Introduction

In this course we will take a close look at programming languages. We will focus on how one can define precisely what a programming language *is* – i.e., how the programs of the language behave, or, more generally, what their meaning, or *semantics*, is.

Semantics - What is it?

How to describe a programming language? Need to give:

- the *syntax* of programs; and
- their *semantics* (the meaning of programs, or how they behave).

Styles of description:

- the language is defined by whatever some particular compiler does
- natural language 'definitions'
- mathematically

Mathematical descriptions of syntax use formal grammars (eg BNF) – precise, concise, clear. In this course we'll see how to work with mathematical definitions of semantics/behaviour.

Many programming languages that you meet are described only in *natural language*, e.g. the English standards documents for C, Java, XML, etc. These are reasonably accessible (though often written in 'standardsese'), but there are some major problems. It is very hard, if not impossible, to write really precise definitions in informal prose. The standards often end up being ambiguous or incomplete, or just too large and hard to understand. That leads to differing implementations and flaky systems, as the language implementors and users do not have a common understanding of what it is. More fundamentally, natural language standards obscure the real structure of languages – it's all too easy to add a feature and a quick paragraph of text without thinking about how it interacts with the rest of the language.

Instead, as we shall see in this course, one can develop *mathematical* definitions of how programs behave, using logic and set theory (e.g. the definition of Standard ML, the .NET CLR, recent work on XQuery, etc.). These require a little more background to understand and use, but for many purposes they are a much better tool than informal standards.

What do we use semantics for?

1. to understand a particular language - what you can depend on as a programmer; what you must provide as a compiler writer
2. as a tool for language design:
   (a) for expressing design choices, understanding language features and how they interact.
   (b) for proving properties of a language, eg type safety, decidability of type inference.
3. as a foundation for proving properties of particular programs

Semantics complements the study of language implementation (cf. *Compiler Construction* and *Optimising Compilers*). We need languages to be *both* clearly understandable, with precise definitions, *and* have good implementations.

This is true not just for the major programming languages, but also for intermediate languages (JVM, CLR), and the many, many scripting and command languages, that have often been invented on-the-fly without sufficient thought. How many of you will do language design? lots!

More broadly, while in this course we will look mostly at semantics for conventional programming languages, similar techniques can be used for hardware description languages, verification of distributed algorithms, security protocols, and so on – all manner of subtle systems for which relying on informal intuition alone leads to error. Some of these are explored in *Specification and Verification* and *Topics in Concurrency*.

Warmup

In C, if initially x has value 3, what's the value of the following?

```
x++ + x++ + x++ + x++
```

C$^\sharp$

```
delegate int IntThunk();

class M {
  public static void Main() {
    IntThunk[] funcs = new IntThunk[11];
    for (int i = 0; i <= 10; i++)
    {
      funcs[i] = delegate() { return i; };
    }
    foreach (IntThunk f in funcs)
    {
      System.Console.WriteLine(f());
    }
  }
}
```

Ruby (expected)

```
def printdouble(x) print x*2, "\n" end
x = 123
print "x is ", x, "\n"
printdouble(7)
print "x is ", x, "\n"
```

Output:

```
x is 123
14
x is 123
```

Ruby (unexpected)

```
def applydouble(y) yield y*2 end
x = 123
print "x is ", x, "\n"
applydouble(7) {|x| print x, "\n" }
print "x is ", x, "\n"
```

Output of this program is

?

from Micro to Macro

- simple evaluation order
- what can be stored
- evaluation strategy (call-by-value, call-by-name)
- what can be abstracted over; what can be passed around
- what can/should type systems guarantee at compile-time
- ...

Various different approaches have been used for expressing semantics.

Styles of Semantic Definitions

- Operational semantics
- Denotational semantics
- Axiomatic, or Logical, semantics

...Static and dynamic semantics...

Operational: define the meaning of a program in terms of the computation steps it takes in an idealised execution. Some definitions use *structural operational semantics*, in which the intermediate states are described using the language itself; others use *abstract machines*, which use more ad-hoc mathematical constructions.

Denotational: define the meaning of a program as elements of some abstract mathematical structure, e.g. regarding programming-language functions as certain mathematical functions. cf. the Denotational Semantics course.

Axiomatic or Logical: define the meaning of a program indirectly, by giving the axioms of a logic of program properties. cf. Specification and Verification.

All these are *dynamic* semantics, describing behaviour in one way or another. In contrast the *static* semantics of a language describes its compile-time typechecking.

