Implementing the Semantics of Programming Languages

Lorenzo Ceragioli

October 16, 2024

IMT Lucca

Programming Languages

First part of the Project: 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

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

MiniImp

Syntax

A program p is defined as follows

$$
\begin{aligned}
p &:= \texttt{def main with input } x \texttt{ output } y \texttt{ as } c \\
c &:= \texttt{skip} \mid x := a \mid c; c \\
&\quad \mid \texttt{if } b \texttt{ then } c \texttt{ else } c \mid \texttt{while } b \texttt{ do } c \\
b &:= v \mid b \text{ and } b \mid \text{not } b \mid a < a \\
a &:= x \mid n \mid a + a \mid a - a \mid a * a
\end{aligned}
$$

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 $(0, 1, -1, \dots)$;
- $v, v', v'', \cdots \in \mathbb{B}$ are boolean literals (`true`, `false`).

Intuitive Semantics

```
1  def main with input in output out as
2    x := in;
3    out := 0;
4    while not x < 1 do (
5      out := out + x;
6      x := x - 1
7    );
```

Execution with input 2:

Memory: [, ,]
[, ,]
[, ,]

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

Intuitive Semantics

```
1  def main with input in output out as
2    x := in;
3    out := 0;
4    while not x < 1 do (
5      out := out + x;
6      x := x − 1
7    );
```

Execution with input 2:

Memory: $[\texttt{in} \mapsto 2, \quad , \quad]$

$[\quad , \quad , \quad]$

$[\quad , \quad , \quad]$

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

Intuitive Semantics

```
1  def main with input in output out as
2    x := in;
3    out := 0;
4    while not x < 1 do (
5      out := out + x;
6      x := x - 1
7    );
```

Execution with input 2:

$$\textbf{Memory: } [\texttt{in} \mapsto 2, \texttt{x} \mapsto 2, \quad]$$
$$[\quad , \quad , \quad]$$
$$[\quad , \quad , \quad]$$

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

Intuitive Semantics

```
1  def main with input in output out as
2    x := in;
3    out := 0;
4    while not x < 1 do (
5      out := out + x;
6      x := x − 1
7    );
```

Execution with input 2:

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

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

Intuitive Semantics

```
1  def main with input in output out as
2    x := in;
3    out := 0;
4    while not x < 1 do (
5      out := out + x;
6      x := x - 1
7    );
```

Execution with input 2:

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

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

Intuitive Semantics

```
1  def main with input in output out as
2    x := in;
3    out := 0;
4    while not x < 1 do (
5      out := out + x;
6      x := x − 1
7    );
```

Execution with input 2:

$$\textbf{Memory: } [\texttt{in} \mapsto 2, \texttt{x} \mapsto 2, \texttt{out} \mapsto 0]$$
$$[\texttt{in} \mapsto 2, \texttt{x} \mapsto 1, \texttt{out} \mapsto 2]$$
$$[\quad , \quad , \quad]$$

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

Intuitive Semantics

```
1  def main with input in output out as
2    x := in;
3    out := 0;
4    while not x < 1 do (
5      out := out + x;
6      x := x - 1
7    );
```

Execution with input 2:

$$\textbf{Memory: } [\texttt{in} \mapsto 2, \texttt{x} \mapsto 2, \texttt{out} \mapsto 0]$$
$$[\texttt{in} \mapsto 2, \texttt{x} \mapsto 1, \texttt{out} \mapsto 2]$$
$$[\texttt{in} \mapsto 2, \quad\quad\quad , \texttt{out} \mapsto 3]$$

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

Intuitive Semantics

```
1  def main with input in output out as
2    x := in;
3    out := 0;
4    while not x < 1 do (
5      out := out + x;
6      x := x - 1
7    );
```

Execution with input 2:

$$\textbf{Memory: } [\texttt{in} \mapsto 2, \texttt{x} \mapsto 2, \texttt{out} \mapsto 0]$$
$$[\texttt{in} \mapsto 2, \texttt{x} \mapsto 1, \texttt{out} \mapsto 2]$$
$$[\texttt{in} \mapsto 2, \texttt{x} \mapsto 0, \texttt{out} \mapsto 3]$$

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.

