Language Engineering: Semantics Lecture 4.

In this unit we shall use partial and total functions, so we won't assume a function is total.

We will use the following notation.

- Let f be a relation from X to Y ie. $f \subseteq X \times Y$
- We will use $f(x)=y$, $(fx)=y$ or, $f(x,y)$ or xfy to mean $(x,y) \in f$
- $f: X \to Y$ is a total function.
- $f: X \hookrightarrow Y$ is a partial function.

Configurations relate syntax, state and semantics

A configuration can have one of two forms:

1. Either it is an intermediate configuration represented by a pair of the form $\langle x, \sigma \rangle$ where x is a syntactic expression and σ is a state.
2. It is a final configuration denoting a semantic value y.

The types of 'y' depend on the syntactic category 'x':

- If $x \in Aexp$ then $y \in \mathbb{Z}$
- If $x \in Bexp$ then $y \in T$
- If $x \in Stm$ then $y \in State$.

An operational semantics is given by a set of rules of the following form.

$$\frac{\delta_1 \to \gamma_1 \quad \dots \quad \delta_n \to \gamma_n}{\delta_0 \to \gamma_0}$$

Here $\delta_1 \to \gamma_1 \dots \delta_n \to \gamma_n$ are the prerequisites. They can be used to derive $\delta_0 \to \gamma_0$.

there are the axioms for the <u>While</u> language.

$$\frac{}{\langle skip, \sigma \rangle \to \sigma}$$

$$\frac{}{\langle x := a, \sigma \rangle \to \sigma[x \to \mathcal{A}[\![a]\!]_\sigma]}$$

$$\frac{\langle S_1, \sigma \rangle \to \sigma' \quad \langle S_2, \sigma' \rangle \to \sigma''}{\langle S_1 ; S_2, \sigma \rangle \to \sigma''}$$

$$\frac{\langle S_1, \sigma \rangle \to \sigma'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \to \sigma'} \text{ if } \mathcal{B}[\![b]\!]_\sigma = tt$$

$$\frac{\langle S_2, \sigma \rangle \to \sigma'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \to \sigma'} \text{ if } \mathcal{B}[\![b]\!]_\sigma = ff$$

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

$$\langle \text{while } b \text{ do } S, \sigma \rangle \to \sigma \quad \text{if } \mathcal{B}[\![b]\!]_\sigma = ff$$

The execution of Statement S in state σ terminates if and only if there exists a state σ' such that $\langle S, \sigma \rangle \to \sigma'$

The execution of Statement S in state σ loops if and only if there exists no state σ' such that $\langle S, \sigma \rangle \to \sigma'$

The Statement S always terminates if and only if it terminates in all states σ.

The Statement S always loops if and only if it loops in all states σ.

Two Statements S_1 and S_2 are semantically equivalent (under N.O.S.) whenever it holds that

$$\langle S, \sigma \rangle \to \sigma' \text{ iff } \langle S_2, \sigma \rangle \to \sigma' \text{ for all } \sigma, \sigma'$$

A natural semantics is deterministic if it holds that

$$\langle S, \sigma \rangle \to \sigma' \text{ and } \langle S, \sigma \rangle \to \sigma'' \text{ imply } \sigma' = \sigma'' \text{ forall } \sigma, \sigma', \sigma''$$

For a deterministic natural semantics we can define a corresponding semantic function.

$$S_{ns} : Stm \to (State \hookrightarrow State)$$

$$S_{ns}[\![S]\!]\sigma = \begin{cases} \sigma' & \text{if } \langle S, \sigma \rangle \to \sigma' \\ \text{undefined} & \text{otherwise} \end{cases}$$

The Semantic function is only guaranteed to return a partial function between states due to the existence of statements whose execution loops in at least 1 state.

$$S_{ns}[\![\text{while true do skip}]\!] = \{\} = \text{undefined}.$$

Language Engineering: Semantics Lecture 5/6

Structural Operational Semantics

In S.O.S. the emphasis is on the individual steps of the execution. The transition relation has the form:

$$\langle S, s \rangle \Rightarrow \gamma$$

Where γ is of the form $\langle S', s' \rangle$ or s'.

The transition expresses the first step of the execution of S from the state s.

4 Continues on Page 13.

-Backto Page4.

If γ is of the form:

- $\langle S', s' \rangle$ then the execution of S from s has not completed. The remaining computation is expressed by the intermediate configuration $\langle S', s' \rangle$
- s', then the execution of S from s has terminated and the final state is s'.

If there is no such γ then we say that $\langle S, s \rangle$ is stuck.

* For the structural operational semantics of while see table 2.2 on page 33 of text book.

A derivation sequence of a statement S from a state s is either:

- A finite sequence.

$\gamma_0, \gamma_1, \ldots, \gamma_k$ with $\langle S, s \rangle = \gamma_0$ and $\gamma_i \Rightarrow \gamma_{i+1}$ for $0 \leq i < k$, $k \geq 0$.

here γ_k is either a terminating configuration or a stuck configuration.