'Toy' languages

Real programming languages are large, with many features and, often, with redundant constructs – things that can be expressed in the rest of the language.

When trying to understand some particular combination of features it's usual to define a small 'toy' language with just what you're interested in, then scale up later. Even small languages can involve delicate design choices.

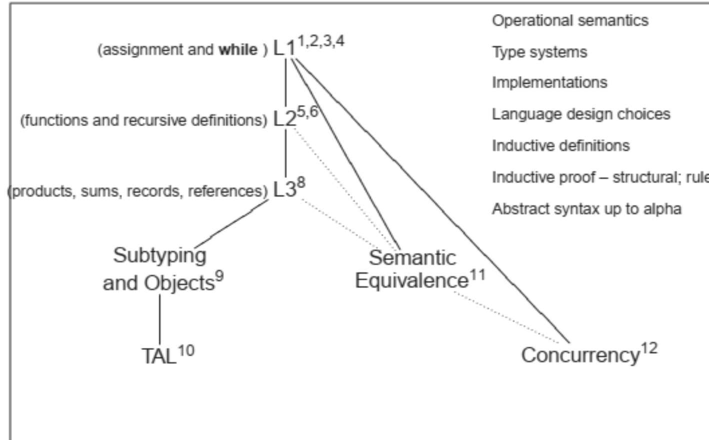
What's this course?

Core

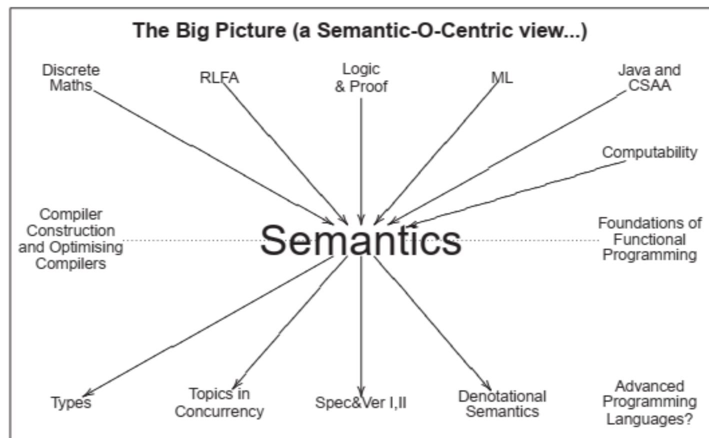
- operational semantics and typing for a tiny language
- technical tools (abstract syntax, inductive definitions, proof)
- design for functions, data and references

More advanced topics

- Subtyping and Objects
- Low-level Semantics (Typed Assembly Language)
- Semantic Equivalence
- Concurrency



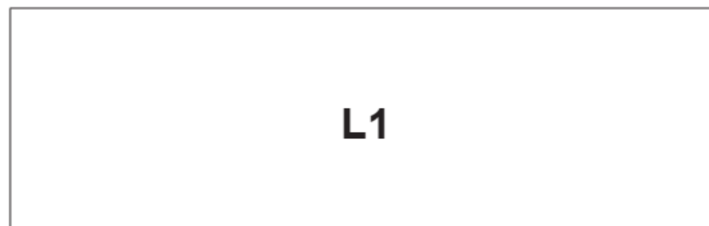
In the core we will develop enough techniques to deal with the semantics of a non-trivial small language, showing some language-design pitfalls and alternatives along the way. It will end up with the semantics of a decent fragment of ML. The second part will cover a selection of more advanced topics.



Admin

- Please let me know of typos, and if it is too fast/too slow/too interesting/too dull (please complete the on-line feedback at the end)
- Not all previous Tripos questions are relevant (see the notes)
- Exercises in the notes.
- Implementations on web.
- Books (Hennessy, Pierce, Winskel)

2 A First Imperative Language



L1 – Example

L1 is an imperative language with store locations (holding integers), conditionals, and **while** loops. For example, consider the program

$$
\begin{aligned}
& l_2 := 0; \\
& \textbf{while } !l_1 \geq 1 \textbf{ do } (\\
& \quad l_2 := !l_2 + !l_1; \\
& \quad l_1 := !l_1 + -1)
\end{aligned}
$$

in the initial store $\{l_1 \mapsto 3, l_2 \mapsto 0\}$.