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

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

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

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 do 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!

```
1  def main with input a output b as
2     x := 1;
3     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

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

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`

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

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

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

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

Dealing with Non-termination

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

This is why also OCaml programs are *partial* functions

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

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

MiniFun

Syntax

A MiniFun program is a term t

$$
\begin{aligned}
t \coloneqq\ & n \mid v \mid x \mid \texttt{fun}\ x\ \texttt{=>}\ t \\
& \mid t\ t \mid t\ op\ t \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \\
& \mid \texttt{let}\ x = t\ \texttt{in}\ t \mid \texttt{letfun}\ f\ x\ \texttt{=}\ t\ \texttt{in}\ t
\end{aligned}
$$

where

- $f, x, x', x'' \in X$ are variables;
- $n, n', n'' \in \mathbb{Z}$ are integer numbers $(0, 1, -1, \dots)$;
- $v, v', v'', \cdots \in \mathbb{B}$ are boolean literals (`true`, `false`);
- op is an operator `+`, `-`, `*`, `<`, `not`, `and`.

A term denotes a value, i.e. an integer, a boolean or a **function**.

Intuitive Semantics: The meaning of terms

- trivial for literals n and v
- a function `fun` x `=>` t is value itself, we must only take care of local names
- variables are resolved according to the bindings of `let` $\cdot$ `=` $\cdot$ `in` $\cdot$ and `letfun` $\cdot$ $\cdot$ `=` $\cdot$ `in` $\cdot$
- the result of performing an operation, for t *op* t
- the result of applying a function t to a term t', for $t\ t'$
- the result of the conditional, for `if` t `then` t' `else` t''
- the value denoted by t' when x is then name for t, when the term is `let` x `=` t `in` t'
- the value denoted by t' when f is then name for a **recursive** function defined as t with parameter x, when the term is `letfun` f x `=` t `in` t'

Intuitive Semantics: An example

```
1  let f =
2    fun x =>
3      fun y => x + y
4  in
5    f 3
```

```
1  letfun f x =
2    fun y => x + y
3  in
4    f 3
```

- the programs return the input integer plus 3
- the `let` construct can be used for defining non-recursive functions
- their semantics is **a function from integers to integers**

Intuitive Semantics: Another example

```
1  let f =
2    fun z =>
3      fun y =>
4        fun x =>
5          if x < 0 then y x else z x
6  in
7    f (fun x => x + 1) (fun x => x - 1)
```

- functions can be passed as parameters and returned by functions
- its semantics is **a function from integers to integers**

Intuitive Semantics: Deadlocks

```
1  letfun f x =
2    true + y
3  in
4    f 3
```

Two problems:

- the variable `y` is undefined
- the sum + is not defined between booleans and integer values

Thus:

- the program has no semantics
- the OCaml implementation should fail in this case
- we will see how to detect this problems at static time

Intuitive Semantics: A third example

```
1  letfun f x =
2    if x < 2 then 1 else x + (f (x - 1))
3  in
4    f
```

- we have recursive functions (when using `letfun`)
- we use parenthesis for representing the order of evaluation in the concrete syntax
- the abstract syntax tree already defines the order of evaluation, so please ignore them (we will parse the language later)
- its semantics is **a function from integers to integers**

Intuitive Semantics: A final example

Program A:

```
1  fun x => x + 4
```

Program B:

```
1  fun x => x and false
```

- both are legal MiniFun programs
- the semantics of program A is **a function from integers to integers**
- the semantics of program B is **a function from booleans to booleans**
- the latter will not be accepted by our interpreter (just notice that when writing tests)

Semantic Domains

Values are either (i.e. the union of)

- $\mathbb{Z}$ integers
- $\mathbb{B}$ booleans
- functions represented as **closures**, i.e. the code with the needed information for evaluating the contained variables

An **environment** $\rho : X \longrightarrow (\mathbb{Z} \cup \mathbb{B} \cup \textbf{Closures})$ is needed to represent the association between variables and values

As for memories, we will write $\rho[x \mapsto val]$ for the environment ρ' such that

$$\rho'(y) = \begin{cases} val & \text{if } y = x \\ \rho(y) & \text{otherwise} \end{cases}$$

Closures: function code and variable definitions

Two kinds of closures

- (x, t, ρ) for non-recursive functions

$$\texttt{fun}\ x\ \texttt{=>}\ t$$

$$\texttt{let}\ y = \texttt{fun}\ x\ \texttt{=>}\ t\ \texttt{in}\ \cdot$$

- (f, x, t, ρ) for recursive functions

$$\texttt{letfun}\ f\ x\ \texttt{=}\ t\ \texttt{in}\ \cdot$$

Closures: function code and variable definitions

Consider the first one (x, t, ρ)

- the name of the formal parameter x tells us how to bind the actual parameter
- the code of the function t is for computing the final result
- the environment is for resolving non-local names

Example:

```
1  let a = 1
2  in (fun y => y + a) 6
```

- the closure is $(y, y + a, \{(a, 1)\})$
- the application returns $y + a$ where y is 6 and a is 1, it gives 7

Recall: Staci scoping

MiniFun Example:

```
1  let f =
2    (let a = 1
3     in (fun y => y + a))
4  in
5    (let a = 2
6     in f 4)
```

OCaml Example:

```
1  let f =
2    (let a = 1
3     in (fun y -> y + a))
4  in
5    (let a = 2
6     in f 4)
```

- which is the value of a when evaluating the function?
- is the result 5 or 6?

Closures: function code and variable definitions

Consider the second one (f, x, t, ρ)

- x, t and ρ are used as before
- f is needed to recognize recursion: the variable f must be bounded to the closure itself!

Example:

```
1  letfun g y = g (y - 1)
2  in g 5
```

- the closure is $(g, y, g\ (y-1), \varnothing)$
- the application returns the same result of $g\ (y-1)$ where y is 5 and g is $(g, y, g\ (y-1), \varnothing)$

Formal Semantics

The semantics is given in terms of sequents $\rho \vdash t \longrightarrow val$ where

- ρ is an environment
- t is a MiniFun term
- val is some value (integer, boolean or closure)

Read as: "the semantics of t in the environment ρ is val"

The simplest cases:

$$\frac{}{\rho \vdash n \longrightarrow n}\ \textsc{Num}$$

$$\frac{}{\rho \vdash v \longrightarrow v}\ \textsc{Bool}$$

$$\frac{}{\rho \vdash x \longrightarrow \rho(x)}\ \textsc{Var}$$

Formal Semantics

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

$$\frac{\rho \vdash t_1 \longrightarrow t_1' \quad \rho \vdash t_2 \longrightarrow t_2'}{\rho \vdash t_1 \ op \ t_2 \longrightarrow t_1' \ \mathcal{O}(\text{op}) \ t_2'} \textsc{Op}$$

$$\frac{\rho \vdash t_1 \longrightarrow \texttt{true} \quad \rho \vdash t_2 \longrightarrow t_2'}{\rho \vdash \texttt{if} \ t_1 \ \texttt{then} \ t_2 \ \texttt{else} \ t_3 \longrightarrow t_2'} \textsc{IfTrue}$$

$$\frac{\rho \vdash t_1 \longrightarrow \texttt{false} \quad \rho \vdash t_3 \longrightarrow t_3'}{\rho \vdash \texttt{if} \ t_1 \ \texttt{then} \ t_2 \ \texttt{else} \ t_3 \longrightarrow t_3'} \textsc{IfFalse}$$

Formal Semantics

$$\frac{}{\rho \vdash \texttt{fun}\ x\ \texttt{=>}\ t \longrightarrow (x, t, \rho)}\ \textsc{Fun}$$

$$\frac{\rho \vdash t_1 \longrightarrow t_1' \quad \rho[x \mapsto t_1'] \vdash t_2 \longrightarrow t_2'}{\rho \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \longrightarrow t_2'}\ \textsc{Let}$$

$$\frac{\rho \vdash t_1 \longrightarrow (x, t_1', \rho') \quad \rho \vdash t_2 \longrightarrow t_2' \quad \rho'[x \mapsto t_2'] \vdash t_1' \longrightarrow t}{\rho \vdash t_1\ t_2 \longrightarrow t}\ \textsc{FunApp}$$

Formal Semantics

$$\frac{\rho[f \mapsto (f, x, t_1, \rho)] \vdash t_2 \longrightarrow t}{\rho \vdash \texttt{letfun}\ f\ x\ \texttt{=}\ t_1\ \texttt{in}\ t_2 \longrightarrow t}\ \textsc{LetFun}$$

$$\frac{\begin{array}{c}\rho \vdash t_1 \longrightarrow (f, x, t_1', \rho') \\ \rho \vdash t_2 \longrightarrow t_2' \\ \rho'[f \mapsto (f, x, t_1', \rho')][x \mapsto t_2'] \vdash t_1' \longrightarrow t\end{array}}{\rho \vdash t_1\ t_2 \longrightarrow t}\ \textsc{RecFunApp}$$

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 2. Create a module for MiniFun that exposes the type of the abstract syntax tree and an evaluation function. The evaluation function can both fail and diverge, in agreement with the semantics of MiniFun.