- An infinite sequence

$\gamma_0, \gamma_1, \gamma_2, \ldots$ with $\langle S, s \rangle = \gamma_0$ and $\gamma_i \Rightarrow \gamma_{i+1}$ for $i \geq 0$

As notation we shall use:

- $\gamma_0 \Rightarrow^i \gamma_i$ to indicate that there are i steps in the execution from γ_0 to γ_i.
- $\gamma_0 \Rightarrow^* \gamma_i$ to indicate that there is a finite number of steps from γ_0 to γ_i.

Note: $\gamma_0 \Rightarrow^i \gamma_i$ and $\gamma_0 \Rightarrow^* \gamma_i$ are not neccesarily derivation sequences. They will only be such if γ_i is a terminating or stuck configuration.

Given a Statement S and state s it is always possible to find atleast 1 derivation sequence that starts at $\langle S, s \rangle$.

- Simply apply axioms and rules forever or until a terminal or stuck configuration is reached.
- While does not have any stuck configurations.

We shall say that an execution of S on a state s:

- Terminates if there is a finite derivation sequence starting with $\langle S, s \rangle$.
- Loops if there is an infinite derivation sequence starting with $\langle S, s \rangle$
- Terminates successfully if $\langle S, s \rangle \Rightarrow^* s'$ for some state s' (In While a statement terminates successfully iff it terminates, since there are no stuck states).
- Always loops if it loops on all states.
- Always terminates if it terminates on all states.

Properties of Semantics

For S.O.S. it is often useful to be able to conduct proofs by induction on the length of the derivation sequences.

The steps are as follows:

1. Prove that the property holds for all sequences of length 0.
2. Prove that the property holds for all other derivation sequences. Assume the property holds for all sequences of length k, show it holds for derivation sequences of length k+1.

The inductive step of a proof will often be done by inspecting either:

- The structure of the syntactic element
- The derivation tree validating the first transition of the derivation sequence.

The semantic function S_{sos}

For a deterministic structural operational semantics we can define a semantic function as follows:

$$S_{sos} : Stm \to (State \hookrightarrow State)$$

$$S_{sos}[\![S]\!]_s = \begin{cases} s' & \text{if } \langle S, s\rangle \Rightarrow^* s' \\ \text{undef} & \text{otherwise.} \end{cases}$$

An equivalence result

Theorem

For every statement S in While $S_{ns}[\![S]\!]_s = S_{sos}[\![S]\!]_s$

To prove this we need two Lemmas:

1. For every statement S of While and states s and s'. we have:

   $\langle S, s\rangle \to s'$ implies $\langle S, s\rangle \Rightarrow^* s'$.

2. For every statement S of while and states s and s'. we have:

   $\langle S, s\rangle \Rightarrow^k s'$ implies $\langle S, s\rangle \to s'$

Using the two lemmas for an arbitrary Statement S and states, it follows that if $S_{ns}[\![S]\!]_s = s'$ then $S_{sos}[\![S]\!]_s = s'$ and vice versa.

Hence $S_{ns} \bullet [\![S]\!]_s = S_{sos}[\![S]\!]_s$

Language Engineering: Semantics - Lecture 7

Provably Correct Implementation

The Abstract Machine

When specifying an abstract machine it is usually convenient to first preset its configurations and next its instructions and their meanings.

The abstract machine AM has configurations in the form $\langle c, e, s \rangle$ where:

- `c` is the sequence of instructions to be executed.
- `e` is the evaluation stack.
- `s` is the storage.

We use the stack for evaluating arithmetic and boolean expressions. Formally it is a list of values so,

$$Stack = (\mathbb{Z} \cup T)^*$$

We ~~also~~ have $e \in Stack$.

For simplicity we assume that the storage is similar to the State, so $s \in State$. and it is used to hold the variables and their values.

The instructions of AM are given by the abstract Syntax.

```
inst ::= PUSH-n | ADD | MULT | SUB
       | TRUE | FALSE | EQ | LE | AND | NEG
       | FETCH-x | STORE-x | NOOP | BRANCH(c,c)
       | LOOP(c,c)
  c ::= ε | inst:c
```

 continues on Page 19

← Back to Page 16

We shall write Code for the syntactic category of sequences, so c is a meta-variable ranging over Code.

$\langle c, e, s \rangle \in Code \times Stack \times State$.

A configuration is terminal if the code component is the empty sequence: eg $\langle \varepsilon, e, s \rangle$.

The semantics of the instructions is given by an operational semantics. So there will need to be a transition system, this will be Δ.

$\langle c, e, s \rangle \; \Delta \; \langle c', e', s' \rangle$

See axioms in table 3.1.

We shall use the following example.

PUSH-1 : FETCH-X : ADD : STORE-X.