L1 – Syntax

Booleans $b \in \mathbb{B} = \{\textbf{true}, \textbf{false}\}$
Integers $n \in \mathbb{Z} = \{..., -1, 0, 1, ...\}$
Locations $\ell \in \mathbb{L} = \{l, l_0, l_1, l_2, ...\}$

Operations $op ::= + \mid \geq$

Expressions

$$
\begin{aligned}
e \quad ::= \quad & n \mid b \mid e_1 \; op \; e_2 \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \mid \\
& \ell := e \mid !\ell \mid \\
& \textbf{skip} \mid e_1; e_2 \mid \\
& \textbf{while } e_1 \textbf{ do } e_2
\end{aligned}
$$

Write L_1 for the set of all expressions.

Points to note:

- we'll return later to *exactly* what the set L_1 is when we talk about abstract syntax
- unbounded integers
- abstract locations – can't do pointer arithmetic on them
- untyped, so have nonsensical expressions like $3 + \textbf{true}$
- what kind of grammar is that?
- don't have expression/command distinction

- doesn't much matter what basic operators we have
- carefully distinguish metavariables $b, n, \ell,\ op\ , e$ etc. from program locations l etc..

2.1 Operational Semantics

In order to describe the behaviour of L1 programs we will use structural operational semantics to define various forms of automata:

Transition systems

A *transition system* consists of

- a set Config, and
- a binary relation $\longrightarrow \subseteq$ Config $*$ Config.

The elements of Config are often called *configurations* or *states*. The relation $\longrightarrow$ is called the *transition* or *reduction* relation. We write $\longrightarrow$ infix, so $c \longrightarrow c'$ should be read as 'state c can make a transition to state c''.

To compare with the automata you saw in *Regular Languages and Finite Automata*: a transition system is like an NFA$^{\varepsilon}$ with an empty alphabet (so only ε transitions) except (a) it can have infinitely many states, and (b) we don't specify a start state or accepting states. Sometimes one adds labels (e.g. to represent IO) but mostly we'll just look at the values of terminated states, those that cannot do any transitions.

Some handy auxilary notation:

- $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$, so $c \longrightarrow^* c'$ iff there exist $k \geq 0$ and $c_0, .., c_k$ such that $c = c_0 \longrightarrow c_1 ... \longrightarrow c_k = c'$.
- $\not\longrightarrow$ is a unary predicate (a subset of Config) defined by $c \not\longrightarrow$ iff $\neg \exists\ c'.c \longrightarrow c'$.
- The transition relation is *deterministic* if for all states c there is at most one c' such that $c \longrightarrow c'$, ie if $\forall\ c.\forall\ c', c''.(c \longrightarrow c' \wedge c \longrightarrow c'') \implies c' = c''$.

The particular transition systems we use for L1 are as follows.

L1 Semantics (1 of 4) – Configurations

Say *stores* s are finite partial functions from $\mathbb{L}$ to $\mathbb{Z}$. For example:

$$\{l_1 \mapsto 7,\ l_3 \mapsto 23\}$$

Take *configurations* to be pairs $\langle e, s\rangle$ of an expression e and a store s, so our transition relation will have the form

$$\langle e, s\rangle \longrightarrow \langle e', s'\rangle$$

A *finite partial function* f from a set A to a set B is a set containing a finite number $n \geq 0$ of pairs $\{(a_1, b_1), ..., (a_n, b_n)\}$, often written $\{a_1 \mapsto b_1, ..., a_n \mapsto b_n\}$, for which

- $\forall\ i\ \in \{1, .., n\}.a_i\ \in\ A$ (the domain is a subset of$A$)
- $\forall\ i\ \in \{1, .., n\}.b_i\ \in\ B$ (the range is a subset of B)
- $\forall\ i\ \in \{1, .., n\}, j\ \in \{1, .., n\}.i \neq j \Rightarrow a_i \neq a_j$ (f is functional, i.e. each element of A is mapped to at most one element of B)

For a partial function f, we write $\mathrm{dom}(f)$ for the set of elements in the domain of f (things that f maps to something) and $\mathrm{ran}(f)$ for the set of elements in the range of f (things that something is mapped to by f). For example, for the s above we have $\mathrm{dom}(s) = \{l_1, l_3\}$ and $\mathrm{ran}(s) = \{7, 23\}$. Note that a finite partial function can be *empty*, just $\{\}$.

We write store for the set of all stores.

