Laboratorio

Fondamenti di linguaggi di programmazione e specifica (2021/2022)

Outline

Goals

1. Learn a functional programming language
2. Apply notions learned in class in order to create an interpreter / type-checker for an arbitrary language (defining syntax and semantic)

Functional Programming

Functions

Functions are first class citizens

```
F(x) = x + 10;
```

Where x is an INPUT to the function F

```
F(10) = 20
F(20) = 30
```

A function might have an unlimited number of parameters as INPUT

Requirements for functional programs:

- Only functions
- No side effects
- Fixed control flow

Example

GOAL:
Write a function that takes 2 parameters as inputs and returns their sum squared

IMPERATIVE SOLUTION

```
....
```

FUNCTIONAL SOLUTION

```
....
```

Example

GOAL:

Write a function that takes 2 parameters as inputs and returns their sum squared

IMPERATIVE SOLUTION

```
define sommaquad(a,b):
    somma = a+b
    quad = (a+b) ^2
    return quad
```

FUNCTIONAL SOLUTION

```
define sommaquad(a,b):
    return (a+b)^2
```

Moscow ML: History

Moscow ML is a light-weight implementation of *Standard ML (SML)*, a strict functional language widely used in teaching and research.

- It implements the *SML Modules* language and some extensions. Moreover, Moscow ML supports most required parts of the *SML Basis Library*.
- It supports separate compilation and the generation of stand-alone executables.

Moscow ML was created by Sergei Romanenko, Claudio Russo, Niels Kokholm, Ken Friis Larsen and Peter Sestoft in the 90's.

Download Moscow ML

Offline (Recommended)

Available for Windows, Mac and Linux at: `https://mosml.org/` (current version 2.10.1)
Compiled installer at `http://www.itu.dk/~sestoft/mosml.html` (Old version)

Online

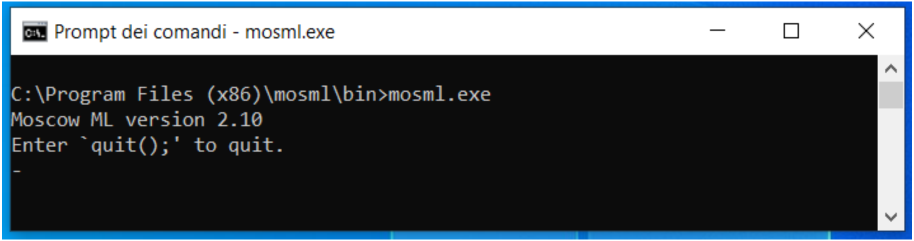
try mosml: A simple web-based environment for experimenting with SML code at `http://try.mosml.org/`

Or if the above is offline:

Online

SOSML: The Online Interpreter for Standard ML at `https://sosml.org/editor`

Moscow ML: Test



Moscow ML

- *Syntax*: how to write correct *expressions*
- *Semantics*: what is the meaning of these expressions
    1. *Type-checking* checks the static environment (types and declarations)
    2. *Evaluation* checks the dynamic environment

The *type-checking* is performed before the program execution

The *evaluation* is performed at runtime

Moscow ML

- Need ";" at the end of each expression
- To load from a file:

```
use file_name.ml;
```

or:

```
mosml.exe file_name.ml
```

Moscow ML: The interactive shell

Just type your declarations and expressions into the interactive shell and the system will respond. Here is an example session:

```
$ mosml
Moscow ML version 2.10
- 4;
> val it = 4 : int
- "hello␣" ^ "␣there";
> val it = "hello␣␣there" : string
- val x = 12 and y = ~5;
> val x = 12 : int
  val y = ~5 : int
- x - y * 2;
> val it = 22 : int
```

If you don't specify types, it figures them out for you (as long as the syntax is correct). If you type in an expression directly, it turns it into a declaration binding the result to the special name `it`.