Assuming the initial storage s has $sx = 3$,

$\langle$PUSH-1 : FETCH-X : ADD : STORE-X, ε, s$\rangle$

$\Delta \langle$FETCH-X : ADD : STORE-X, 1, s$\rangle$

$\Delta \langle$ADD : STORE-X, 3 : 1, s$\rangle$

$\Delta \langle$STORE-X, 4, s$\rangle$

$\Delta \langle \varepsilon, \varepsilon, s[x \mapsto 4] \rangle$

The execution function $\mathcal{M}$

$\mathcal{M} : Code \to (State \hookrightarrow State)$

$$\mathcal{M}[\![c]\!]_s = \begin{cases} s' & \text{if } \langle c, \varepsilon, s \rangle \; \Delta^* \; \langle \varepsilon, e, s' \rangle \\ \text{undef} & \text{otherwise} \end{cases}$$

Specification of the translation

Expressions

Arithmetic and boolean expressions are evaluated on the stack and the code generated must effect this.

This is accomplished by the functions.

CA: Aexp → Code

CB: Bexp → code.

see table 3.2 for the translation

Statements

CS: Stm → Code.

see table 3.3 for the translation.

The semantic function Sam

$Sam : Stm \to (State \hookrightarrow State)$

$Sam[\![S]\!] = (M \cdot CS)[\![S]\!]$

Language Engineering: Semantics - Lecture 9.

- Label your quantifiers clearly as (1), (2), ...
- When carrying out proofs label which quantifiers you are using.

 continues on Page 27

← Back to Page 20.

Denotational Semantics - Direct Style.

Here is the denotational semantics of While.

$S_{ds} : Stm \to (State \hookrightarrow State)$

$S_{ds}[\![x := a]\!]s = s[x \mapsto \mathcal{A}[\![a]\!]s]$

$S_{ds}[\![skip]\!] = id$ or $S_{ds}[\![skip]\!]s = s$

$S_{ds}[\![S_1; S_2]\!] = S_{ds}[\![S_2]\!] \cdot S_{ds}[\![S_1]\!]$

↑ function composition.

$S_{ds}[\![\text{if } b \text{ then } S_1 \text{ else } S_2]\!] = cond(B[\![b]\!], S_{ds}[\![S_1]\!], S_{ds}[\![S_2]\!])$

$S_{ds}[\![\text{while } b \text{ do } S_1]\!] = FIX\ F$

where $Fg = cond(B[\![b]\!], g \cdot S_{ds}[\![S]\!], id)$

$cond : (State \to T) \times (State \hookrightarrow State) \times (State \hookrightarrow State)$

$\to (State \hookrightarrow State).$

$$cond(p, g_1, g_2)s = \begin{cases} g_1 s & \text{if } ps = tt \\ g_2 s & \text{if } ps = ff. \end{cases}$$

(i) cond is like TBL fix

$FIX : ((State \hookrightarrow State) \to (State \hookrightarrow State)) \to (State \hookrightarrow State)$

this is a functional.

Language Engineering: Semantics 4 - Lecture 10.

Partial Orders

- A weak partial order is
  - Reflexive, $a \leq a$ $\forall a$.
  - transitive, $a \leq b \wedge b \leq c \Rightarrow a \leq c$. $\forall a, b, c$
  - antisymmetric, $a \leq b \wedge b \leq a \Rightarrow a = b$. $\forall a, b$.

continues on Page 32.

←Back to Page 27

A strong partial order is:
- Irreflexive $a \not\leq a$ for no a.
- Transitive
- And hence anti symmetric.

A total Order is:
- Connex $a \leq b$ or $b \leq a$ for all a, b.
- Hence reflexive
- Transitive
- Anti symmetric.

Language Engineering: Semantics - Lecture II.

The solution to the problems will be to ~~use~~ impose requirements on the fix points and show there is at most one of them, fulfilling these requirements.

All functionals originating from statements in While. do have a fixed point that satisfies these requirements.

To motivate our choice of requirements we consider the execution of a statement

'while b do S' from a state S_0.

There are 3 possible outcomes:

1. The loop terminates.
2. ~~the~~ It loops locally, ie there ~~is~~ is a loop inside the statement S.
3. It loops globally, the While loops.

We will now investigate the fix point in each outcome.

Continues on page 41.

← Back to page 32.

Out come 1 : It terminates.

Here 'while b do s' terminates from state s_0. This means that there exists a state s_n that $B[\![b]\!]_{s_n} = ff$.
More formally,

$$B[\![b]\!]_{s_i} = \begin{cases} tt & \text{if } i < n \\ ff & \text{if } i = n \end{cases}$$

and,

$$S_{ds}[\![S]\!]_{s_i} = s_{i+1} \quad \text{for all } i < n.$$

Let g_0 be a fixpoint of F, assume $Fg_0 = g_0$
In the case where $i < n$,

$$\begin{aligned} g_0 s_i &= (F g_0)\, s_i \\ &= cond(B[\![b]\!], g_0 \cdot S_{ds}[\![S]\!], id)\, s_i \\ &= g_0\, (S_{ds}[\![S]\!]_{s_i}) \\ &= g_0\, s_{i+1} \end{aligned}$$

In the case where $i = n$,

$$\begin{aligned} g_0 s_n &= (F g_0) s_n \\ &= cond\,(B[\![b]\!], g_0 \cdot S_{ds}[\![S]\!], id)\, s_n \\ &= id\, s_n \\ &= s_n \end{aligned}$$

Thus $g_0 s_0 = g_0 s_1 = \dots = g_0 s_n = s_n$.

