Programming Language Semantics: Chapter 04 - Operational and Axiomatic Semantics

Tarek Boutefara – t_boutefara@univ-jijel.dz – Version 0.1, 20-11-2024

Table of Contents

1. Introduction

To be able to practise language semantics, we need an experimental language. In fact, ‘complete’ programming languages are too complex and have very rich syntaxes that evolve continuously.

In the context of this course, it will be difficult to include all the syntax and API of a complete general-purpose language such as Java or C. So we’re going to look at L1, a reduced and limited language but one that offers, at the same time, enough elements to experiment with.

After the presentation of L1, we will look at operational semantics. Operational semantics is the first type of semantics. It represents the meaning of a program in terms of the computational steps it performs in an idealised execution.

2. Presentation of L1

L1 is an imperative programming language for manipulating integers. It offers a storage system (locations) where each location stores an integer. The language also has a conditional structure and the "while" loop.

2.1. Syntax

2.1.1. Definition

- Booleans $b \in \mathbb{B} = \{true, false\}$
- Integers $n \in \mathbb{Z} = \{..., -2, -1, 0, 1, 2, ...\}$
- Locations $l \in \mathbb{L} = \{l_0, l_1, l_2, l_3, l_4, ...\}$
- Operations $op : := + \mid \geq$
- Expressions

e ::= n | b | e_1 op e_2 | e_1; e_2 |
l := e | !l |
if e_1 then e_2 else e_3 |
while e_1 do e_2 |
skip

2.1.2. Booleans

The set of Booleans contains only two values, ‘true’ and ‘false’.

2.1.3. Integers

The set of integers corresponds to the mathematical set $\mathbb{Z}$.

2.1.4. Locations

Locations refer to the memory areas in which integers are stored. The syntax defines names of the form l_i.

2.1.5. Les opérations

For this reduced syntax, we define two operations:

- $+$ the addition operation, it takes two operands and returns their sum,
- $\geq$ the ‘greater than or equal to’ binary operator, which returns "true" if the first operand is greater than or equal to the second operand.

2.1.6. Expressions

- An integer n is an expression,
- A boolean b is an expression,

- Applying an operation is an expression :
  - $x + y$,
  - $x \geq y$,
- Sequencing two expressions is an expression. Sequencing is defined by a semi-colon ';',
- The assignment is an expression. It is represented by the symbol ':='. The left part of the assignment is a location and the right part is an expression such as $l_1 := 1 + 10$
- Access to the value of the rental: if l_i refers to the rental itself, its value is obtained by $!l_i$,
- The traditional If...Else...Endif structure,
- The traditional While loop.

2.2. Example

This program calculates the sum of integers from 1 to 10.

```
l1 := 10;
l2 := 0;

while !l1 ≥ 1 do(
    l2 := !l2+!l1;
    l1:= !l1 + -1)
```

- Note the construction of a block using parentheses.
- Note also that we don't have subtraction, so we use sum with negative integers.

3. Structural Operational Semantics

3.1. Presentation

There are two approaches to operational semantics:

- **Small step:** this approach describes the individual execution steps of a program. This approach is also known as Structural Operational Semantics.
- **Big step:** this approach describes the overall result of execution

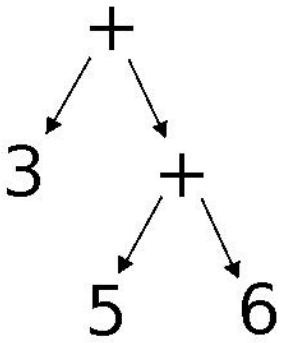
We can define small-step semantics for a language L by defining two components:

- A set S of execution states,
- A transition system on S that links each state to all the states that can be reached by executing a single step.

If two states s and s' are linked, then there is a transition from s to s' that we note $s \rightarrow s'$. For example in the case of an arithmetic expression, we can have this transition:

- $3 + (5 + 6) \rightarrow 3 + 11$
- $3 + 11 \rightarrow 14$

This evaluation can be represented in the form of an evaluation tree:



In this way, operational semantics can be seen as the discovery of an execution tree.

3.2. Storage space definition

We can define a storage space as a finite partial function of the set of memory spaces (or locations) $\mathbb{L}$ to the set of integers $\mathbb{Z}$.

For example, let be the storage space with two locations that saves two values. It can be defined by :

- $\{l_1 \mapsto 54, l_3 \mapsto 62\}$

3.3. Transition system definition

A transition system consists of a :

- A Configuration set, and

- A binary relation → in Configuration * Configuration.