Moscow ML

`it` refers to the value of the last expression

```
3+4;
> val it=7 :int
```

Can be called directly

```
it;
> val it=7 :int
```

Can be used in expressions

```
it+1;
> val it=8 :int
```

Moscow ML: Compiling

The Moscow ML compiler is called mosmlc.
In Windows you can enter:

```
> mosmlc test.sml -o test.exe
```

and in Unix you can enter:

```
$ mosmlc test.sml -o test
```

and you have your executable. To compile and run:

```
> mosmlc test.sml -o test.exe && test
```

or:

```
$ mosmlc test.sml && ./a.out
```

Moscow ML: Declarations

A program is a sequence of declarations. Declarations begin with:

- val, to declare a new value binding
- fun, to declare a new function
- type, to introduce a type abbreviation
- datatype, to define a new type with visible constructors
- abstype, to define a new type with hidden constructors
- exception, to define a new exception constructor
- structure, to define a new structure
- local, to make a declaration that uses private local declarations

Moscow ML: Structures

Most program entities are defined in structures. For example, the structure Timer contains the type `cpu_timer` and the functions `startCPUTimer` and `checkCPUTimer`, so you need to write `Timer.cpu_timer` and `Timer.checkCPUTimer`. But entities in the package General lik int, string, real, functions and not do not have to be qualified with the package name.

```
fun count(n) =
  if (n > 0) then count(n - 1) else ();

val _ =
   let
     val t = Timer.startCPUTimer()
   in
     count(10000000);
     print (Time.toString(#usr(Timer.checkCPUTimer
         (t))) ^ "\n")
   end;
```

Moscow ML: Reserved Words

The following words can't be used as names (in functions etc...):

`abstype`	`and`	`andalso`	`as`	`case`	`do`	`datatype`	`else`
`end`	`eqtype`	`exception`	`fn`	`fun`	`functor`	`handle`	`if`
`in`	`include`	`infix`	`infixr`	`let`	`local`	`nonfix`	`of`
`op`	`open`	`orelse`	`raise`	`rec`	`sharing`	`sig`	`signature`
`struct`	`structure`	`then`	`type`	`val`	`where`	`with`	`withtype`
`while`	`(`	`)`	`[`	`]`	`{`	`}`	`,`
`:`	`:>`	`;`	`...`	`_`	`|`	`=`	`=>`
`->`	`#`						

Moscow ML: Identifiers

Alphanumeric identifiers

a sequence of letters, digits, primes (′) and underbars (_) starting with a letter or prime

Symbolic identifiers

any non-empty sequence of the following symbols: `! % & $ # + - / : < = > ? @ \ ~ ' ^ | *`

Reserved words are excluded.

Moscow ML: Types

- **Primitive types**
  unit, int, real, char, string, ..., instream, outstream, ...
- **Composite types**
  unit, tuples, records, function types
- **Datatypes**
  types and n-ary type operators, tagged unions, recursive, nominal type equality, bool, list, user defined: trees, expressions, etc.
- **Type Abbreviations**
  types and n-ary type operators, structural type equality, type ‘a pair = ‘a * ‘a

Moscow ML: Integer

Integer

```
6;
> val it=6 :int
```

Negative number

```
~6;
> val it=~6 :int
```

Operation

```
6+~6;
> val it=0 :int
```

Division

```
6 div 3;
> val it=2 :int
```

Moscow ML: Real

Real

```
6.0;
> val it=6.0 :real
```

Negative number

```
~6.0;
> val it=~6.0 :real
```

Operation

```
6.0+~6.0;
> val it=0.0 :real
```

Division

```
4.0/6.0;
> val it=0.66667 :real
```

Moscow ML: More operators

- `div` Integer division
- / "Regular" division
- ~ Less than zero
- `round(4.5)= 4` Integer
- `trunc(4.5)= 4` Integer
- `ceil(4.5)= 5` Integer
- `floor(4.5)= 4` Integer
- `real(6) Real number`