Outcome 2 It loops locally.

This means that there are states $s_1, \ldots, s_n$ such that,

$$B[\![b]\!]_{s_i} = tt \quad \text{for all } 0 \le i < n$$

and,

$$S_{ds}[\![S]\!]_{s_i} = \begin{cases} s_{i+1} & \text{if } i < n \\ \underline{\text{undef}} & \text{if } i = n \end{cases}$$

Let g_0 be any fixpoint of F, $g_0 = F g_0$

In the case $i < n$,

$$\begin{aligned} g_0\, s_i &= (F g_0)\, s_i \\ &= \text{Cond}(B[\![b]\!], g_0 \cdot S_{ds}[\![S]\!], id)\, s_i \\ &= g_0 \cdot S_{ds}[\![S]\!]\, s_i \\ &= g_0 (S_{ds}[\![S]\!]\, s_i) \\ &= g_0\, s_{i+1}. \end{aligned}$$

In the case $i = n$,

$$\begin{aligned} g_0\, s_n &= (F g_0)\, s_n \\ &= \text{Cond}(B[\![b]\!], g_0 \cdot S_{ds}[\![S]\!], id)\, s_n \\ &= g_0 (S_{ds}[\![S]\!]\, s_n) \\ &= g_0\, \underline{\text{undef}} \end{aligned}$$

Hence any fixpoint g_0 of F will satisfy $g_0 s_0 = g_0 s_1 = \cdots = g_0 s_n = \underline{\text{undef}}$.

Outcome 3 it loops globally.

Here, this means there are infinitely many states $s_1, \ldots$ such that,

$B[\![S]\!]s_i = tt$ for all i.

and,

$S_{as}[\![S]\!]s_i = s_{i+1}$ for all i.

Let g_0 be any fixpoint of F. $Fg_0 = g_0$

As in the previous cases we get,

$$g_0 s_i = g_0 s_{i+1}$$

for all $i \geq 0$.

Hence $g_0 * s_0 = g_0 s_i$ for all i.

Here we cannot determine the value of $g_0 s_0$ in this way. This is the situation where fixed points of F may differ.

Generalising this experience leads to the following requirement:

The desired fixed point $FIX\ F$ should be sompartial function g_0: State $\hookrightarrow$ State such that,

- g_0 is a fixed point of F, $Fg_0 = g_0$
- If g is another fixed point of F, $Fg = g$ then $g_0 s = s' \Rightarrow g\, s = s'$. for all choices of s and s'.

Fixed Point Theory

To prepare for a frame work that guarantees the existence of the desired fixed point FIX ∈ F, we shall reformulate the requirements,

The first step will be to formalize the requirement that FIX F shares its results with other fixed points.

To do so we define an ordering $\sqsubseteq$ on partial functions of State ↪ State. We set,

$g_1 \sqsubseteq g_2$

when the partial function g_1: State ↪ State shares its results with the partial function g_2: State ↪ State in the sense that,

if $g_1\, s = s'$ then $g_2\, s = s'$

for all choices of s and s'.

(State ↪ State, $\sqsubseteq$) is a partial order.
The partial function $\perp$: State ↪ State.
defined by,

$\perp\, s$ = undef for all s

is the least element of State ↪ State.

Having introduced an ordering on the partial functions we can now give a more precise statement of the requirements of FIX F:

- FIX F is a fixed point of F. $F(FIX\,F) = FIX\,F$.
- FIX F is a least fixed point of F, that is

  if $Fg = g$ then $FIX\,F \sqsubseteq g$.

Complete Partially ordered Sets,

Consider a partially ordered set $(D, \sqsubseteq)$ and assume we have a subset Y of D.

We are interested in an element of D, that summarises all the information of Y. This is called the upper bound of Y. formally,

it is an element $d \in D$ such that,

$\forall d' \in Y, d' \sqsubseteq d$.

An upper bound d of Y is a least upper bound if and only if,

d' is an upper bound of Y implies that $d \sqsubseteq d'$

Thus the least upper bound of Y will add as little extra information as possible to that already present in the elements of Y.

If Y has a unique ~~four~~ least upper bound d, we denote it using $d = \bigsqcup Y$.

Y is called a chain if it is consistent in the sense that if we take any two elements of Y then one will share its information with the other, formally,

$\forall d_1, d_2 \in Y, d_1 \sqsubseteq d_2$ or $d_2 \sqsubseteq d_1$.

A partially ordered set $(D, \sqsubseteq)$ is called, a chain complete partially ordered set (ccpo) whenever $\bigsqcup Y$ exists for all chains Y.

It is a complete lattice if $\bigsqcup Y$ exists for all subsets Y of D.

Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be ccpo's and consider the total function $f: D \to D'$.

If $d_1 \sqsubseteq d_2$ then the intuition is that d_1 shares information with d_2. So when the function f has been applied to the 2 elements, we expect a similar relationship to hold.