The elements of the Configuration set are called configurations or states.

The relation → is called transition or reduction. We note `c → c'` to designate that c can make a transition to c'.

Transitions can be stopped when there is no state c' to which the current state c can transition.

That is:

- $\neg\ \exists c' \,.\, c \rightarrow c'$

In this case, we note:

- $c \not\rightarrow$;

A configuration can be seen as a pair $\langle e, s \rangle$ which consists of an expression e and a storage space s. Thus, a transition can be defined by:

- $\langle e, s \rangle \rightarrow \langle e' , s' \rangle$

A transition is a single execution step (of the computation).

This transition relation is said to be deterministic if for each state c there exists at most one state c' such that $c \rightarrow c'$. This property can also be written in the following form:

- $\forall c.\ \forall c' , c' '\,.\, (c \rightarrow c' \ \wedge c \rightarrow c' ') \Rightarrow c' \ = c' '$

3.4. Operational semantics of the L1 language

The L1 language defines the set of locations $\mathbb{L}$, thus, the notion of storage space. The L1 language also defines the 'skip' instruction to designate the end of execution.

We note that execution also stops when there is no state to which it is possible to transit from the current state. This can be translated by the appearance of an execution error (such as: 10 + false).

3.4.1. Operations: $+$ and $\geq$

We can define the operational semantics of operations in L1 by the following rules :

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

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

(op1) $$\frac{\langle e_1, s \rangle \rightarrow \langle e'_1 , s' \rangle}{\langle e_1 \ \ \text{op} \ \ e_2, s \rangle \rightarrow \langle e'_1 \ \ \text{op} \ \ e_2, s' \rangle}$$

(op2) $$\frac{\langle e_2, s \rangle \rightarrow \langle e'_2 , s' \rangle}{\langle v \ \ \text{op} \ \ e_2, s \rangle \rightarrow \langle v \ \ \text{op} \ \ e'_2 , s' \rangle}$$

3.4.2. Dereferencing

We define the dom function on the storage space s. It returns the list of currently defined locations. The dereferencing function '!' can be defined using the following semantics:

(deref) $\langle\, !l, s \rangle \rightarrow \langle n, s \rangle$ if $l \in dom(s) \quad \text{et} \quad s(l) = n$

Indeed, if $l \notin dom(s)$, we have the equivalent of the 'variable l has not been declared' error.

3.4.3. Assignment ":="

The assignment is the instruction used to change a value in the storage space. For example, if l is a location and v1 and v2 are two values and there is no instruction after the assignment :

- $\langle l := v2, \{l \mapsto v1\} \rangle \rightarrow \langle skip, \{l \mapsto v2\} \rangle$

The assignment can go through several steps if the expression on the right is not a value.

(assign1) $\langle l := n, s \rangle \rightarrow \langle skip, s + \{l \mapsto n\} \rangle$ if $l \in dom(s)$

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

Example :

- $\langle l := 8 + \ !l, \{l \mapsto 54\} \rangle \rightarrow \langle l := 8 + 54, \{l \mapsto 54\} \rangle$
- $\langle l := 8 + 54, \{l \mapsto 54\} \rangle \rightarrow \langle l := 62, \{l \mapsto 54\} \rangle$
- $\langle l := 62, \{l \mapsto 54\} \rangle \rightarrow \langle skip, \{l \mapsto 62\} \rangle$
- $\langle skip, \{l \mapsto 62\} \rangle \not\rightarrow$

3.4.4. Sequencing

(seq1) $\langle skip; e_2, s \rangle \rightarrow \langle e_2, s \rangle$

(seq2) $$\frac{\langle e_1, s \rangle \rightarrow \langle e'_1 , s' \rangle}{\langle e_1 ; e_2, s \rangle \rightarrow \langle e'_1 \,; e_2, s' \rangle}$$

3.4.5. Conditional structure

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

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

(if3) $$\frac{\langle e_1, s\rangle \rightarrow \langle e'_1, s'\rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, s\rangle \rightarrow \langle \text{if } e'_1 \text{ then } e_2 \text{ else } e_3, s'\rangle}$$

3.4.6. While loop

(while) $\langle \text{while } e_1 \text{ do } e_2, s\rangle \rightarrow \langle \text{if } e_1 \text{ then } (e_2; \text{while } e_1 \text{ do } e2) \text{ else } skip, s\rangle$

3.4.7. Exercise

Let be :

- $e = (l_2 := 0; \text{while } !l_1 \geq 1 \text{ do } (l_2 := !l_2 + !l_1; l_1 := !l_1 + -1))$
- $s = \{l_1 \mapsto 3, l_2 \mapsto 0\}$

Evaluate $\langle e, s\rangle$

3.5. Operational semantics and language design

In the semantic definition of the different elements of the language, we have already made decisions concerning the order of evaluation and the result of the different instructions.