Transitions are single computation steps. For example we will have:

$$
\begin{array}{lll}
 & \langle l := 2+!l, & \{l \mapsto 3\}\rangle \\
\longrightarrow & \langle l := 2+3, & \{l \mapsto 3\}\rangle \\
\longrightarrow & \langle l := 5, & \{l \mapsto 3\}\rangle \\
\longrightarrow & \langle \textbf{skip}, & \{l \mapsto 5\}\rangle \\
\not\longrightarrow & &
\end{array}
$$

want to keep on until we get to a *value* v, an expression in

$$\mathbb{V} = \mathbb{B} \cup \mathbb{Z} \cup \{\textbf{skip}\}.$$

Say $\langle e, s\rangle$ is *stuck* if e is not a value and $\langle e, s\rangle \not\longrightarrow$. For example $2 + \textbf{true}$ will be stuck.

We could define the values in a different, but equivalent, style: Say *values* v are expressions from the grammar $v ::= b \mid n \mid \textbf{skip}$.

Now define the behaviour for each construct of L1 by giving some rules that (together) define a transition relation $\longrightarrow$.

L1 Semantics (2 of 4) – Rules (basic operations)

(op +) $\langle n_1 + n_2, s\rangle \longrightarrow \langle n, s\rangle$ if $n = n_1 + n_2$

(op $\geq$) $\langle n_1 \geq n_2, s\rangle \longrightarrow \langle b, s\rangle$ if $b = (n_1 \geq n_2)$

(op1) $$\frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1 \ \ op \ \ e_2, s\rangle \longrightarrow \langle e_1' \ \ op \ \ e_2, s'\rangle}$$

(op2) $$\frac{\langle e_2, s\rangle \longrightarrow \langle e_2', s'\rangle}{\langle v \ \ op \ \ e_2, s\rangle \longrightarrow \langle v \ \ op \ \ e_2', s'\rangle}$$

How to read these? The rule (op +) says that for any instantiation of the metavariables n, n_1 and n_2 (i.e. any choice of three integers), that satisfies the sidecondition, there is a transition from the instantiated configuration on the left to the one on the right.

We use a strict naming convention for metavariables: n can *only* be instantiated by integers, not by arbitrary expressions, cabbages, or what-have-you.

The rule (op1) says that for any instantiation of e_1, e_1', e_2, s, s' (i.e. any three expressions and two stores), *if* a transition of the form above the line can be deduced *then* we can deduce the transition below the line. We'll be more precise about this later.

Observe that – as you would expect – none of these first rules introduce changes in the store part of configurations.

Example

If we want to find the possible sequences of transitions of $\langle (2+3)+(6+7), \emptyset \rangle$ ... look for derivations of transitions.
(you might think the answer *should be* 18 – but we want to know what *this definition* says happens)

$$\text{(op1)}\ \dfrac{\text{(op +)}\ \dfrac{}{\langle 2+3, \emptyset\rangle \longrightarrow \langle 5, \emptyset\rangle}}{\langle (2+3)+(6+7), \emptyset\rangle \longrightarrow \langle 5+(6+7), \emptyset\rangle}$$

$$\text{(op2)}\ \dfrac{\text{(op +)}\ \dfrac{}{\langle 6+7, \emptyset\rangle \longrightarrow \langle 13, \emptyset\rangle}}{\langle 5+(6+7), \emptyset\rangle \longrightarrow \langle 5+13, \emptyset\rangle}$$

$$\text{(op +)}\ \dfrac{}{\langle 5+13, \emptyset\rangle \longrightarrow \langle 18, \emptyset\rangle}$$

First transition: using (op1) with $e_1 = 2+3$, $e_1' = 5$, $e_2 = 6+7$, $op = +$, $s = \emptyset$, $s' = \emptyset$, and using (op +) with $n_1 = 2$, $n_2 = 3$, $s = \emptyset$. Note couldn't begin with (op2) as $e_1 = 2+3$ is not a value, and couldn't use (op +) directly on $(2+3)+(6+7)$ as $2+3$ and $6+7$ are not numbers from $\mathbb{Z}$ – just expressions which might eventually evaluate to numbers (recall, by convention the n in the rules ranges over $\mathbb{Z}$ only).

Second transition: using (op2) with $e_1 = 5$, $e_2 = 6+7$, $e_2' = 13$, $op = +$, $s = \emptyset$, $s' = \emptyset$, and using (op +) with $n_1 = 6$, $n_2 = 7$, $s = \emptyset$. Note that to use (op2) we needed that $e_1 = 5$ is a value. We couldn't use (op1) as $e_1 = 5$ does not have any transitions itself.

