Implementing the Semantics of Programming Languages

Lorenzo Ceragioli

October 2, 2024

IMT Lucca

Programming Languages

For first part of your project is to implement the semantics of two simple programming languages: MiniImp and MiniFun.

Each language is specified by:

- its syntax (what is a program)
  – given as a grammar
- its semantics (what do programs mean)
  – given in terms of a deduction system

As a simplifying assumption, all valid programs will define (partial) functions from integers to integers!

Deduction Systems

Deduction systems come from logic: they are a way of defining how validity (truth) propagates from a formula to the others

$$\frac{}{\top}\ \textsc{true} \qquad \frac{A \quad B}{A \wedge B}\ \textsc{And} \qquad \frac{A}{A \vee B}\ \textsc{Or1} \qquad \frac{B}{A \vee B}\ \textsc{Or2}$$

A formula is valid if there is a proof for it

$$\frac{\dfrac{}{\top}\ \textsc{true} \qquad \dfrac{\dfrac{}{\top}\ \textsc{true}}{A \vee \top}\ \textsc{Or2}}{\top \wedge (A \vee \top)}\ \textsc{And}$$

MiniImp

Syntax

A program p is defined as follows

$$p \coloneqq \texttt{def main with input } x \texttt{ output } y \texttt{ as } c$$

$$c \coloneqq \texttt{skip} \mid x \texttt{ := } a \mid c; c$$

$$\mid \texttt{if } b \texttt{ then } c \texttt{ else } c \mid \texttt{while } b \texttt{ do } c$$

$$b \coloneqq v \mid b \text{ and } b \mid \text{not } b \mid a < a$$

$$a \coloneqq x \mid n \mid a + a \mid a - a \mid a * a$$

where

- $x, x', x'' \in X$ are integer variables (any sequence of letters and numbers starting with a letter);
- $n, n', n'' \in \mathbb{Z}$ are integer numbers $(0, 1, -1, \ldots)$;
- $v, v', v'', \dots \in \mathbb{B}$ are boolean literals (`true`, `false`).

Intuitive Semantics

```
def main with input in output out as
  x := in;
  out := 0;
  while not x < 1 do (
    out := out + x;
    x := x − 1
  );
```

Execution with input 2:

Memory: [,]

[,]

[,]

Returns the final value of `out`, i.e. 3

Intuitive Semantics

```
def main with input in output out as
  x := in;
  out := 0;
  while not x < 1 do (
    out := out + x;
    x := x - 1
  );
```

Execution with input 2:

Memory: $[\mathrm{x} \mapsto 2, \qquad]$

$[\qquad , \qquad]$

$[\qquad , \qquad]$

Returns the final value of `out`, i.e. 3

Intuitive Semantics

```
def main with input in output out as
  x := in;
  out := 0;
  while not x < 1 do (
    out := out + x;
    x := x − 1
  );
```

Execution with input 2:

Memory: $[\mathrm{x} \mapsto 2, \mathrm{out} \mapsto 0]$

$[\quad , \quad]$

$[\quad , \quad]$

Returns the final value of `out`, i.e. 3

Intuitive Semantics

```
def main with input in output out as
  x := in;
  out := 0;
  while not x < 1 do (
    out := out + x;
    x := x - 1
  );
```

Execution with input 2:

$$\textbf{Memory:}\ [\mathrm{x} \mapsto 2, \mathrm{out} \mapsto 0]$$
$$[\quad\quad , \mathrm{out} \mapsto 2]$$
$$[\quad\quad , \quad\quad]$$

Returns the final value of `out`, i.e. 3

Intuitive Semantics

```
def main with input in output out as
  x := in;
  out := 0;
  while not x < 1 do (
    out := out + x;
    x := x - 1
  );
```

Execution with input 2:

$$\begin{aligned}\textbf{Memory:}\ & [\mathrm{x} \mapsto 2, \mathrm{out} \mapsto 0] \\ & [\mathrm{x} \mapsto 1, \mathrm{out} \mapsto 2] \\ & [\qquad , \qquad\quad]\end{aligned}$$

Returns the final value of `out`, i.e. 3

Intuitive Semantics

```
def main with input in output out as
  x := in;
  out := 0;
  while not x < 1 do (
    out := out + x;
    x := x − 1
  );
```

Execution with input 2:

$$\textbf{Memory:}\ [\mathrm{x} \mapsto 2, \mathrm{out} \mapsto 0]$$

$$[\mathrm{x} \mapsto 1, \mathrm{out} \mapsto 2]$$

$$[\qquad, \mathrm{out} \mapsto 3]$$

Returns the final value of `out`, i.e. 3

Intuitive Semantics

```
def main with input in output out as
  x := in;
  out := 0;
  while not x < 1 do (
    out := out + x;
    x := x − 1
  );
```