3.5.1. Order of evaluation of operations

For example, the order defined for the evaluation of an expression $e_1 \text{ op } e_2$ specifies that the first operand must be completely evaluated before starting to evaluate the second operand. This is a left-to-right evaluation. It is possible to define a right-to-left evaluation by changing the operational semantics of the operations as follows:

(op1b) $$\frac{\langle e_2, s\rangle \rightarrow \langle e'_2, s'\rangle}{\langle e_1 \text{ op } e_2, s\rangle \rightarrow \langle e_1 \text{ op } e'_2, s'\rangle}$$

(op2b) $$\frac{\langle e_1, s\rangle \rightarrow \langle e'_1, s'\rangle}{\langle e_1 \text{ op } v, s\rangle \rightarrow \langle e'_1 \text{ op } v, s'\rangle}$$

3.5.2. Assignment result

Another example of the design choices made in the programming language is the result of the assignment. By reviewing the definition of (assign1), we see that the assignment returns nothing. In conjunction with the sequencing rule, the $skip$ returned by the assignment allows the evaluation to continue. However, according to the definition given, it is not possible to have an instruction of the form:

- $l_1 := l_2 = v$ where v is a value.

This instruction is valued at :

- $l_1 := skip \not\rightarrow$

This state cannot be evaluated.

To support the above instruction, a new assignment rule and a new sequencing rule can be defined as follows:

- (assign1b) $\langle l := v, s\rangle \rightarrow \langle v, s + \{l \mapsto v\}\rangle$ if $l \in dom(s)$
- (seq1b) $\langle v; e_2, s\rangle \rightarrow \langle e_2, s\rangle$

This new definition is widely used in languages including C, C++ and Java.

3.5.3. Implicit declaration of variables

The assignment also defines a condition. This is $l \in dom(s)$. This means that the assignment requires a prior definition of the locations. This can be seen in the Pascal language, for example, where all variables must be declared before starting the main program.

It is possible to define the possibility where $l \notin dom(s)$ and the ability to add new rentals to $dom(s)$.

3.6. Other elements of Operational Semantics

The definition of a complete operational semantics requires the definition of rules for other elements, even for an experimental language like L1. For example :

3.6.1. Variable typing

The semantics can be enriched by adding variable typing (locations). In fact, the defined demantics does not allow :

- Store data other than integers, so the expression $l := false$ cannot be evaluated,
- It is possible to write the expression $2 + (5 \geq 3)$, however, it will evaluate to $2 + false$ which cannot be evaluated.

In the case of L1, we can define three types:

- The integer type,
- The Boolean type,
- A type for other units such as $skip$.

Thus, we can define the set:

- $T = \{\text{int, bool, unit}\}$

As syntax, we can add the type by:

- T ::= int | bool | unit

It is possible to add the type as a premise to their definitions given earlier:

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

$$(\text{op } \geq)\ \frac{\Gamma \vdash e_1 : \text{int}, \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \geq e_2 : \text{bool}}$$

3.6.2. Definition of functions

This first version of L1 does not define the notion of 'functions'. It is possible to extend this first version to obtain a second version, L2, which supports the declaration of functions.

Defining functions requires extending the syntax to support the notion of variable :

- $x \in \mathbb{X}$

We also need to extend the expressions:

- $e ::= \ldots | \text{fn} \quad x : T \Rightarrow ; e | e_1 e_2 \ | \ x$

The type can be:

- $T ::= \text{int}|\text{bool}|\text{unit} \ | \ T_1 \rightarrow T_2$

This syntax allows the definition of function like:

- $(\text{fn} \quad x : \text{int} \Rightarrow x + 1)$
- $(\text{fn} \quad x : \text{int} \Rightarrow (\text{fn} \quad y : \text{int} \Rightarrow x + y))$

These two functions are equivalent to the following C functions:

```
int succ(int x){
    return x + 1;
}

int somme(int x, int y){
    return x + y;
}
```

Or in Javascript :

```
let succ = (x) => x + 1

let somme = (x) => (y) => x + y
```

4. Axiomatic Semantics (Program Proof)

Axiomatic or Logical Semantics is aimed at verifying (proving) programs. This semantics makes it possible to follow the execution of a program without influencing the definition of the language itself; this differs from the Operational Semantics we saw in the previous section.