When this is the case we say f is <u>monotone</u>, formally,

f is monotone iff $d_1 \sqsubseteq d_2$ implies $fd_1 \sqsubseteq' fd_2$.

for all choices $d_1, d_2 \in D$.

We say that a function ~~f~~ $f: D \to D'$ defined on ccpo's $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ is continuous if it is,

monotone

and,

$\bigsqcup' \{fd \mid d \in Y\} = f(\bigsqcup Y)$.

Holds for all non-empty chains Y.

If it also holds for the empty chain, $\bot = f\bot$ then we say, f is strict.

We can now define the required fixed point operator FIX:

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.

 Continues on page 52

← Back to Page 46.

Example $\langle P(\{a,b,c\}), \subseteq \rangle$

$\{\emptyset, \{a\}, \{a,c\}\}$ is a chain. with least upper bound $\{a,c\}$.

$\{\emptyset, \{a\}, \{c\}\}$ is not a chain but still has a least upper bound $\{a,c\}$,

in general here $\sqcup = \cup$ ← union.

This makes it a <u>complete Lattice</u>.

Now consider the infinite ccpo $\langle P(\mathbb{N} \cup \{a\}), \subseteq \rangle$ and determine if $f : P(\mathbb{N} \cup \{a\}) \to P(\mathbb{N} \cup \{a\})$ is monotone and/or continuous.

~~f is monotone~~.

$$f(X) = \begin{cases} X & \text{if } X \text{ is finite} \\ X \cup \{a\} & \text{if } X \text{ is infinite.} \end{cases}$$

f is monotone

f is not continuous.

We can also show $\langle A \hookrightarrow B, \subseteq \rangle$ is a ccpo by generalising the proof that $\langle State \hookrightarrow State, \subseteq \rangle$ is a ccpo.
it is however not a complete lattice.

<u>Fix Point Theorem</u>

if $f : D \to D$ is a continuous function on ccpo $\langle D, \sqsubseteq \rangle$ with least element $\bot$.

Then the element $FIX\ F = \bigsqcup \{f^n(\bot) \mid n \geq 0\} \in D$.

1. first we show that set $Y = \{f^n(\bot) \mid n \geq 0\}$
$= \{f^0(\bot), f^1(\bot), \ldots\} \subseteq D$. is a non-empty chain.

It is non-empty since it contains $\bot$.
$f^n(\bot) \sqsubseteq f^{n+1}(\bot)$. for all $n \geq 0$,
Show by weak induction.

2. Next we show the least upper bound of Y is also the LUB of Y's image under f.

$$\begin{aligned}
\bigsqcup Y &= \bigsqcup \{f^n(\bot) \mid n \geq 0\} \\
&= \bigsqcup \{f^n(\bot) \mid n > 0\} \cup \{f^0(\bot)\} \\
&= \bigsqcup \{f^n(\bot) \mid n > 0\} \cup \{\bot\} \\
&= \bigsqcup \{f^n(\bot) \mid n > 0\} \\
&= \bigsqcup \{f(f^{n-1}(\bot)) \mid n > 0\} \\
&= \bigsqcup \{f(f^{n'}(\bot)) \mid n' \geq 0\}. \\
&= \bigsqcup \{f(y) \mid y \in Y\}.
\end{aligned}$$

3. Now show FIX F is a fixpoint of f.

$$\begin{aligned}
f(\mathrm{Fix} f) &= f(\bigsqcup Y) \\
&= \bigsqcup \{f(y) \mid y \in Y\} = \bigsqcup Y = \mathrm{Fix} f.
\end{aligned}$$

4. Finally we show FIX F is the least fixpoint of f.
This is because $f^n(\bot) \sqsubseteq f^n(d)$ for all $n \geq 0$ and $d \in D$, follows by weak induction on n for any arbitrary d.

So $f^n(\bot) \sqsubseteq d$ for any fixpoint d of f.

Thus any fixpoint d is an upper bound of the chain Y. But $\mathrm{FIX} f \sqsubseteq d$ as FIX f is the LUB of Y.

This means FIX f is the least fixpoint of f.

Continues on page 64

← back to Page 53

Language Engineering: ~~Parsers - Lecture~~ Semantics - Abstract Machine

Here are the axioms for operational Semantics of an Abstract Machine.

