Semantic Equivalence

One of the reasons for wanting to have a formal definition of the semantics of a programming language is that it can serve as the basis of methods for reasoning about program properties and program specifications. In particular, a precise mathematical semantics is necessary for settling questions of *semantic equivalence* of program phrases, in other words for saying precisely when two phrases *have the same meaning*. The different styles of semantics mentioned on Slide 3 have different strengths and weaknesses when it comes to this task.

In an *axiomatic* approach to semantic equivalence, one just postulates axioms and rules for semantic equivalence which will include the general properties of equality shown on Slide 29, together with specific axioms and rules for the various phrase constructions. The importance of the Congruence rule cannot be over emphasised: it lies at the heart of the familiar process of *equational reasoning* whereby an equality is deduced in a number of steps, each step consisting of replacing a subphrase by another phrase already known to be equal to it. (Of course stringing the steps together relies upon the Transitivity rule.) For example, if we already know that $(C\,;C')\,;C''$ and $C\,;(C'\,;C'')$ are equivalent, then we can deduce that

$$\textbf{while}\ B\ \textbf{do}\ ((C\,;C')\,;C'') \quad \text{and} \quad \textbf{while}\ B\ \textbf{do}\ (C\,;(C'\,;C''))$$

are too, by applying the congruence rule with $\mathcal{C}[-] = \textbf{while}\ B\ \textbf{do}\ -$. Note that while Reflexivity, Symmetry and Transitivity are properties that can apply to any binary relation on a set, the Congruence property only makes sense once we have fixed which language we are talking about, and hence which 'contexts' $\mathcal{C}[-]$ are applicable.

How does one know which language-dependent axioms and rules to postulate in an axiomatisation of semantic equivalence? The approach we take here is to regard an operational semantics as part of a language's definition, develop a notion of semantic equivalence based on it, and then validate axioms and rules against this operational equivalence. We will illustrate this approach with respect to the language LC.

Basic properties of equality

Reflexivity	$P = P$
Symmetry	$\dfrac{P' = P}{P = P'}$
Transitivity	$\dfrac{P = P' \quad P' = P''}{P = P''}$
Congruence	$\dfrac{P = P'}{\mathcal{C}[P] = \mathcal{C}[P']}$

where $\mathcal{C}[P]$ is a phrase containing an occurrence of P and $\mathcal{C}[P']$ is the same phrase with that occurrence replaced by P'.

Definition of semantic equivalence of LC phrases

Two phrases of the same type are semantically equivalent

$$\boxed{P_1 \cong P_2}$$

if and only if for all states s and all terminal configurations $\langle V, s' \rangle$

$$\langle P_1, s \rangle \Downarrow \langle V, s' \rangle \quad \Leftrightarrow \quad \langle P_2, s \rangle \Downarrow \langle V, s' \rangle.$$

Semantic equivalence of LC phrases

It is natural to say that two LC phrases of the same type (i.e. both integer expressions, boolean expressions, or commands) are *semantically equivalent* if evaluating them in any given starting state produces exactly the same final state and value (if any).

the definition of $\cong$ according to the type of phrase:

- Two LC commands are semantically equivalent, $C_1 \cong C_2$, if and only if they determine the same partial function from states to states: for all s, either $\langle C_1, s\rangle \not\Downarrow$ & $\langle C_2, s\rangle \not\Downarrow$, or $\langle C_1, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle$ & $\langle C_2, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle$ for some s'.
- Two LC integer expressions are semantically equivalent, $E_1 \cong E_2$, if and only if they determine the same partial function from states to integers: for all s, either $\langle E_1, s\rangle \not\Downarrow$ & $\langle E_2, s\rangle \not\Downarrow$, or $\langle E_1, s\rangle \Downarrow \langle n, s\rangle$ & $\langle E_2, s\rangle \Downarrow \langle n, s\rangle$ for some $n \in \mathbb{Z}$.
- Two LC boolean expressions are semantically equivalent, $B_1 \cong B_2$, if and only if they determine the same partial function from states to booleans: for all s, either $\langle B_1, s\rangle \not\Downarrow$ & $\langle B_2, s\rangle \not\Downarrow$, or $\langle B_1, s\rangle \Downarrow \langle b, s\rangle$ & $\langle B_2, s\rangle \Downarrow \langle b, s\rangle$ for some $b \in \mathbb{B}$.