Moscow ML: Example Program

1. Create a new file called "example.ml"
2. Copy and paste the following instructions

```
val a = 3;
val b = 10;
val c = 1;
val d = (a+b) + (c+1+a);
val sse = if b<a then b+10 else a+1;
(* For the IF clause the type-checker checks
   for the guard to be type BOOL, and that both
    branches have the same type *)
```

3. Open Moscow ML interactive shell and type: `use "example.ml";`

Moscow ML: Questions

For each expression we encounter we need to ask ourselves the following:

1. **What is its syntax?**
   i.e., how do I write it down?
2. **What is its semantics?**
   2.1 *Type-checking Rules*
   i.e., what is the type? What could make the type-checking fail?
   2.2 *Evaluation Rules*
   i.e., what is the value?

Moscow ML: Example Addition

1. **Syntax**
   Example:

   `expression1 + expression2`

2. **Semantics**
   2.1 **Type-checking**
   If **expression1** and **expression2** are **Int**
   Then **expression1 + expression2** is **Int**
   2.2 **Evaluation**
   If **expression1** evaluates to **1** and **expression2** evaluates to **1**
   Then **expression1 + expression2** evaluates to **2**

$$(\text{op } +)\ \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}$$

Moscow ML: Example Conditional

1. **Syntax**

   ```
   IF expression1 THEN expression2 ELSE expression3
   ```

   where if, then and else are keywords

2. **Semantics**

   2.1 **Type-checking**
   expression1 must have type **BOOL**
   expression2 and **expression3** can have any type, but it must be the same. This will also be the type of the entire conditional expression

   2.2 **Evaluation**
   If **expression1** evaluates to **TRUE** Then **expression2** is evaluated Else **expression3** is evaluated

Moscow ML: Example Pair

1. **Syntax**
   Example:

   `(expression1, expression2)`

2. **Semantics**
   2.1 **Type-checking**
   If **expression1** has type **t1** and **expression2** has type **t2**, then the pair has type **t1 * t2**
   2.2 **Evaluation**
   Evaluate **expression1** to **value1** and **expression2** to **value2**, the result is **(value1,value2)**

To access the pieces of the pair you can use `#1 e` to access the first part of the pair and `#2 e` to access the second part of the pair

Moscow ML: Tuples

1. **Syntax**
   Example:

   ```
   (expression1, expression2,..,expressionn)
   ```

   To access the different pieces of the tuples you can use the same principle as for pairs.

   ```
   val x1 = (7,(true,9));
   (* int * (bool*int) *)

   val x2 = #1 (#2 x1)     (* bool *)
   val x3 = (#2 x1)        (* bool*int *)
   ```

Moscow ML: Tuples

```
val x1 = (7,(true,9),4,(false,4,3.4));
val x4 = #1 (#4 x1)
```

What is the type of x4?

- `> val x4 = false : bool`
- `> val x4 = true : bool`
- `> val x4 = 4 : int`
- `> val x4 = 3 : int`

Moscow ML: Shadowing

```
val a = 14;
val b = 10 * a;
val a = 3;
val c = b;
```

What does the dynamic environment look like?

Moscow ML: Functions

Functions are just regular values who happen to have a type like 'a $- >$ 'b.
Functions must have exactly one parameter and exactly one return type.
The syntax of a function is:

```
fn pattern => exp ('|' pattern => exp)*
```

Function calls are straighforward:

```
(fn x => x + 6) 20;
(fn [] => false | (h::t) => true) [4, 5, ~3];
(fn _ => 0) (4, "dog", [4.5, 1.1], (1,1));
(fn 5 => 0 | x => x) 5;
(fn 5 => 0 | x => x) 12;
```