Third transition: using (op +) with $n_1 = 5$, $n_2 = 13$, $s = \emptyset$.

To find each transition we do something like *proof search* in natural deduction: starting with a state (at the bottom left), look for a rule and an instantiation of the metavariables in that rule that makes the left-hand-side of its conclusion match that state. Beware that in general there might be more than one rule and one instantiation that does this. If there isn't a derivation concluding in $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$ then there isn't such a transition.

L1 Semantics (3 of 4) – store and sequencing

(deref) $\langle !\ell, s\rangle \longrightarrow \langle n, s\rangle$ if $\ell \in \text{dom}(s)$ and $s(\ell) = n$

(assign1) $\langle \ell := n, s\rangle \longrightarrow \langle \textbf{skip}, s + \{\ell \mapsto n\}\rangle$ if $\ell \in \text{dom}(s)$

(assign2) $\dfrac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle \ell := e, s\rangle \longrightarrow \langle \ell := e', s'\rangle}$

(seq1) $\langle \textbf{skip}; e_2, s\rangle \longrightarrow \langle e_2, s\rangle$

(seq2) $\dfrac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1; e_2, s\rangle \longrightarrow \langle e_1'; e_2, s'\rangle}$

Example

$$\begin{aligned}
\langle l := 3; !l, \{l \mapsto 0\}\rangle &\longrightarrow \langle \textbf{skip}; !l, \{l \mapsto 3\}\rangle \\
&\longrightarrow \langle !l, \{l \mapsto 3\}\rangle \\
&\longrightarrow \langle 3, \{l \mapsto 3\}\rangle
\end{aligned}$$

$$\langle l := 3; l := !l, \{l \mapsto 0\}\rangle \longrightarrow ?$$

$$\langle 15 + !l, \emptyset\rangle \longrightarrow ?$$

L1 Semantics (4 of 4) – The rest (conditionals and while)

(if1) $\langle \textbf{if}\ \ \textbf{true}\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3, s\rangle \longrightarrow \langle e_2, s\rangle$

(if2) $\langle \textbf{if}\ \ \textbf{false}\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3, s\rangle \longrightarrow \langle e_3, s\rangle$

(if3) $$\frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle \textbf{if}\ \ e_1\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3, s\rangle \longrightarrow \langle \textbf{if}\ \ e_1'\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3, s'\rangle}$$

(while)

$\langle \textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2, s\rangle \longrightarrow \langle \textbf{if}\ \ e_1\ \ \textbf{then}\ (e_2; \textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2)\ \textbf{else}\ \ \textbf{skip}, s\rangle$

Example

If

$e = (l_2 := 0; \textbf{while}\ !l_1 \geq 1\ \ \textbf{do}\ (l_2 :=!l_2+!l_1; l_1 :=!l_1 + -1))$

$s = \{l_1 \mapsto 3, l_2 \mapsto 0\}$

then

$\langle e, s\rangle \longrightarrow^* ?$

L1: Collected Definition

Syntax

Booleans $b \in \mathbb{B} = \{\textsf{true}, \textsf{false}\}$
Integers $n \in \mathbb{Z} = \{..., -1, 0, 1, ...\}$
Locations $\ell \in \mathbb{L} = \{l, l_0, l_1, l_2, ...\}$
Operations $op ::= + \mid \geq$
Expressions

$$\begin{aligned} e \quad ::= \quad & n \mid b \mid e_1\ \ op\ \ e_2 \mid \textbf{if}\ \ e_1\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3 \mid \\ & \ell := e \mid !\ell \mid \\ & \textbf{skip} \mid e_1; e_2 \mid \\ & \textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2 \end{aligned}$$

Operational Semantics

Note that for each construct there are some *computation* rules, doing 'real work', and some *context* (or *congruence*) rules, allowing subcomputations and specifying their order.
Say *stores* s are finite partial functions from $\mathbb{L}$ to $\mathbb{Z}$. Say *values* v are expressions from the grammar $v ::= b \mid n \mid \textbf{skip}$.