Semantic inequivalence of LC commands

To show $C_1 \ncong C_2$, it suffices to find states s, s' such that

either $\langle C_1, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle$ and $\langle C_2, s\rangle \not\Downarrow \langle \mathbf{skip}, s'\rangle$,

or $\langle C_1, s\rangle \not\Downarrow \langle \mathbf{skip}, s'\rangle$ and $\langle C_2, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle$

E.g. (Exercise 4.3.2) when $C = \mathbf{skip}$, $C' = \mathbf{while\ true\ do\ skip}$, $C'' = (\ell := 1)$ and $B = (!\ell = 0)$, then

$$C''; (\mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C') \ncong \mathbf{if}\ B\ \mathbf{then}\ (C''; C)\ \mathbf{else}\ C''; C')$$

Example

$$(\mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C')\ ;C'' \cong \mathbf{if}\ B\ \mathbf{then}\ (C\ ;C'')\ \mathbf{else}\ (C'\ ;C'')$$

Proof. Write

$$C_1 \stackrel{\text{def}}{=} (\textbf{if } B \textbf{ then } C \textbf{ else } C') \,;\, C''$$
$$C_2 \stackrel{\text{def}}{=} \textbf{if } B \textbf{ then } (C \,;\, C'') \textbf{ else } (C' \,;\, C'').$$

We exploit the structural nature of the rules in Figure 2 that inductively define the LC evaluation relation (and also the properties listed on Slide 28, in order to mildly simplify the case analysis). If it is the case that $\langle C_1, s\rangle \Downarrow \langle \textbf{skip}, s'\rangle$, because of the structure of C_1 this must have been deduced using ($\Downarrow_{\text{seq}}$). So for some s'' we have

(17) $$\langle \textbf{if } B \textbf{ then } C \textbf{ else } C', s\rangle \Downarrow \langle \textbf{skip}, s''\rangle$$

(18) $$\langle C'', s''\rangle \Downarrow \langle \textbf{skip}, s'\rangle.$$

The rule used to deduce (17) must be either ($\Downarrow_{\text{if1}}$) or ($\Downarrow_{\text{if2}}$). So

(19) $$\begin{cases} \text{either} & \langle B, s\rangle \Downarrow \langle \textbf{true}, s\rangle \;\&\; \langle C, s\rangle \Downarrow \langle \textbf{skip}, s''\rangle, \\ \text{or} & \langle B, s\rangle \Downarrow \langle \textbf{false}, s\rangle \;\&\; \langle C', s\rangle \Downarrow \langle \textbf{skip}, s''\rangle \end{cases}$$

(where we have made use of the fact that evaluation of expressions is side-effect free—

Examples of semantically equivalent LC commands

$$C \,;\, \textbf{skip} \cong C \cong \textbf{skip} \,;\, C$$
$$(C \,;\, C') \,;\, C'' \cong C \,;\, (C' \,;\, C'')$$
$$\textbf{if true then } C \textbf{ else } C' \cong C$$
$$\textbf{if false then } C \textbf{ else } C' \cong C'$$
$$\textbf{while } B \textbf{ do } C \cong \textbf{ if } B \textbf{ then } C \,;\, (\textbf{while } B \textbf{ do } C) \textbf{ else skip}$$
$$\ell := n \,;\, \ell' := n' \cong \begin{cases} \ell' := n' \,;\, \ell := n & \text{if } \ell \neq \ell' \\ \ell := n' & \text{if } \ell = \ell'. \end{cases}$$

Theorem 4.1.2. LC *semantic equivalence satisfies the properties of Reflexivity, Symmetry, Transitivity and Congruence given on Slide* 29.