Moscow ML: Bindings

Declaration: binding name to a value

- variable bindings

```
val x=3
val y=x+1
```

- function bindings

```
fun fact n=
if n=0 then 1
else n*fact(n-1)
```

Moscow ML: Defining Functions 1/3

Since **functions** are values, you declare them just like any other value.
You can take advantage of the fact that each declaration can use previous declarations:

```
val two = 2;
val addSix = fn x => x + two + 4;
val square = fn x:real => x * x;
val magnitude = fn (x,y) => Math.sqrt(square x +
    square y);
val rec fact = fn n => if n = 0 then 1 else n *
    fact (n-1);
```

Moscow ML: Defining Functions 2/3

You can use `fun` for convenience (it's basically val rec):

```
fun addSix x = x + 6;
fun square (x:real) =  x * x;
fun magnitude (x,y) = Math.sqrt(square x + square
    y);
fun fact n = if n = 0 then 1 else n * fact (n-1);
```

Therefore: Function: `fn var => exp`
function with formal parameter `var` and body `exp`

```
val inc = fn x => x+1
```

is equivalent to

```
fun inc x= x+1
```

Moscow ML: Defining Functions 3/3

You can use multiple patterns with fun too.
It really helps when writing recursive functions.

```
fun fact 0 = n
  | fact n = n * fact(n - 1);

fun pow (0, _) = 1.0
  | pow (n, x) = x * pow(n - 1, x);
```

Moscow ML: Example Function

1. **Syntax**
   fun x (x1: type1,...,xn: typen) = **expression**
2. **Semantics**
   2.1 **Type-checking**
   If **expression** type-checks to have a type **type**
   Then **x : (type1*...*typen)** -> **type** is added
   2.2 **Evaluation**
   A function is a value, therefore **x** is added to the dynamic environment to be called later

Moscow ML: High Order Functions

Definition 1 (High Order Functions)

A higher order function is one that takes a function as a parameter and/or returns a function as its result.

Let's see how High Order Functions are used with the help of the Currying and Uncurrying in functions.

Moscow ML: Currying and Uncurrying

The functions

- `fn (x,y)=> x + y`
- `fn x => fn y => x + y`

are similar.
The former is the uncurried version, the latter is curried.
Curried functions allow “partial application”:

```
fun add x y = x + y;
fun addSix = add 6;
add 6 5;        (* returns 11 *)
addSix 5;       (* also returns 11 *)
```

Moscow ML: Partial Application 1/3

Let's take a look at the following expressions

```
val plus = fn (a, b) => a+b ;
plus (2,3);
```

As we already know the result will be:

```
> val plus = fn : int * int -> int
> val it = 5 : int
```

What happens if we write the following instruction?

```
plus 2;
```

Moscow ML: Partial Application 2/3

Answer: we get an error

```
! plus (2);
! ^
! Type clash: expression of type
! int
! cannot have type
! int * int
```

Why?

Because the type of the function plus is `int * int -> int`

It expects a type `int * int` as argument

What would have happened if we had defined the function plus as follows?

```
val plus = fn a => fn b => a+b;
(* Or fun plus a b = a+b; *)
plus 2;
```

Moscow ML: Partial Application 3/3

Answer: it would have worked

```
> val plus = fn : int -> int -> int
> val it = fn : int -> int
```

Why?

Because the type of the function plus is `int -> int -> int`

It expects a type `int` as argument

Let's give it a name and use it

```
val piudue = plus 2;
piudue 1;
piudue 5;
```

Answer:

```
> val piudue = fn : int -> int
> val it = 3 : int
> val it = 7 : int
```

Moscow ML: High order function recall

As we already said:
A **higher order function** is one that takes a function as a parameter and/or returns a function as its result.

Therefore, as we saw:
the *curried* `plus` function above is a higher order function, because it takes in an integer and returns a function.