4.1. Program proof

The purpose of program proof is to check whether a program written in a given language really does what it is supposed to do.

Example:

Consider the program :

```
S := 0
N := 1

While (Not (N = 101)) Do
    S := S + N;
    N := N + 1;
EndWhile
```

This program calculates the sum $\sum_{1 \leq i \leq 100} i$. It is possible to check its value by performing a step-by-step execution based on the rules defined in the Operational Semantics of the language used.

However, if we change the program so that it becomes:

```
Read(P)

S := 0
N := 1

While (Not (N = P)) Do
    S := S + N;
    N := N + 1;
EndWhile
```

This small modification generalises the calculation and the new formula is: $\sum_{1 \leq i \leq P} i$.

It is difficult to run for all values of P given that the user can introduce an infinite number of values (a different value for each run). Thus, this testing approach no longer becomes sufficient to prove that a program does exactly what is expected of it.

So, to prove a program, it is necessary to use a more logical proof system. In this context, Hoare (or Floyd Hoare) logic is designed to solve this problem. As we introduced in Chapter 2, it is a formal system that defines a set of rules for proving a program.

In addition to proving programs, Hoare's logic can be used to define the semantics of program structures using 'axioms', hence the name ' Axiomatic Semantics'.

4.2. Program Correctness

The correction of a program mentioned above is called total correction. It specifies that the programme returns the correct value. It can be seen in two parts:

- The programme returns a value,
- If the programme returns a value, this value is correct.

The first point is called the 'termination' of the programme. We prove that the programme terminates and does not continue to run ad infinitum (infinite loop). It is possible to prove that the function converges and does not diverge.

The second point is 'partial correction'. The total correction is obtained by combining the two points.

4.3. Application of Hoare logic: Proof tree and program annotation

To follow the axiomatic semantics, it is possible to opt for two representations:

- Write a proof tree: the neouds are Hoare triplets that represent the different structures present in the program to be proved. The extraction of the different neouds is based on sequencing, conditionals and loops.
- Annotate the program: i.e. add notes related to its proof. Each note is a distributed Hoare triplet.

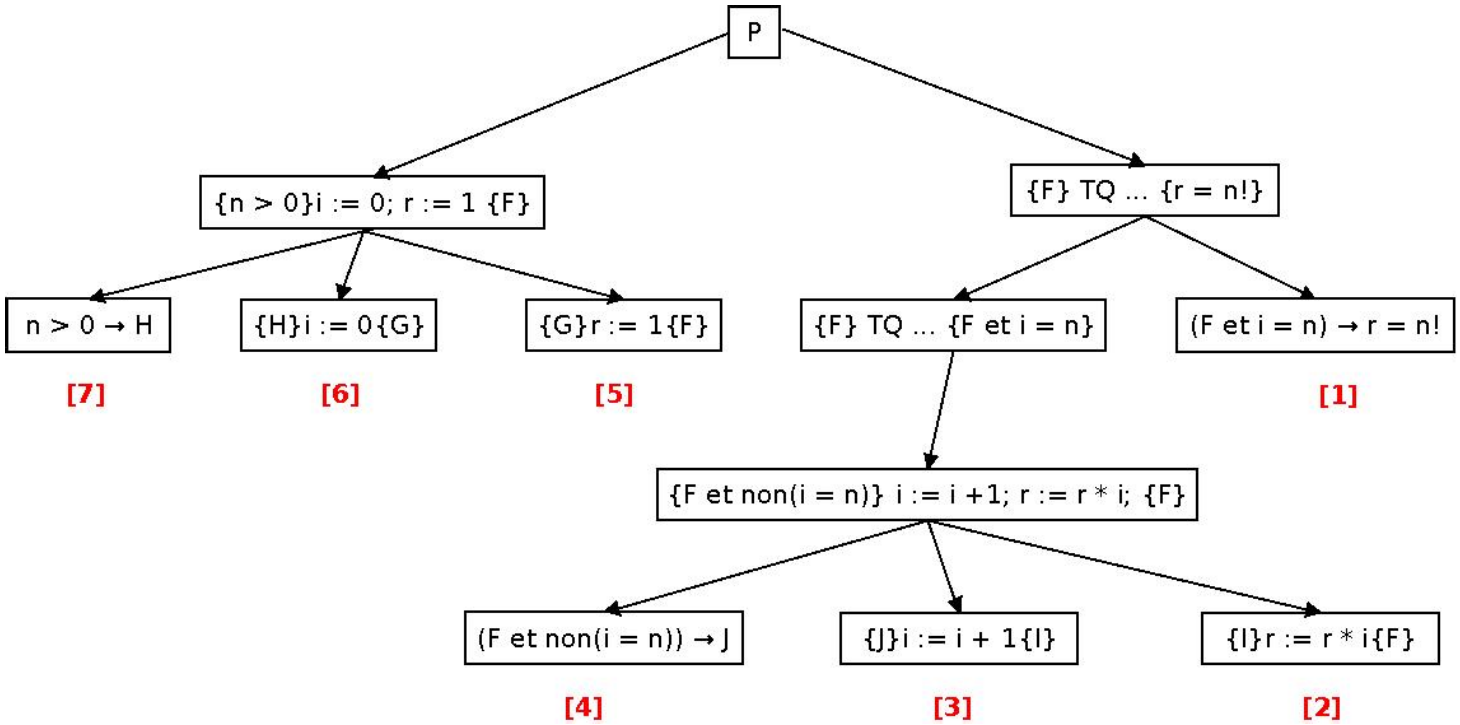
Consider the following program:

```
i := 0
r := 1

While (Not (i = n)) Do
    i := i + 1
    r := r * i
EndWhile

return r
```