Proof. The only one of the properties that does not follow immediately from the definition of $\cong$ is Congruence:

$$P_1 \cong P_2 \quad \Rightarrow \quad \mathcal{C}[P_1] \cong \mathcal{C}[P_2].$$

Analysing the structure of of LC contexts, this amounts to proving each of the properties listed in Figure 4. Most of these follow by routine case analysis of proofs of evaluation, along the lines of Example 4.1.1. The only non-trivial one is

$$C_1 \cong C_2 \quad \Rightarrow \quad \textbf{while } B \textbf{ do } C_1 \cong \textbf{while } B \textbf{ do } C_2$$

the proof of which we give.

For commands: if $C_1 \cong C_2$ then for all C and B

$$\begin{array}{rcl} C_1 \,;\, C & \cong & C_2 \,;\, C \\ C \,;\, C_1 & \cong & C \,;\, C_2 \\ \textbf{if } B \textbf{ then } C_1 \textbf{ else } C & \cong & \textbf{if } B \textbf{ then } C_2 \textbf{ else } C \\ \textbf{if } B \textbf{ then } C \textbf{ else } C_1 & \cong & \textbf{if } B \textbf{ then } C \textbf{ else } C_2 \\ \textbf{while } B \textbf{ do } C_1 & \cong & \textbf{while } B \textbf{ do } C_2. \end{array}$$

For integer expressions: if $E_1 \cong E_2$ then for all ℓ and E

$$\begin{array}{rcl} \ell := E_1 & \cong & \ell := E_2 \\ E_1 \; op \; E & \cong & E_2 \; op \; E \\ E \; op \; E_1 & \cong & E \; op \; E_2. \end{array}$$

For boolean expressions: if $B_1 \cong B_2$ then for all C and C'

$$\begin{array}{rcl} \textbf{if } B_1 \textbf{ then } C \textbf{ else } C' & \cong & \textbf{if } B_2 \textbf{ then } C \textbf{ else } C' \\ \textbf{while } B_1 \textbf{ do } C & \cong & \textbf{while } B_2 \textbf{ do } C. \end{array}$$

Figure 4: Congruence properties of LC semantic equivalence

Proof of

$$\boxed{C_1 \cong C_2 \quad \Rightarrow \quad \textbf{while } B \textbf{ do } C_1 \cong \textbf{while } B \textbf{ do } C_2}$$

is via:

Lemma. *If* $C_1 \cong C_2$, *then for all* $n \geq 0$

$$(\dagger_n) \begin{cases} \forall m \leq n.\, \forall s, s'. \\ \quad \langle \textbf{while } B \textbf{ do } C_1, s\rangle \to^m \langle \textbf{skip}, s'\rangle \\ \quad\quad \Rightarrow \langle \textbf{while } B \textbf{ do } C_2, s\rangle \to^* \langle \textbf{skip}, s'\rangle \end{cases}$$

(where $\to^m$ *means the composition of* m *transitions and* $\to^*$ *means* $\to^m$ *holds for some* $m \geq 0$*).*

Block structured local state

Because of the need to control interference between the state in different program parts, most procedural languages include the facility for declarations of locally scoped locations (program variables) whose evaluation involves the dynamic creation of fresh storage locations. In this section we consider semantic equivalence of commands involving a particularly simple form of such *local state*, $\textbf{begin loc } \ell := E;\ C\ \textbf{end}$, in which the life time of the freshly created location correlates precisely with the textual scope of the declaration: the location ℓ is created and initialised with the value of E at the beginning of the program 'block' C and deallocated at the end of the block. We call the language obtained from LC by adding this construct $\mathsf{LC}^{\mathrm{loc}}$: see Slide 34. Taking configurations to be as before (i.e. (command,state)-pairs), we can specify the operational semantics of $\mathsf{LC}^{\mathrm{loc}}$ by an evaluation relation inductively defined by the rules in Figure 2

$\mathsf{LC}^{\mathrm{loc}}$: LC+ **block structured local state**

Phrases: $P ::= C \mid E \mid B$

Commands:

$$\begin{array}{rcl} C & ::= & \textbf{skip} \mid \ell := E \mid C\,;C \mid \textbf{if } B \textbf{ then } C \textbf{ else } C \\ & \mid & \textbf{while } B \textbf{ do } C \mid \boxed{\textbf{begin loc } \ell := E;\ C \textbf{ end}} \end{array}$$