Moscow ML: Type inference

Moscow ML type-inference algorithm in action:

```
[true, true, false]   (*  bool list *)
[]  (* 'a list *)
fn s => length s * 2  (* 'a list -> int *)
fn x => (15.2, x)  (*  'a -> real * 'a *)
{x = 3, y = 5}  (* {x : int, y : int} *)
fn x => fn y => fn z => (z, x, y)
(* 'a -> 'b -> 'c -> 'c * 'a * 'b  *)
fn (x, y) => x = y; (* ''a * ''a -> bool *)
(fn s => SOME s)[]  (* 'a list option *)
```

Types with a double apostrophe prefix are equality types. Equality doesn't work on all types. For example, the type `int->int` will match (unify) with 'a, but it will not unify with ''a. (More details within a slide)

Moscow ML: Polymorphic types

Definition 2 (Polymorphic types)

A type that contains one or more type variables is called polymorphic
-Graham Hutton, University of Nottingham

```
fun ident x=x;
> val ident=fn :'a->'a [type variable]
fun pair x=(x,x);
> val pair=fn :'a->'a * 'a [polymorphic type]
fun fst (x,y)=x;
> val fst=fn :'a * 'b -> 'a
val foo=pair 4.0;
> val foo :real*real
fst foo;
> val it=4.0 :real
```

Moscow ML: Equality type ''a 1/4

Remark

The type variable ''a indicates that the operands of $=$ must have equality types.

Let's compare two integers:

```
val oneIsOne = (1 = 1);
```

What will be the type of oneIsOne?

Moscow ML: Equality type ”a 2/4

Answer:

```
> val oneIsOne = true: bool;
```

What about the following?

```
val oneIsOne = (1.0 = 1.0);
```

Moscow ML: Equality type ”a 3/4

Answer (Unfortunately):

```
Elaboration failed: Type clash. Functions of type
    "''a␣*␣''a␣   ␣bool" cannot take an argument of
      type "real␣*␣real": Type "real" does not admit
      equality.
```

The type-checker found an error.
Equality is defined:

- NaN is not equal to NaN
- 0.0 is equal to 0.0
- 1/0.0 is not equal to 1/ 0.0

The language designers decided that they didn’t want = to differ from the semantics specified by IEEE 754,
but that they also didn’t want the semantics of = to be different for real than for all other types in the language.
So, they completely eliminated = for real.

Moscow ML: Equality type ''a 4/4

How do I compare reals?

Remark

One can use `Real.==` to compare reals for equality, but beware that this has different algebraic properties than polymorphic equality.

```
Real.== (0.1, 0.2)
> val it = false : bool
```

Moscow ML: Defining a Custom Type 1/2

The type declaration does not make a new type, just a type abbreviation:

```
type intpair = int * int;
type person =
{name: string, birthday: date, weight: real};
type simplefun = real -> real;
type text = string;
```

For the IF clause the *type-checker* checks for the guard to be type BOOL, and that both branches have the same type

Moscow ML: Defining a Custom Type 2/2

You need to use datatype or abstype to create a new type, distinct from any other. Datatypes are defined with constructors:

```
datatype weekday = Monday | Tuesday | Wednesday | Thursday | Friday;

datatype file_descriptor = Stdin | Stdout | Stderr | Other of int;

datatype 'a tree = Empty
                 | Node of 'a * 'a tree * 'a tree;

datatype Shape = Circle of {radius: real}
               | Rectangle of {height: real, width: real}
               | Polygon of (real * real) list;
```

Moscow ML: Datatypes

DATATYPES

Used to introduce new types

```
datatype colour = red | blue | green;
```

Introduces the type *colour*
and three constructors for that type *red*, *blue* and *green*.
"|" can be intended as "OR"

A value of a **new type** introduced by datatype will always be composed by:

1. The *TAG*, which identifies the constructor used
2. The *DATA*, which is the value (can be none)

Moscow ML: Datatypes Example

```
datatype TipoEsempio = PAIO of int * int
                     | TESTO of string
                     | LIBERO
```

```
1 val x = PAIO(1+2,2+5);
2 val y = TESTO "foo";
3 val z = LIBERO;
```