4.3.1. Example of a proof tree



In general, the proof is made in reverse order. You start by proving [1], then [2], and so on.

In this tree, F is called the invariant of the loop and is the key to the problem to be solved. We need to propose an invariant and try to build the proof around it.

In this example, we can take:

- F = (r = i!)

4.3.2. Example of annotation

```
{True}
f1 i := 0
{e1}
f2 r := 1
{e2}

While (Not (i = n)) Do
    {e3}
    f3 i := i + 1
    {e4}
    f4 r := r * i
    {e5}
f5 EndWhile
{e6}
f6 return r
{r = n!}
```

The aim is to find the expressions e_i and show the validity of the formulae $f_i : \{e_i\}code\{e_i + 1\}$.

4.3.3. Example of a programme proof

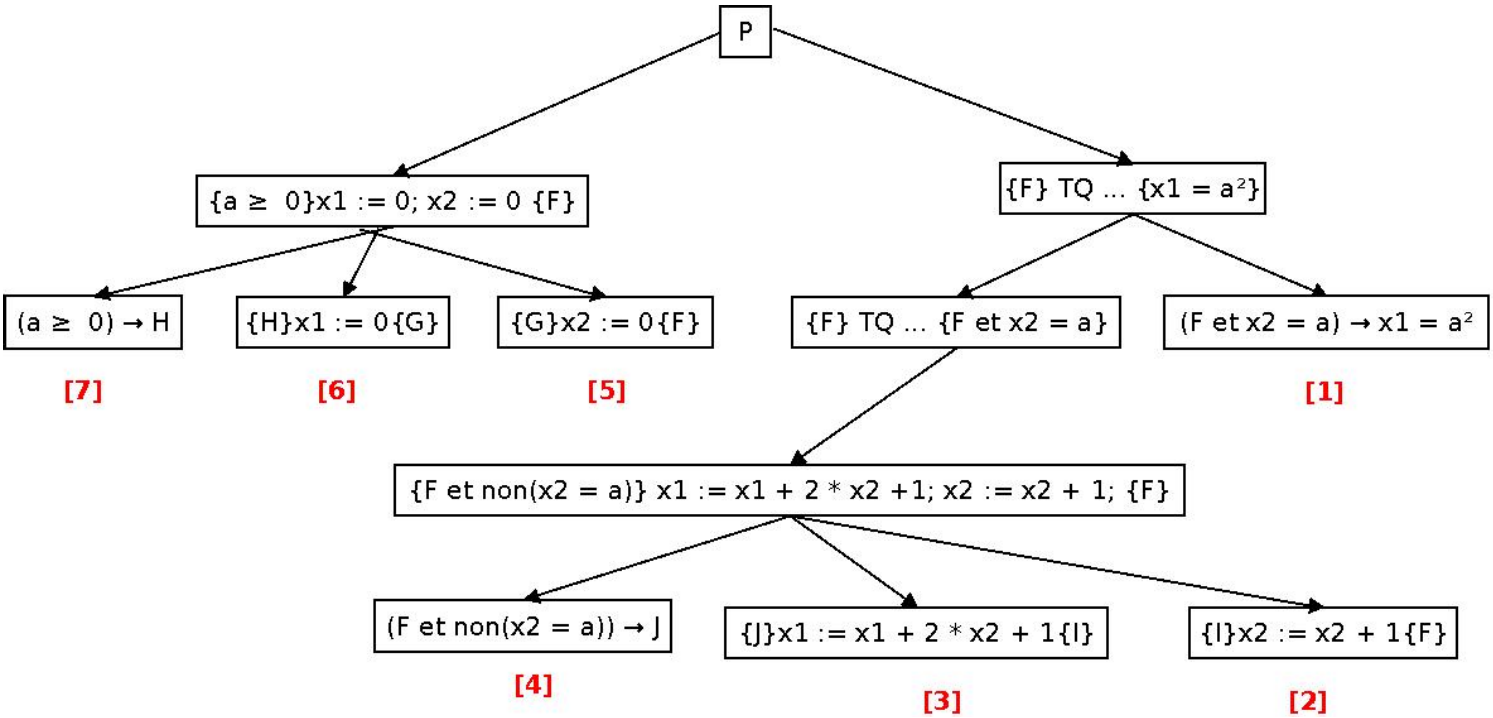
Consider the following program:

```
Read(a)

x1 := 0
x2 := 0

While (Not (x2 = a)) Do
    x1 := x1 + 2 * x2 + 1
    x2 := x2 + 1
EndWhile
```

