\; & \ldots \\ & \mid \texttt{'begin'}\; s_1\; \texttt{'handle'}\; r : s_2\; \texttt{'end'} \\ & \mid \texttt{'raise'}\; r \end{aligned}$$

- r is a meta-variable for the syntactic category Exception of exceptions
- The `raise` r statement transfers control flow to the handler of exception e

Exceptions: Example

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

- If the example statement is executed from a state σ with $\sigma(\mathrm{x}) > 0$ then it will terminate in a state where $\mathrm{x} = 0$ and $\mathrm{y} = 7$

Remainder of the Program

- With exceptions, the meaning of a statement cannot be defined independently of following statements
- Example 1: What is the meaning of the conditional?

```
if x <= 0 then raise exit
          else x := x -1   end
```

- Example 2: Do we have to execute s_3?

```
if b then s1 else s2 end; s3
```

- We define the meaning of a statement by the effect of executing **the remainder of the program**

Continuations

- A **continuation** is a function c of the domain Cont:

$$\text{Cont} = \text{State} \hookrightarrow \text{State}$$

- For a statement s, we can assume that we have the meaning of the program **after** s, $c(s)$

$$\ldots \; ; \; s \; ; \underbrace{\ldots}_{c}$$

- We want to define the meaning of s **and** the program after s, $c'(s)$

$$\ldots \; ; \underbrace{s \; ; \; \ldots}_{c'}$$

Continuation Style Semantics of IMP

- With continuations, we get a new semantic function $\mathcal{S}'_{CS}$: Stm $\longrightarrow$ Env$_E$ $\longrightarrow$ (Cont $\longrightarrow$ Cont)

- `skip`

$$\mathcal{S}_{CS}[\![\texttt{skip}]\!](\Phi_E) = id$$

  - id is the identity on Cont

- Assignment

$$\mathcal{S}_{CS}[\![x\texttt{:=}e]\!](\Phi_E, c, \sigma) = c(\sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma])$$

Continuation Style Semantics (cont'd)

- Sequential composition

$$\mathcal{S}_{CS}[\![s_1 ; s_2]\!](\Phi_E) = (\mathcal{S}_{CS}[\![s_1]\!](\Phi_E)) \circ (\mathcal{S}_{CS}[\![s_2]\!](\Phi_E))$$

  - The functional composition is **reversed** compared to the direct style semantics
  - The continuations are pulled backwards

- Conditional statement

$$\mathcal{S}_{CS}[\![\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}]\!](\Phi_E, c) = cond(\mathcal{B}[\![b]\!], \mathcal{S}_{CS}[\![s_1]\!](\Phi_E, c), \mathcal{S}_{CS}[\![s_2]\!](\Phi_E, c))$$

Continuation Style Semantics (cont'd)

- Loop

$$\mathcal{S}_{CS}[\![\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}]\!](\Phi_E) = FIX\ F$$
where $F(g)c = cond(\mathcal{B}[\![b]\!], \mathcal{S}_{CS}[\![s]\!](\Phi_E, g(c)), c)$

- If the condition is ff, we simple return the continuation of the remainder of the program
- If the condition is tt, then $g(c)$ denotes the effect of executing the remainder of the loop, followed by the remainder of the program (continuation for first unfolding)

Example

- What is the meaning of the following statement?

```
z:=x; x:=y; y:=z
```

$$\mathcal{S}_{CS}[\![\mathsf{z:=x;\ x:=y;\ y:=z}]\!](\Phi_E, id)$$

$$= (\mathcal{S}_{CS}[\![\mathsf{z:=x}]\!](\Phi_E) \circ \mathcal{S}_{CS}[\![\mathsf{x:=y}]\!](\Phi_E) \circ \mathcal{S}_{CS}[\![\mathsf{y:=z}]\!](\Phi_E))(id)$$

$$= (\mathcal{S}_{CS}[\![\mathsf{z:=x}]\!](\Phi_E) \circ \mathcal{S}_{CS}[\![\mathsf{x:=y}]\!](\Phi_E))(g_1)$$

where $g_1(\sigma) = id(\sigma[\mathsf{y} \mapsto \sigma(\mathsf{z})]) = \sigma[\mathsf{y} \mapsto \sigma(\mathsf{z})]$

$$= \mathcal{S}_{CS}[\![\mathsf{z:=x}]\!](\Phi_E, g_2)$$

where $g_2(\sigma) = g_1(\sigma[\mathsf{x} \mapsto \sigma(\mathsf{y})]) = \sigma[\mathsf{x} \mapsto \sigma(\mathsf{y})][\mathsf{y} \mapsto \sigma(\mathsf{z})]$

$$= g_3$$

where $g_3(\sigma) = g_2(\sigma[\mathsf{z} \mapsto \sigma(\mathsf{x})]) = \sigma[\mathsf{z} \mapsto \sigma(\mathsf{x})][\mathsf{x} \mapsto \sigma(\mathsf{y})][\mathsf{y} \mapsto \sigma(\mathsf{x})]$

Semantics of Exceptions

- An exception environment maps each exception name r to the effect of executing the remainder of the program starting from r's handler

$$\text{Env}_E : \text{Exception} \longrightarrow \text{Cont}$$

  - We use the meta-variable Φ_E for exception environments

- Semantic clauses

$$\mathcal{S}_{CS}[\![\texttt{begin}\ s_1\ \texttt{handle}\ r : s_2\ \texttt{end}]\!](\Phi_E, c) = \mathcal{S}_{CS}[\![s_1]\!](\Phi_E[r \mapsto \mathcal{S}_{CS}[\![s_2]\!](\Phi_E, c)])$$

$$\mathcal{S}_{CS}[\![\texttt{raise}\ r]\!](\Phi_E, c) = \Phi_E(r)$$

Denotational Semantics

- Denotational semantics describes the effect of a computation
  - Direct style: effect of the statement s
  - Continuation style: effect of the remainder of the program, starting from statement s
- A semantic function is defined compositionally for each syntactic construct
- Expressiveness
  - Denotational semantics is well understood for imperative, functional, and logical programming languages
  - Very little work on object-oriented and concurrent languages