Execution with input 2:

$$\begin{aligned}\textbf{Memory:}\ & [\mathrm{x} \mapsto 2, \mathrm{out} \mapsto 0] \\ & [\mathrm{x} \mapsto 1, \mathrm{out} \mapsto 2] \\ & [\mathrm{x} \mapsto 0, \mathrm{out} \mapsto 3]\end{aligned}$$

Returns the final value of `out`, i.e. 3

Semantics: Memory and Reductions

An imperative language operates by reading and updating a memory σ, in our case, it associates variables with integer numbers:

$$\sigma \text{ is a partial function from } X \text{ to } \mathbb{Z}$$

The semantics is given by four reductions:

- for arithmetical expressions $\langle \sigma, a \rangle \longrightarrow_a n$
- for boolean expressions $\langle \sigma, b \rangle \longrightarrow_b v$
- for commands $\langle \sigma, c \rangle \longrightarrow_c \sigma$
- for programs $\langle p, n \rangle \longrightarrow_p n'$
  (recall, the semantics is a function from integers to integers)

Semantics: Arithmetic Expressions

We assume a function $\mathcal{O}(\,\cdot\,)$ that maps each syntactical operator to its corresponding operation (e.g. the symbol + to addition)

$$\frac{}{\langle \sigma, n \rangle \longrightarrow_a n}\ \textsc{Num} \qquad \frac{}{\langle \sigma, x \rangle \longrightarrow_a \sigma(x)}\ \textsc{Var}$$

$$\frac{\langle \sigma, a_1 \rangle \longrightarrow_a n_1 \quad \langle \sigma, a_2 \rangle \longrightarrow_a n_2}{\langle \sigma, a_1 + a_2 \rangle \longrightarrow_a n_1\,\mathcal{O}(+)\,n_2}\ \textsc{Plus}$$

$$\frac{\langle \sigma, a_1 \rangle \longrightarrow_a n_1 \quad \langle \sigma, a_2 \rangle \longrightarrow_a n_2}{\langle \sigma, a_1 - a_2 \rangle \longrightarrow_a n_1\,\mathcal{O}(\text{-})\,n_2}\ \textsc{Minus}$$

$$\frac{\langle \sigma, a_1 \rangle \longrightarrow_a n_1 \quad \langle \sigma, a_2 \rangle \longrightarrow_a n_2}{\langle \sigma, a_1 * a_2 \rangle \longrightarrow_a n_1\,\mathcal{O}(*)\,n_2}\ \textsc{Times}$$

Semantics: Boolean Expressions

$$\frac{}{\langle \sigma, v \rangle \longrightarrow_b v} \textsc{Bool}$$

$$\frac{\langle \sigma, b_1 \rangle \longrightarrow_b n_1 \quad \langle \sigma, b_2 \rangle \longrightarrow_b n_2}{\langle \sigma, b_1 \text{ and } b_2 \rangle \longrightarrow_b n_1 \, \mathcal{O}(\text{and}) \ n_2} \textsc{And}$$

$$\frac{\langle \sigma, b \rangle \longrightarrow_b b}{\langle \sigma, \text{not } b \rangle \longrightarrow_b \mathcal{O}(\text{not}) \, b} \textsc{Not}$$

$$\frac{\langle \sigma, a_1 \rangle \longrightarrow_a n_1 \quad \langle \sigma, a_2 \rangle \longrightarrow_b n_2}{\langle \sigma, a_1 < a_2 \rangle \longrightarrow_b n_1 \, \mathcal{O}(<) \, n_2} \textsc{Less}$$

Implementing Expressions

You already know from exercises 1 and 2 how to implement a simpler version of the abstract syntax three of arithmetical and boolean expressions and an evaluation function for them

Note: Something is still missing: **memory** and **variables**

But the same approach also works for more complex languages:

- a type for the abstract syntax tree
- an evaluation function defined (possibly recursively) over the abstract syntax tree
- in addition, it may be the case that you have to implement the run-time environment, i.e. the infrastructure needed for executing the code (in our case, the memory)

Semantics: Commands

We write $\sigma[x \mapsto n]$ for the memory obtained by updating (i.e. adding or overwriting) the binding for x, associating it to n.

$$\frac{}{\langle \sigma, \texttt{skip} \rangle \longrightarrow_c \sigma}\ \textsc{Skip}$$

$$\frac{\langle \sigma, a \rangle \longrightarrow_a n}{\langle \sigma, x \mathrel{\texttt{:=}} a \rangle \longrightarrow_c \sigma[x \mapsto n]}\ \textsc{Assign}$$

$$\frac{\langle \sigma, c_1 \rangle \longrightarrow_c \sigma_1 \quad \langle \sigma_1, c_2 \rangle \longrightarrow_c \sigma_2}{\langle \sigma, c_1; c_2 \rangle \longrightarrow_c \sigma_2}\ \textsc{Seq}$$