Question: Prove that this program calculates a^2.



For this algorithm, the loop invariant can be set as :

- F = (x1 = x2²)

In fact, we can check :

- {x1 = x2²} x1 := x1 + 2 * x2 + 1; x2 := x2 + 1 {x1 = x2²}

Since the precondition is assumed to be correct, then :

- x1 := x1 + 2 * x2 + 1 ⇒
- x1 := x2² + 2 * x2 + 1 ⇒
- x1 := (x2 + 1)²

And with the second instruction :

- x2 := x2 + 1

So the post-condition x1 = $x2^2$ is correct.

By defining F, we can start to prove the different nodes in reverse post-order.

- Proof of [1]: F and (x2 = a) ⇒ (x1 = a^2) :
  - By replacing F by x1 = $x2^2$, we find (x1 = a^2).
- Proof of [2]: {I} x2 := x2 + 1 {F}
  - Using F, we find that I is: x1 = $(x2 + 1)^2$
- Proof of [3]: {J} x1 := x1 + 2 * x2 + 1 {I}
  - Using I, J is then: x1 = $x2^2$
- Proof of [4]: F & non (x2 = a) ⇒ J
  - We already have the value of F and J, verifying that (x1 = $x2^2$) and (non(x2 = a)) ⇒ (x1 = $x2^2$)
- Proof of [5]: {G} x2 := 0 {F}
  - Using F, it is possible to set G as x1 = 0
- Proof of [6]: {H} x1 := 0 {G}
  - Using the definition of assignment, it is sufficient to set 0 = 0
- Proof of [7]: (a ≥ 0) ⇒ H
  - Since H (0 = 0) is always true, then the implication is true.

4.4. Proof of functional programs

The advantage of the functional programming paradigm is that programmes are easy to prove. Functional programmes are :

- Without repetitive processing: in this case, it is a simple process or, at most, a conditional structure. There is no assignment or sequencing.
- With repetitive processing: these are recursive programmes. In this case, the dy fixed point theory is used.

We will focus on this second case, where the use of Hoare logic may not be appropriate.

4.4.1. Reminder of the fixed point theory

Inclusion of two functions

Consider two functions f and g in the set of functions from E into F (noted $\mathfrak{R}(E, F)$). We define the inclusion $f \subseteq g$ if any x in E such that $f(x)$ is defined, then $g(x)$ is also defined and $f(x) = g(x)$.

The set $\mathfrak{R}(E, F)$ with the relation $\subseteq$ The set is inductive (complete lattice) and its smallest element Ω (the function that is nowhere defined).

The fixed point theory

Consider (E <) a complete lattice and F a function from E into E.