(op +) $\langle n_1 + n_2, s\rangle \longrightarrow \langle n, s\rangle$ if $n = n_1 + n_2$

(op $\geq$) $\langle n_1 \geq n_2, s\rangle \longrightarrow \langle b, s\rangle$ if $b = (n_1 \geq n_2)$

(op1) $$\frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1\ \ op\ \ e_2, s\rangle \longrightarrow \langle e_1'\ \ op\ \ e_2, s'\rangle}$$

(op2) $$\frac{\langle e_2, s\rangle \longrightarrow \langle e_2', s'\rangle}{\langle v\ \ op\ \ e_2, s\rangle \longrightarrow \langle v\ \ op\ \ e_2', s'\rangle}$$

(deref) $\langle !\ell, s\rangle \longrightarrow \langle n, s\rangle$ if $\ell \in \text{dom}(s)$ and $s(\ell) = n$

(assign1) $\langle \ell := n, s\rangle \longrightarrow \langle \textbf{skip}, s + \{\ell \mapsto n\}\rangle$ if $\ell \in \text{dom}(s)$

(assign2) $$\frac{\langle e, s\rangle \longrightarrow \langle e', s'\rangle}{\langle \ell := e, s\rangle \longrightarrow \langle \ell := e', s'\rangle}$$

(seq1) $\langle \textbf{skip}; e_2, s\rangle \longrightarrow \langle e_2, s\rangle$

(seq2) $$\frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle e_1; e_2, s\rangle \longrightarrow \langle e_1'; e_2, s'\rangle}$$

(if1) $\langle \textbf{if}\ \ \textbf{true}\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3, s\rangle \longrightarrow \langle e_2, s\rangle$

(if2) $\langle \textbf{if}\ \ \textbf{false}\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3, s\rangle \longrightarrow \langle e_3, s\rangle$

(if3) $$\frac{\langle e_1, s\rangle \longrightarrow \langle e_1', s'\rangle}{\langle \textbf{if}\ \ e_1\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3, s\rangle \longrightarrow \langle \textbf{if}\ \ e_1'\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3, s'\rangle}$$

(while)
$\langle \textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2, s\rangle \longrightarrow \langle \textbf{if}\ \ e_1\ \ \textbf{then}\ (e_2; \textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2)\ \textbf{else}\ \ \textbf{skip}, s\rangle$

Determinacy

Theorem 1 (L1 Determinacy) *If* $\langle e, s\rangle \longrightarrow \langle e_1, s_1\rangle$ *and* $\langle e, s\rangle \longrightarrow \langle e_2, s_2\rangle$ *then* $\langle e_1, s_1\rangle = \langle e_2, s_2\rangle$.

Proof - see later

Note that top-level universal quantifiers are usually left out – the theorem really says "For all e, s, e_1, s_1, e_2, s_2, if $\langle e, s\rangle \longrightarrow \langle e_1, s_1\rangle$ and $\langle e, s\rangle \longrightarrow \langle e_2, s_2\rangle$ then $\langle e_1, s_1\rangle = \langle e_2, s_2\rangle$".

L1 Implementation

Many possible implementation strategies, including:

1. animate the rules — use unification to try to match rule conclusion left-hand-sides against a configuration; use backtracking search to find all possible transitions. Hand-coded, or in Prolog/LambdaProlog/Twelf.
2. write an interpreter working directly over the syntax of configurations. Coming up, in ML and Java.
3. compile to a stack-based virtual machine, and an interpreter for that. See Compiler Construction.
4. compile to assembly language, dealing with register allocation etc. etc. See Compiler Construction/Optimizing Compilers.

L1 Implementation

Will implement an interpreter for L1, following the definition. Use mosml (Moscow ML) as the implementation language, as datatypes and pattern matching are good for this kind of thing.

First, must pick representations for locations, stores, and expressions:

```
type loc = string
type store = (loc * int) list
```

We've chosen to represent locations as strings, rather arbitrarily (really, so they pretty-print trivially). A lower-level implementation would use ML references or, even lower, machine pointers.

In the semantics, a store is a finite partial function from locations to integers. In the implementation, we represent a store as a list of `loc*int` pairs containing, for each ℓ in the domain of the store and mapped to n, exactly one element of the form `(l,n)`. The order of the list will not be important. This is not a very efficient implementation, but it is simple.

```
datatype oper = Plus | GTEQ

datatype expr =
   Integer of int
 | Boolean of bool
 | Op of expr * oper * expr
 | If of expr * expr * expr
 | Assign of loc * expr
 | Deref of loc
 | Skip
 | Seq of expr * expr
 | While of expr * expr
```

The expression and operation datatypes have essentially the same form as the abstract grammar. Note, though, that it does not exactly match the semantics, as that allowed arbitrary integers whereas here we use the bounded Moscow ML integers – so not every term of the abstract syntax is representable as an element of type `expr`, and the interpreter will fail with an overflow exception if `+` overflows.