Semantics: Commands

$$\frac{\langle \sigma, b \rangle \longrightarrow_b \texttt{true} \quad \langle \sigma, c_1 \rangle \longrightarrow_c \sigma_1}{\langle \sigma, \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \rangle \longrightarrow_c \sigma_1} \ \textsc{IfTrue}$$

$$\frac{\langle \sigma, b \rangle \longrightarrow_b \texttt{false} \quad \langle \sigma, c_2 \rangle \longrightarrow_c \sigma_2}{\langle \sigma, \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \rangle \longrightarrow_c \sigma_2} \ \textsc{IfFalse}$$

Semantics: Commands

$$\frac{\langle \sigma, b \rangle \longrightarrow_b \texttt{true} \quad \langle \sigma, c; \texttt{while}\ b\ \texttt{do}\ c \rangle \longrightarrow_c \sigma_1}{\langle \sigma, \texttt{while}\ b\ \texttt{do}\ c \rangle \longrightarrow_c \sigma_1} \ \textsc{WhileTrue}$$

$$\frac{\langle \sigma, b \rangle \longrightarrow_b \texttt{false}}{\langle \sigma, \texttt{while}\ b\ \texttt{do}\ c \rangle \longrightarrow_c \sigma} \ \textsc{WhileFalse}$$

Semantics: Programs

We write σ_0 for the memory that is always undefined.

$$\frac{\langle \sigma_0[x \mapsto n], c \rangle \longrightarrow_c \sigma'}{\langle \texttt{def main with input } x \texttt{ output } y \texttt{ as } c, n \rangle \longrightarrow_p \sigma'(y)} \ \textsc{Prog}$$

Implementing the Semantics

1. Define a type for the abstract syntax tree
2. Define types and function for the run-time environment
3. Translate the deduction system into functions (not always easy)
4. Encode the functions into OCaml

Project Fragment 1. Create a module for MiniImp that exposes the type of the abstract syntax tree and an evaluation function.

A Remark on Deadlock and Non-termination

Notice: not every program has a defined semantics.

Programs may:

- **Fail** – i.e. arrives in erroneous states where we don't know how to proceed
- **Diverge** – basically loop forever

Also in a simple language like MiniImp: this is why the semantics is a *partial function*!

How de we deal with these problems?

A Remark on Deadlock

The only case in which a MiniImp program reaches a deadlock is when a variable is undefined!

```
def main with input a output b as
  x := 1;
  b := a + x + y
```

If we try to build a derivation for the semantics of this program, we reach a certain point where we cannot proceed

$$\dfrac{\dfrac{\dfrac{\dots}{\langle \sigma, a+x\rangle \longrightarrow_a n}\ \textsc{Plus} \qquad \dfrac{???}{\langle \sigma, y\rangle \longrightarrow_a \sigma(y)}\ \textsc{Var}}{\langle \sigma, (a+x)+y\rangle \longrightarrow_a n+?}\ \textsc{Plus}}{\langle \sigma, b := a+x+y\rangle \longrightarrow_c \sigma[x \mapsto n+?]}\ \textsc{Assign}$$

Dealing with Deadlocks

Two possible approaches :

- raise an error at run-time
  – that's the right way for the moment
  – use OCaml exceptions (e.g. use `failwith "message"`)
- prove before running the code that no deadlock will ever occur
  – this require approximating, i.e. some program will be rejected even if not problematic
  – we will see later in the course how to implement it

A Remark on Non-termination

The only cause for non-termination in MiniImp is the `while`

```
def main with input a output b as
  while true do
    x := 1
```

There is no derivation for the semantics of this program, while searching, the derivation tree grows infinitely

Let A = `x := 1` and W = `while true do x := 1`.

$$
\dfrac{
  \dfrac{}{\langle \sigma, \texttt{true} \rangle \longrightarrow_b \texttt{true}}\ \textsc{Bool}
  \qquad
  \dfrac{
    \dfrac{\cdots}{\langle \sigma, A \rangle \longrightarrow_c \sigma''}\ \textsc{Assign}
    \qquad
    \dfrac{\cdots}{\langle \sigma'', W \rangle \longrightarrow_c \sigma'}\ \textsc{WhileTrue}
  }{\langle \sigma, A; W \rangle \longrightarrow_c \sigma'}\ \textsc{Seq}
}{\langle \sigma, W \rangle \longrightarrow_c \sigma'}\ \textsc{WhileTrue}
$$

Dealing with Non-termination

You can only accepts that this is how programs behaves sometimes!

This is why also OCaml programs are *partial* functions

```
let rec f x = f x in f 0
```

Your evaluation function is allowed to diverge if the MiniImp program itself is non-terminating!