Type-checking

Line 1: *x*, is a function from type **int * int** to type **TipoEsempio**

Line 2: *y*, is a function from type **string** to type **TipoEsempio**

Line 3: *z*, is not a function. It's type is **TipoEsempio**

(To extract the pieces of the datatype case expressions are used)

Moscow ML: Case Of

Before seeing how to extract constructs and data values from datatypes

```
case expression0 of    p1 => expression1
                     | p2 => expression2
                     | .....
                     | pn => expressionn
```

Each pattern is a constructor name followed by the right number of variables.
For example: *Constructor1* or *Constructor2 x* or *Constructor3 (x,y)* or ...

Moscow ML: Datatypes Extract

```
datatype TipoEsempio = PAIO of int * int
                          | TESTO of string
                          | LIBERO
```

```
fun extr (x: TipoEsempio) =
        case x of
            LIBERO => 1
            | TESTO s => 2
            | PAIO (i1,i2) => 3
> val extr = fn : TipoEsempio -> int
```

```
extr(PAIO (1,2));
> val it = 3 : int
```

(see Binary Tree Example File)

Moscow ML: Tuples

```
datatype TipoEsempio = PAIO of int * int
                            | TESTO of string
                            | LIBERO

fun extr (x: TipoEsempio) =
        case x of
            LIBERO => 1
            | TESTO s => 2
            | PAIO (i1,i2) => 3

x= LIBERO;
```

What is the result of val extr x;?

- 1
- 2
- 3
- Error

Moscow ML: Operators

Operators are really just functions that are given precedence and fixity in a *infix, infixr or nonfix* directive.

These are some of the *infix* operators defined in the standard library.

Note that: many of the built-in operators are overloaded, so the specification has “fake types” — wordint means either int, word or word8; num means wordint or real; numtxt means num, char or string.

Moscow ML: Boolean Operators

`e1 andalso e2`

- **Type-checking** e1 and e2 must have type bool
- **Evaluation** if result of e1 is false then false, else result of e2

`e1 orelse e2`

- **Type-checking** e1 and e2 must have type bool
- **Evaluation** if result of e1 is true then true, else result of e2

`not e1`

- **Type-checking** e1 must have type bool
- **Evaluation** if result of e1 is true then false, else true

Moscow ML: Boolean Operators

The following are not functions:

```
e1 andalso e2
```

```
e1 orelse e2
```

```
orelse;;
> Error: syntax error
andalso;;
> Error: syntax error
(* NOTE: some branches of the orelse/andalso
    might be NEVER evaluated *)
```

The following is a function:

```
not e1
```

```
not;
> val it = fn : bool -> bool
```

Moscow ML: Comparison Operators

The following are not functions:

= <> > < >= <=

Remember:

- >,<,>=,<= can not be used with int and real in the same expression
- =,<> can not be used with real (as we already saw)

```
5>1;
> val it = true :bool
5.0>1.0;
> val it = true :bool
5.0>1;
> Error: Type Error
(* You can call Real.fromInt to translate a
    integer into a real *)
```

Moscow ML: Custom Operators

And you can define your own operator:

```
fun |-*-| (x, y) = 2 * x + 3 * y;

infix 5 |-*-|;

10 |-*-| 4;                    (* returns 32 *)

1 |-*-| 2 |-*-| 3;             (* returns 25 *)

1 |-*-| (2 |-*-| 3);           (* returns 41 *)

5 + 3 |-*-| 6 before 8;        (* returns 34 *)
```

Moscow ML: LET

Let expressions: local definitions

LET ___bindings__ IN ___body_of_let___ END

- test1: type int->int

```
    fun test1 a=
    let
        val x= if a>0 then a else 34
        val y= x+a+3
    in
        if x>y then x*2 else y*y
    end
```