Integer expressions: $E ::= n \mid !\ell \mid E\ iop\ E$

Boolean expressions: $B ::= b \mid E\ bop\ E$

Slide 34

Evaluation rule for blocks

$$(\Downarrow_{\mathrm{bs}}) \quad \frac{\langle E, s\rangle \Downarrow \langle n, s'\rangle \quad \langle C[\ell'/\ell], s'[\ell' \mapsto n]\rangle \Downarrow \langle \mathbf{skip}, s''[\ell' \mapsto n']\rangle}{\langle \mathbf{begin\ loc}\, \ell := E;\ C\ \mathbf{end}, s\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}\ *$$

$*$ if $\ell' \notin dom(s') \cup dom(s'')$ and ℓ' does not occur in C.

$C[\ell'/\ell]$ indicates the $\mathsf{LC}^{\mathrm{loc}}$ command obtained from C by replacing all occurrences of ℓ with ℓ'.

Slide 35

Example 4.2.1. To see how rule $(\Downarrow_{\mathrm{bs}})$ works in practice, consider a command to swap the contents of two locations using a temporary location that happens to have the same name as a global one.

$$C \stackrel{\mathrm{def}}{=} \mathbf{begin\ loc}\, \ell := !\ell_1;\ \ell_1 := !\ell_2\, ; \ell_2 := !\ell\ \mathbf{end}.$$

Here we assume ℓ, ℓ_1, ℓ_2 are three distinct locations. Then for all states s with $\ell_1, \ell_2 \in dom(s)$ we have

$$\langle C, s\rangle \Downarrow \langle \mathbf{skip}, s[\ell_1 \mapsto s(\ell_2), \ell_2 \mapsto s(\ell_1)]\rangle.$$

and in particular the value stored at ℓ in the final state $s[\ell_1 \mapsto s(\ell_2), \ell_2 \mapsto s(\ell_1)]$ (if any) is the same as it is in the initial state s.

Proof. Let $n_i = s(\ell_i)$ $(i = 1, 2)$ and

$$s_1 \stackrel{\mathrm{def}}{=} s[\ell_1 \mapsto n_2], \quad s_2 \stackrel{\mathrm{def}}{=} s[\ell_1 \mapsto n_2, \ell_2 \mapsto n_1]$$

and choose any $\ell' \notin \{\ell, \ell_1, \ell_2\}$. Then

$$\dfrac{\dfrac{}{\langle !\ell_1, s\rangle \Downarrow \langle n_1, s\rangle}\,(\Downarrow_{\mathrm{loc}}) \quad \dfrac{\dfrac{\dfrac{}{\langle !\ell_2, s[\ell' \mapsto n_1]\rangle \Downarrow \langle n_2, s[\ell' \mapsto n_1]\rangle}\,(\Downarrow_{\mathrm{loc}})}{\langle \ell_1 := !\ell_2, s[\ell' \mapsto n_1]\rangle \Downarrow \langle \mathbf{skip}, s_1[\ell' \mapsto n_1]\rangle}\,(\Downarrow_{\mathrm{set}}) \quad \dfrac{\dfrac{}{\langle !\ell', s_1[\ell' \mapsto n_1]\rangle \Downarrow \langle n_1, s_1[\ell' \mapsto n_1]\rangle}\,(\Downarrow_{\mathrm{loc}})}{\langle \ell_2 := !\ell', s_1[\ell' \mapsto n_1]\rangle \Downarrow \langle \mathbf{skip}, s_2[\ell' \mapsto n_1]\rangle}\,(\Downarrow_{\mathrm{set}})}{\langle \ell_1 := !\ell_2\, ; \ell_2 := !\ell', s[\ell' \mapsto n_1]\rangle \Downarrow \langle \mathbf{skip}, s_2[\ell' \mapsto n_1]\rangle}\,(\Downarrow_{\mathrm{seq}})}{\langle \mathbf{begin\ loc}\, \ell := !\ell_1;\ \ell_1 := !\ell_2\, ; \ell_2 := !\ell\ \mathbf{end}, s\rangle \Downarrow \langle \mathbf{skip}, s_2\rangle}\,(\Downarrow_{\mathrm{bs}}).$$

is a proof for the claimed evaluation. □