Part 1:

Si F est croissante alors il existe une solution minimale x_0 à l'équation $F(x) = x$, c'est-à-dire, pour toute autre solution y nous avons $x_0 < y$.

If F is increasing then there exists a minimal solution x_0 to the equation $F(x) = x$, that is, for any other solution y we have $x_0 < y$.

Part 2 :

If moreover F is continuous x_0 is equal to the limit of the sequence $F^n(bi)$, bi being the lower bound of E.

4.4.2. Correspondence with recursive programs

À tout programme fonctionnel récursif f correspond une application τ de $\mathfrak{R}(E, F)$ dans $\mathfrak{R}(E, F)$. Autrement dit, un programme prend une fonction de E dans F et retourne une fonction de E dans F. De plus, l'application τ est continue, ainsi, le programme est vu comme une équation à point fixe : $f = \tau(f)$ dont le plus petit point fixe coïncide avec la fonction calculée par le programme.

To any recursive functional program f corresponds an application τ of $\mathfrak{R}(E, F)$ in $\mathfrak{R}(E, F)$. In other words, a program takes a function from E into F and returns a function from E into F. Furthermore, the application τ is continuous, so the program is seen as a fixed-point equation: $f = \tau(f)$ whose smallest fixed point coincides with the function calculated by the program.

4.4.3. Example

The function x! can be seen as the smallest solution of the fixed-point equation :

- $f = \lambda x.\quad \text{if}\quad x = 0\quad \text{then}\quad 1\quad \text{else}\quad x \cdot f(x-1)\quad \text{endif}$

If we put:

- $\tau(f) = \lambda x.\quad \text{if}\quad x = 0\quad \text{then}\quad 1\quad \text{else}\quad x \cdot f(x-1)\quad \text{endif}$

It has the form:

- $\tau(f) = f$

4.4.4. The proof of a recursive program

Thus, based on this theory, we can say:

- To prove that a recursive functional program τ calculates a given function f, it suffices to show that f is the smallest fixed of τ.
- If f is not the smallest fixed point of τ, but simply a fixed point, then we can conslure τ partially computes the function f.

If we do not know f can be found by calculating the smallest fixed point of τ which is equal to the limit of $\tau^n(\Omega)$ when n tends to infinity.

4.4.5. Examples

4.4.6. Example 01

Consider the following functional program :

- $f = \lambda xy.\quad \text{If}\quad x = y\quad \text{Then}\quad y + 1\quad \text{Else}\quad f(x, f(x-1, y+1))\quad \text{EndIf}$

Show that this program calculates :

- $g = \lambda xy.\ x + 1$

To do this, it is sufficient to show that g is a solution of the equation $\tau(g) = g$. That is:

- $\tau(g) = \lambda xy.\quad \text{If}\quad x = y\quad \text{Then}\quad y + 1\quad \text{Else}\quad g(x, g(x-1, y+1))\quad \text{EndIf}$

We have two possibilities depending on the condition in the If...Else...EndIf structure:

- In the case of x = y, the result is y + 1 or x + 1, which is equivalent to the g function.
- In the case of $x \neq y$, the internal call $g(x-1, y+1)$ gives x and the call $g(x, x)$ also gives x + 1, which is also g.

So we can say that the function f calculates (or at least partially calculates) the function g.

4.4.7. Example 02

Consider the following program:

- $f = \tau(f) = \lambda x.\quad \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad x \cdot f(x-1)\quad \text{EndIf}$

To determine the smallest fixed point, simply calculate $\tau(\Omega)$, $\tau^2(\Omega)$, $\tau^3(\Omega)$, $\tau^4(\Omega)$, ...