Store operations

Define auxiliary operations

```
lookup :  store*loc -> int option
update :  store*(loc*int) -> store option
```

which both return `NONE` if given a location that is not in the domain of the store. Recall that a value of type `T option` is either `NONE` or `SOME v` for a value `v` of `T`.

The single-step function

Now define the single-step function

```
reduce : expr*store -> (expr*store) option
```

which takes a configuration `(e,s)` and returns either

`NONE`, if $\langle e, s\rangle \not\longrightarrow$,

or `SOME (e',s')`, if it has a transition $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.

Note that if the semantics didn't define a deterministic transition system we'd have to be more elaborate.

(you might think it would be better ML style to use exceptions instead of these options; that would be fine).

(op $+$), (op $\geq$)

```
fun reduce (Integer n,s) = NONE
  | reduce (Boolean b,s) = NONE
  | reduce (Op (e1,opr,e2),s) =
    (case (e1,opr,e2) of
          (Integer n1, Plus, Integer n2) =>
            SOME(Integer (n1+n2), s)
        | (Integer n1, GTEQ, Integer n2) =>
            SOME(Boolean (n1 >= n2), s)
        | (e1,opr,e2) =>
            ...
```

Contrast this code with the semantic rules given earlier.

(op1), (op2)

```
          ...
          if (is_value e1) then
              case reduce (e2,s) of
                  SOME (e2',s') =>
                    SOME (Op(e1,opr,e2'),s')
                | NONE => NONE
          else
              case reduce (e1,s) of
                  SOME (e1',s') =>
                    SOME(Op(e1',opr,e2),s')
                | NONE => NONE )
```

Note that the code depends on global properties of the semantics, including the fact that it defines a deterministic transition system, so the comments indicating that particular lines of code implement particular semantic rules are not the whole story.

(assign1), (assign2)

```
| reduce (Assign (l,e),s) =
  (case e of
       Integer n =>
          (case update (s,(l,n)) of
               SOME s' => SOME(Skip, s')
             | NONE => NONE)
     | _ =>
          (case reduce (e,s) of
               SOME (e',s') =>
                 SOME(Assign (l,e'), s')
             | NONE => NONE ) )
```

The many-step evaluation function

Now define the many-step evaluation function

`evaluate: expr*store -> (expr*store) option`

which takes a configuration `(e,s)` and returns the `(e',s')` such that $\langle e, s\rangle \longrightarrow^* \langle e', s'\rangle \not\longrightarrow$, if there is such, or does not return.

```
fun evaluate (e,s) =
  case reduce (e,s) of
    NONE => (e,s)
  | SOME (e',s') => evaluate (e',s')
```