TYPE CHECKING:

branch THEN ->INT
branch ELSE ->INT
IF ->INT
LET BODY ->INT

Moscow ML: LET example

Let expressions: local definitions

- test2: type unit->int

```
fun test2 n=
let
    val x = 1
in
    (let val x=2 in x+1 end) +
    (let val y=2+x in y+1 end)
end
```

Moscow ML: Shadowing

```
datatype TipoEsempio = PAIO of int * int
                     | TESTO of string
                     | LIBERO
(* x is a variable with type TipoEsempio *)
case x of
  PAIO(x,y) => f x
 | LIBERO
```

In the branch of the case expression show below, what is the type of the x that is passed to the function f?

- int
- PAIO
- TipoEsempio
- 12

Moscow ML: Shadowing

x:int

x is shadowed inside this case branch, and thus has type int, the type of the first thing carried by a value made from the PAIO constructor.

```
val a = 1;
val b = a; (* b is bound to 1 *)
val a = 4;
```

There is no way to "assign to" in ML
You can only shadow previous bindings in a later environment

Moscow ML: Shadowing

What value is bound to the variable **a** after the following code has been run?

```
val x = 12;
val n = 2 + x;
val x = n - 14;
val n = n * x;
val b = if n = x then 8 else 5;
val a = if b = 5 then x else b;
```

- 8
- 2
- 14
- 0

Moscow ML: Errors

Remark (Not allowed: Real+Integer)

```
5+1.0;
> Error
```

Three possible kind of errors:

- **Syntax** Wrote something incorrectly
- **Type-checking** Wrote using the correct syntax but it doesn't follow the type-checking rules for that construct
- **Evaluation** Wrote something that does type-check but it produces a wrong answer or it goes in an infinite loop

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

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

L1: Semantics

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

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

L1: Semantics

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

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

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

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

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

L1: Semantics

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

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

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

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

L1: Typing

Types of expressions:

$$T \quad ::= \quad \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit}$$

Types of locations:

$$T_{loc} \quad ::= \quad \mathsf{intref}$$

$$\text{(int)} \quad \Gamma \vdash n{:}\mathsf{int} \quad \text{for } n \in \mathbb{Z}$$

$$\text{(bool)} \quad \Gamma \vdash b{:}\mathsf{bool} \quad \text{for } b \in \{\mathbf{true}, \mathbf{false}\}$$

$$\text{(op }+\text{)} \quad \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{int} \\ \Gamma \vdash e_2{:}\mathsf{int}\end{array}}{\Gamma \vdash e_1 + e_2{:}\mathsf{int}} \qquad \text{(op }\geq\text{)} \quad \frac{\begin{array}{c}\Gamma \vdash e_1{:}\mathsf{int} \\ \Gamma \vdash e_2{:}\mathsf{int}\end{array}}{\Gamma \vdash e_1 \geq e_2{:}\mathsf{bool}}$$

$$\text{(if)} \quad \frac{\Gamma \vdash e_1{:}\mathsf{bool} \qquad \Gamma \vdash e_2{:}T \qquad \Gamma \vdash e_3{:}T}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3{:}T}$$

L1: Typing

$$(\text{deref}) \quad \frac{\Gamma(\ell) = \mathsf{intref}}{\Gamma \vdash !\ell{:}\mathsf{int}}$$

$$(\text{skip}) \quad \Gamma \vdash \textbf{skip}{:}\mathsf{unit}$$

$$(\text{seq}) \quad \frac{\Gamma \vdash e_1{:}\mathsf{unit} \qquad \Gamma \vdash e_2{:}T}{\Gamma \vdash e_1; e_2{:}T}$$

$$(\text{while}) \quad \frac{\Gamma \vdash e_1{:}\mathsf{bool} \qquad \Gamma \vdash e_2{:}\mathsf{unit}}{\Gamma \vdash \textbf{while}\ e_1\ \textbf{do}\ e_2{:}\mathsf{unit}}$$