$$\langle \text{Store-x}:c, z:e, s\rangle \triangleright \langle c, e, s[x \mapsto z]\rangle$$
$$\langle \text{Fetch-x}:c, e, s\rangle \triangleright \langle c, s x : e, s\rangle$$
$$\langle \text{Push-n}:c, e, s\rangle \triangleright \langle c, \mathcal{N}[\![n]\!]:e, s\rangle$$
$$\langle \text{Add}:c, x:y:e, s\rangle \triangleright \langle c, (x+y):e, s\rangle \quad \text{if } z_1, z_2 \in \mathbb{Z}.$$
$$\langle \text{Sub}:c, x:y:e, s\rangle \triangleright \langle c, (x-y):e, s\rangle \quad \text{if } x, y \in \mathbb{Z}$$
$$\langle \text{Mult}:c, x:y:e, s\rangle \triangleright \langle c, (x*y):e, s\rangle$$
$$\langle \text{True}:c, e, s\rangle \triangleright \langle c, tt:e, s\rangle$$
$$\langle \text{False}:c, e, s\rangle \triangleright \langle c, ff:e, s\rangle$$
$$\langle \text{EQ}:c, x:y:e, s\rangle \triangleright \langle c, (x=y):e, s\rangle$$
$$\langle \text{LE}:c, x:y:e, s\rangle \triangleright \langle c, (x \le y):e, s\rangle$$
$$\langle \text{And}:c, x:y:e, s\rangle \triangleright \begin{cases} \langle c, tt:e, s\rangle & \text{if } x = tt \text{ and } y = tt \\ \langle c, ff:e, s\rangle & \text{otherwise.} \end{cases}$$
$$\langle \text{Neg}:c, x:e, s\rangle \triangleright \begin{cases} \langle c, tt:e, s\rangle & \text{if } x = ff \\ \langle c, ff:e, s\rangle & \text{otherwise} \end{cases}$$
$$\langle \text{Noop}:c, e, s\rangle \triangleright \langle c, e, s\rangle$$
~~$\langle \text{Branch}:c, b:a:b:c:e, s\rangle \triangleright \{\langle b:c, e, s\rangle$~~

$$\langle \text{Branch}(c_1, c_2):c, t:e, s\rangle \triangleright \begin{cases} \langle c_1:c, e, s\rangle & \text{if } t = tt \\ \langle c_2:c, e, s\rangle & \text{otherwise} \end{cases}$$

~~$\langle \text{Loop}(c_1, c_2):c, b:e, s\rangle \triangleright \{\langle c_1 : \text{Loop}(c_1, c_2):c, e, s\rangle$ if $t = tt$~~

$$\langle \text{Loop}(c_1, c_2):c, e, s\rangle \triangleright \langle c_1 : \text{Branch}(c_2 : \text{Loop}(c_1, c_2), \text{Noop}):c, e, s\rangle$$

To be able to use this with While we need a way to convert While to the abstract syntax. To define this we will define CA, CB, CS to convert Aexp, Bexp, and Stm respectively to ~~the~~ abstract syntax

$$CA : Aexp \to Code$$
$$CA[\![n]\!] = \text{Push } -n$$
$$CA[\![x]\!] = \text{Fetch} - x$$
$$CA[\![x*y]\!] = CA[\![y]\!] : CA[\![x]\!] : \text{Mult}$$
$$CA[\![x+y]\!] = CA[\![y]\!] : CA[\![x]\!] : \text{Add}.$$
$$CA[\![x-y]\!] = CA[\![y]\!] : CA[\![x]\!] : \text{Sub}$$

$CB : Bexp \to Code.$

$CB[\![true]\!] = True.$

$CB[\![false]\!] = False$

$CB[\![x \,\&\&\, y]\!] = CB[\![y]\!] : CB[\![x]\!] : And.$

$CB[\![x = y]\!] = CB[\![y]\!] : CB[\![x]\!] : EQ$

$CB[\![x \leq y]\!] = CB[\![y]\!] : CB[\![x]\!] : LE$

$CB[\![\neg x]\!] = CB[\![x]\!] : Neg.$

$CS : Stm \to Code.$

$CS[\![skip]\!] = Noop$

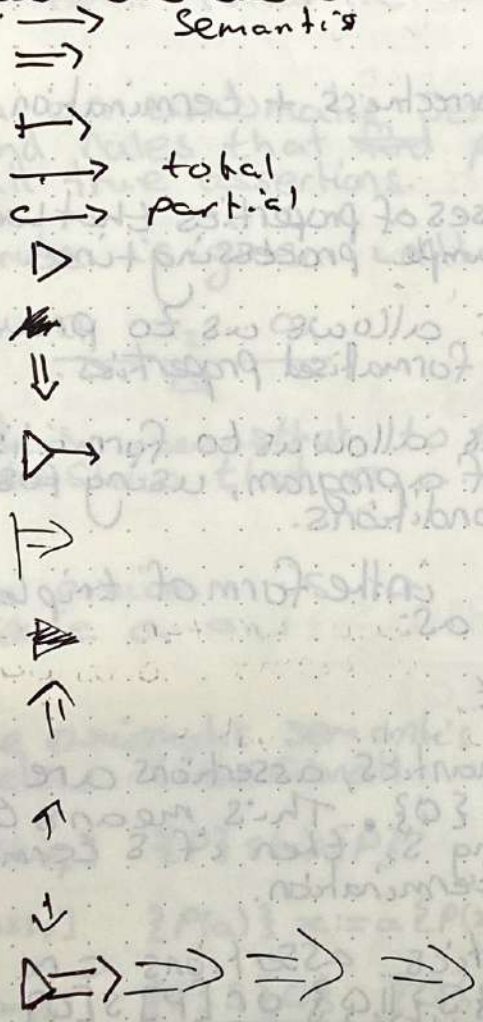
$CS[\![x := a]\!] = CA[\![a]\!] : Store\text{-}x$

$CS[\![S_1 ; S_2]\!] = CS[\![S_1]\!] : CS[\![S_2]\!]$

$CS[\![\text{if } b \text{ then } S_1 \text{ else } S_2]\!] = CB[\![b]\!] : Branch(CS[\![S_1]\!], CS[\![S_2]\!])$

$CS[\![\text{while } b \text{ do } S]\!] = Loop(CB[\![b]\!], CS[\![S]\!]).$

These are the arrows of Language Engineering:



Continues on page 66.