- Evaluation of $\tau(\Omega)$
  - $\tau(\Omega) = \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad x \cdot \Omega(x-1)\quad \text{EndIf}$
  - $= \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad \Omega\quad \text{EndIf}$
- Evaluation of $\tau^2(\Omega) = \tau(\tau(\Omega))$
  - $\tau^2(\Omega) = \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad (x) \cdot \tau(\Omega)(x-1)\quad \text{EndIf}$
  - $= \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad x \cdot (\text{Si}\quad x - 1 = 0\quad \text{Then}\quad 1\quad \text{Else}\quad (x-1) \cdot \Omega(x-1)\quad \text{EndIf}$
  - $= \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad x \cdot (\text{Si}\quad x = 1\quad \text{Then}\quad 1\quad \text{Else}\quad \Omega\quad \text{EndIf})\quad \text{EndIf}$
  - $= \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad (\text{Si}\quad x = 1\quad \text{Then}\quad 1\quad \text{Else}\quad \Omega\quad \text{EndIf})\quad \text{EndIf}$
- Evaluation of $\tau^3(\Omega) = \tau\left(\tau^2(\Omega)\right)$
  - $\tau^3(\Omega) = \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad x \cdot \tau^2(\Omega)(x-1)\quad \text{EndIf}$
  - $\tau^3(\Omega) = \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad x \cdot (\text{Si}\quad x - 1 = 0\quad \text{Then}\quad 1\quad \text{Else}\quad x \cdot (\text{Si}\quad x - 1 = 1\quad \text{Then}\quad 1\quad \text{Else}\quad \Omega\quad \text{EndIf})\quad \text{EndIf})\quad \text{EndIf}$
  - $\tau^3(\Omega) = \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad x \cdot (\text{Si}\quad x = 1\quad \text{Then}\quad 1\quad \text{Else}\quad (\text{Si}\quad x = 2\quad \text{Then}\quad 1\quad \text{Else}\quad \Omega\quad \text{EndIf})\quad \text{EndIf})\quad \text{EndIf}$
  - $\tau^3(\Omega) = \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad (\text{Si}\quad x = 1\quad \text{Then}\quad 1\quad \text{Else}\quad (\text{Si}\quad x = 2\quad \text{Then}\quad 2 \cdot 1\quad \text{Else}\quad \Omega\quad \text{EndIf})\quad \text{EndIf})\quad \text{EndIf}$
- Evaluation of $\tau^4(\Omega) = \tau\left(\tau^3(\Omega)\right)$
  - $\tau^3(\Omega) = \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad x \cdot \tau^3(\Omega)(x-1)\quad \text{EndIf}$
  - $\tau^3(\Omega) = \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad x \cdot (\text{Si}\quad x - 1 = 0\quad \text{Then}\quad 1\quad \text{Else}\quad (\text{Si}\quad x - 1 = 1\quad \text{Then}\quad 1\quad \text{Else}\quad (\text{Si}\quad x - 1 = 2\quad \text{Then}\quad 2 \cdot 1\quad \text{Else}\quad \Omega\quad \text{EndIf})\quad \text{EndIf})\quad \text{EndIf})\quad \text{EndIf}$
  - $\tau^3(\Omega) = \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad x \cdot (\text{Si}\quad x = 1\quad \text{Then}\quad 1\quad \text{Else}\quad (\text{Si}\quad x = 2\quad \text{Then}\quad 1\quad \text{Else}\quad (\text{Si}\quad x = 3\quad \text{Then}\quad 2 \cdot 1\quad \text{Else}\quad \Omega\quad \text{EndIf})\quad \text{EndIf})\quad \text{EndIf})\quad \text{EndIf}$
  - $\tau^3(\Omega) = \text{If}\quad x = 0\quad \text{Then}\quad 1\quad \text{Else}\quad (\text{Si}\quad x = 1\quad \text{Then}\quad 1\quad \text{Else}\quad (\text{Si}\quad x = 2\quad \text{Then}\quad 2 \cdot 1\quad \text{Else}\quad (\text{Si}\quad x = 3\quad \text{Then}\quad 3 \cdot 2 \cdot 1\quad \text{Else}\quad \Omega\quad \text{EndIf})\quad \text{EndIf})\quad \text{EndIf})\quad \text{EndIf}$

We observe that $\tau^n(\Omega)$ tends to calculate the product:

- $\tau^n(\Omega) = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n = n!$

5. Conclusion

In this last chapter, we presented L1, an experimental language which defines a minimum of structure but which makes it possible to write complete programs. The aim is to have a reduced instruction set so that the different semantics can be defined and practised.

We have also seen Operational Semantics. These semantics are intended for compiler designers, as they enable the behaviour of the various language structures to be defined. We also saw the notion of 'conceptual choice', which refers to the different choices made by the compiler designer, and how to express and change this behaviour using operational semantics.

We were also able to define two types of proof:

- Using Hoare logic, hence the name 'axiomatic' for this semantics.

- By using the characteristics of the λ-Calculus and the fixed point theorem to prove programs written in a functional language.

Version 0.1
Last updated 2024-11-21 00:09:08 +0100