The full interpreter code is available on the web, in the file `l1.ml`, together with a pretty-printer and the type-checker we will come to soon. You should make it go...

```
(* 2002-11-08 -- Time-stamp: <2004-01-03 16:17:04 pes20>    -*-SML-*- *)
(* Peter Sewell                                                        *)

(* This file contains an interpreter, pretty-printer and type-checker
   for the language L1.  To make it go, copy it into a working
   directory, ensure Moscow ML is available, and type

       mosml -P full l1.ml

   That will give you a MoscowML top level in which these definitions
   are present.  You can then type

      doit ();

   to show the reduction sequence of < l1:=3;!l1 , l1=0  >, and

      doit2 ();

   to run the type-checker on the same simple example; you can try
   other examples analogously.  This file doesn't have a parser for
```

```
   l1, so you'll have to enter the abstract syntax directly, eg

      prettyreduce (Seq( Assign ("l1",Integer 3), Deref "l1"), [("l1",0)]);

   This has been tested with Moscow ML version 2.00 (June 2000), but
   should work with any other implementation of Standard ML.   *)


(* ********************)
(* the abstract syntax *)
(* ********************)

type loc = string

datatype oper = Plus | GTEQ

datatype expr =
         Integer of int
       | Boolean of bool
       | Op of expr * oper * expr
       | If of expr * expr * expr
       | Assign of loc * expr
       | Deref of loc
       | Skip
       | Seq of expr * expr
       | While of expr * expr


(* **********************************)
(* an interpreter for the semantics *)
(* **********************************)

fun is_value (Integer n) = true
  | is_value (Boolean b) = true
  | is_value (Skip) = true
  | is_value _ = false

  (* In the semantics, a store is a finite partial function from
  locations to integers.  In the implementation, we represent a store
  as a list of loc*int pairs containing, for each l in the domain of
  the store, exactly one element of the form (l,n).  The operations

    lookup : store * loc         -> int option
    update : store * (loc * int) -> store option

  both return NONE if given a location that is not in the domain of
  the store.  This is not a very efficient implementation, but it is
  simple. *)

type store = (loc * int) list

fun lookup ( [], l ) = NONE
  | lookup ( (l',n')::pairs, l) =
    if l=l' then SOME n' else lookup (pairs,l)

fun update'  front [] (l,n) = NONE
 |  update'  front ((l',n')::pairs) (l,n) =
    if l=l' then
        SOME(front @ ((l,n)::pairs) )
    else
        update' ((l',n')::front) pairs (l,n)
```

```
fun update (s, (l,n)) = update' [] s (l,n)


  (* now define the single-step function

     reduce :  expr * store -> (expr * store) option

  which takes a configuration (e,s) and returns either NONE, if it has
  no transitions, or SOME (e',s'), if it has a transition (e,s) -->
  (e',s').

  Note that the code depends on global properties of the semantics,
  including the fact that it defines a deterministic transition
  system, so the comments indicating that particular lines of code
  implement particular semantic rules are not the whole story.  *)

fun reduce (Integer n,s) = NONE
  | reduce (Boolean b,s) = NONE
  | reduce (Op (e1,opr,e2),s) =
    (case (e1,opr,e2) of
         (Integer n1, Plus, Integer n2) => SOME(Integer (n1+n2), s)   (*op + *)
       | (Integer n1, GTEQ, Integer n2) => SOME(Boolean (n1 >= n2), s)(*op >=*)
       | (e1,opr,e2) => (
         if (is_value e1) then (
             case reduce (e2,s) of
                 SOME (e2',s') => SOME (Op(e1,opr,e2'),s')     (* (op2) *)
               | NONE => NONE )
         else (
             case reduce (e1,s) of
                 SOME (e1',s') => SOME(Op(e1',opr,e2),s')      (* (op1) *)
               | NONE => NONE ) ) )
  | reduce (If (e1,e2,e3),s) =
    (case e1 of
         Boolean(true) => SOME(e2,s)                           (* (if1) *)
       | Boolean(false) => SOME(e3,s)                          (* (if2) *)
       | _ => (case reduce (e1,s) of
                   SOME(e1',s') => SOME(If(e1',e2,e3),s')      (* (if3) *)
                 | NONE => NONE ))
  | reduce (Deref l,s) =
    (case lookup  (s,l) of
          SOME n => SOME(Integer n,s)                          (* (deref) *)
        | NONE => NONE )
  | reduce (Assign (l,e),s) =
    (case e of
         Integer n => (case update (s,(l,n)) of
                           SOME s' => SOME(Skip, s')           (* (assign1) *)
                         | NONE => NONE)
       | _ => (case reduce (e,s) of
                   SOME (e',s') => SOME(Assign (l,e'), s')     (* (assign2) *)
                 | NONE => NONE  ) )
  | reduce (While (e1,e2),s) = SOME( If(e1,Seq(e2,While(e1,e2)),Skip),s) (* (while) *)
  | reduce (Skip,s) = NONE
  | reduce (Seq (e1,e2),s) =
    (case e1 of
        Skip => SOME(e2,s)                                     (* (seq1) *)
      | _ => ( case reduce (e1,s) of
                 SOME (e1',s') => SOME(Seq (e1',e2), s')       (* (seq2) *)
               | NONE => NONE ) )


  (* now define the many-step evaluation function
```

```
     evaluate :  expr * store -> (expr * store) option

   which takes a configuration (e,s) and returns the unique (e',s')
   such that   (e,s) -->* (e',s') -/->.  *)

 fun evaluate (e,s) = case reduce (e,s) of
                          NONE => (e,s)
                        | SOME (e',s') => evaluate (e',s')
```

The Java Implementation

Quite different code structure:

- the ML groups together all the parts of each algorithm, into the `reduce`, `infertype`, and `prettyprint` functions;
- the Java groups together everything to do with each clause of the abstract syntax, in the `IfThenElse`, `Assign`, etc. classes.