← back to page 65

Language Engineering: Semantics - Lecture 15

Axiomatic Semantics

There are 2 different classes of properties:

- Partial Correctness Properties: These are properties that will hold true if the program terminates. The properties give a relationship between the initial and final values of the variables in the program.
- Total Correctness Properties: This contains all the partial correctness properties, but also ensures that it terminates.

This can be seen below:

total correctness = partial correctness + termination.

There are also other classes of properties that we will not look at, for example processing time.

An axiomatic semantics allows us to prove that a program satisfies formalised properties.

The axiomatic semantics allow us to formalise the interesting properties of a program, using preconditions and postconditions.

These properties are given in the form of triples which can be wrote as:

Pre Program Post.

In partial correctness semantics, assertions are given in the form $\{P\}\ S\ \{Q\}$. This means that if P holds before running S, then if S terminates then Q will hold after termination.

In total correctness semantics assertions are given in the form $\{P\}\ S\ \{\Downarrow Q\}$ or $[P]\ S\ [Q]$. This means that if P holds, then S will terminate, and Q will hold when S terminates.

When Specifying the pre and postconditions there are two different approaches we can take:

- Intensional Approach:
  This is the idea of introducing an explicit language called an assertion language, the conditions of this language will be the formulae of the language.

  In general, the assertion language is much more powerful, than the Bexp given in While.

- Extensional Approach:
  This is the idea that the conditions are predicates, that is a function in State → T.

~~we will follow the extensional approach~~.

In an axiomatic semantics there are axioms and rules that ~~find~~ provide a method for finding all true assertions.

A rule is given in the form:

$$\frac{a_1 \quad a_2 \cdots a_n}{a}$$

This means that to show a is true we have to show that $a_1 \cdots a_n$ is true.

An axiom is a rule with ~~not the~~ no premises, ie. $a \cdots a_n$.

The axiomatic semantics of While are given below: (intensional).

$[Skip_p]$ $\{P\}\ skip\ \{P\}$

$[Ass_p]$ $\{P(a)\}\ x := a\ \{P(x)\}$.

$[comp_p]$ $$\frac{\{P\}\ S_1\ \{Q\} \quad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1 ; S_2\ \{R\}}$$

$$[if_p] \quad \frac{\{P \wedge b\}\, S_1\, \{Q\} \qquad \{P \wedge \neg b\}\, S_2\, \{Q\}}{\{P\}\ \text{if } b \text{ then } S_1 \text{ else } S_2\ \{Q\}.}$$

$$[while_p] \quad \frac{\{P \wedge b\}\, S\, \{P\}}{\{P\}\ \text{while } b \text{ do } S\ \{P \wedge \neg b\}}$$

$$[cons_p] \quad \frac{\{P'\}\, S\, \{Q'\}}{\{P\}\, S\, \{Q\}} \quad \text{if } P \overset{F}{\Rightarrow} P' \text{ and } Q' \overset{F}{\Rightarrow} Q.$$

In the while rule the precondition P is also called the loop incariant. Hence P must be maintained on every iteration of the loop.

Language Engineering: Semantics - Lecture 16.

Here are the Extensional Axiomatic semantics:

$$\{P\}\ skip\ \{P\}.$$

$$\{P[x \mapsto A[\![a]\!]]\}\ x := a\ \{P\}.$$

$$\frac{\{P\}\, S_1\, \{R\} \qquad \{R\}\, S_2\, \{Q\}}{\{P\}\ S_1 ; S_2\ \{Q\}.}$$

$$\frac{\{P \wedge B[\![b]\!]\}\, S_1\, \{Q\} \qquad \{P \wedge \neg B[\![b]\!]\}\, S_2\, \{Q\}}{\{P\}\ \text{if } b \text{ then } S_1 \text{ else } S_2\ \{Q\}.}$$

$$\frac{\{P \wedge B[\![b]\!]\}\, S\, \{P\}}{\{P\}\ \text{while } b \text{ do } S\ \{P \wedge \neg B[\![b]\!]\}.}$$

$$\frac{\{P'\}\, S\, \{Q'\}}{\{P\}\, S\, \{Q\}.} \quad \text{if } P \Rightarrow P' \text{ and } Q' \Rightarrow Q.$$

An example Axiomatic Proof:

$$\frac{\{n! = (y*x)(x-1)\ \ y := y*x\ \{n! = y(x-1)!\}}{\{n! = yx!\ \&\ \neg(x=1)\}\ y := y*x\ \{n! = y(x-1)!\}\quad \{n! = y(x-1)!\}\ x := x-1\ \{n! = yx!\}}$$

$$\frac{\{n! = yx!\ \&\ \neg(x=1)\}\ y := y*x;\ x := x-1\ \{n! = yx!\}}{\textcircled{1}\quad \{n! = yx!\}\ \text{while } \neg(x=1) \text{ do } (y := y*x;\ x := x-1)\ \{n! = yx!\ \&\ \neg(x=1)\}}$$

$$\{y=1\ \&\ x=n\ \&\ n>0\}\ \text{while } \neg(x=1) \text{ do } (y := y*x;\ x := x-1)\ \{y = n!\}.$$

① first find the loop invariant, this can be difficult A good way is to observe some test values, ~~is a good~~ to find an invariant.

#	0	1	2	3	4
x	5	4	3	2	1
y	1	5	20	60	120

this gives the invariant $n! = x!y$

② $n! = yx!\ \&\ \neg(x=1) \vDash y = n!$

1. $n! = yx!$
2. $\neg(x=1)$
3. $x=1$ from 2.
4. $1! = 1.0! = 1.1 = 1$
5. $x! = 1!$
6. $x! = 1$
7. $n! = y \cdot 1$
8. $n! = y$ or $y = n!$.

③ $y=1 \;\&\; x=n \;\&\; n>0 \vDash n! = yx!$

1. $y=1$
2. $x=n$
3. $n>0$
4. $n! = n!$
5. $n! = x!$
6. $x! = 1 \cdot x!$
7. $n! = 1 \cdot x!$
8. $n! = y \cdot x!$

④ $n! = y(x-1)! \vDash n! = y(x-1)!$

This is trivial.

⑤ $n! = yx! \;\&\; \neg(x=1) \vDash n! = (y*x)(x-1)!$

this is not possible?! what about when $x=0$, $y=1$

$n! = y\,0! = y \cdot 1 = 1 \cdot 1 = 1$

$n! = (1*0)(-1)!$
$= 0 \cdot (-1)! = ?$

So we must also add $x>0$ to the invariant.

1. $n! = yx!$
2. $\neg(x=1)$
3. $x>0$
4. $x! = x(x-1)!$
5. $n! = y(x(x-1)!)$
6. $n! = (y*x)(x-1)!$
7. ~~$x>0 \;\&\; x \neq 1$~~ $x>1$
8. $x-1>0$
9. $n! = (y*x)(x-1)! \;\&\; x-1>0$

Language Engineering: Semantics - Lecture 17

Here is the total correctness schemata for While:

$$[P]\ \text{Skip}\ [P]$$

$$[P(a)]\ x := a\ [P(x)]$$

$$\frac{[P]\ S_1\ [Q] \qquad [Q]\ S_2\ [R]}{[P]\ S_1; S_2\ [R]}$$

$$\frac{[P \wedge b]\ S_1\ [Q] \qquad [P \wedge \neg b]\ S_2\ [Q]}{[P]\ \text{if } b \text{ then } S_1 \text{ else } S_2\ [Q]}$$

$$\frac{[P(z+1)]\ S\ [P(z)]}{[\exists z \in \mathbb{N} \text{ st } P(z)]\ \text{while } b \text{ do } S\ [P(0)]} \quad \text{if} \quad \models \forall z \in \mathbb{N},\ P(z+1) \to b. \quad \models P(0) \to \neg b$$

$$\frac{[P']\ S\ [Q']}{[P]\ S\ [Q]} \quad \text{if} \quad P \models P' \quad Q' \models Q.$$

We can also give a linear representation of Partial axiomatic rules:

```
// P
Skip
//
```

```
// P(a)
x:=a
// P(x).
```

```
// P
 S1
;// Q
 S2
// R
```

```
// P
if b then (
  // P & b
   S1
  // Q
)else(
  // P & ¬b
   S2
  // Q
)
// Q
```

```
// P
while b do (
  // P ^ b
     S
  // P
)
// P ^ ¬b
```

```
// P
 S
// P'
 S
// Q
```

```
// P
 S
// Q'
// Q
```

Key

```
// X
 S
// Y
```
nested axiomatic proof of {X} S {Y}

```
// X
 |
// Y
```
logical proof that X ⊨ Y

The notation $\vdash_p \{P\} S \{Q\}$ is used iff the assertion $\{P\} S \{Q\}$ is provable in the partial axiomatic semantics. - ie there is a valid partial axiomatic proof tree with root $\{P\} S \{Q\}$.

The notation $\models \{P\} S \{Q\}$ is used iff the assertion $\{P\} S \{Q\}$ is a correct partial correctness assertion. - ie P holds in some states and if $S_{ds}[\![S]\!]s = s'$ then Q holds in s'.

In axiomatic semantics we are able to state that two programs are provably equivalent if for all P and Q conditions, we have:

$\vdash_p \{P\} S_1 \{Q\}$ iff $\vdash_p \{P\} S_2 \{Q\}$.

The End!