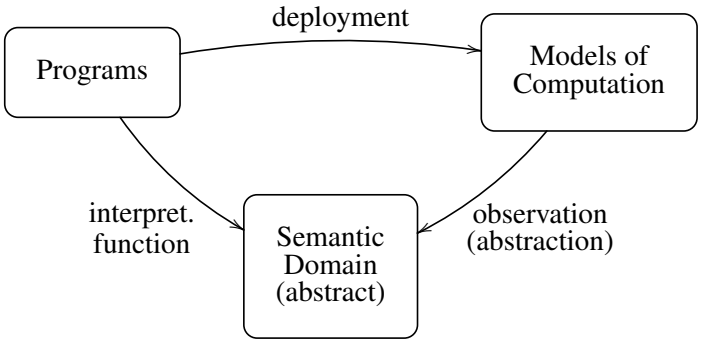
Contents

Introduction

1. Objectives

The objective of the course is to present different models of computation, their programming paradigms, their mathematical descriptions, both concrete and abstract, and also to present some important techniques for reasoning on them and for proving their properties.

To this aim, it is of fundamental importance to have a rigorous view of both syntax and semantics (meanings) of the models we work on. We call *interpretation function* the mapping from program syntax to program semantics and we will address questions that arise naturally, like establishing if two programs are *equivalent* i.e. if they have the same semantics or not.



We focus on two different approaches for assigning meanings to programs:

- *operational semantics*;
- *denotational semantics*.

The *operational semantics* fixes an operational model for the execution of a program (in a given environment). Of course, we could take a Java machine or alike but we prefer to focus on more essential models. We will define the execution as a proof in some logical system and once we are at this formal level, it will be easier to prove properties of the program.

The *denotational semantics* describes an explicit *interpretation function* over a mathematical domain. Like a numerical expression can be evaluated to return a value, the interpretation function for a typical imperative language is a mapping that, given a program, returns a function from its initial states to its final states.

We will study the correspondence between operational semantics and denotational semantics.

If we want to prove nontrivial properties of a program or of a class of programs, we usually have to use *induction* mechanisms which allow us to prove properties of elements of an infinite set (like the steps of a run, or the programs in a class). The most general notion of induction is the so called *well-founded induction* (or *Noether* induction) and we will derive from it all the other inductions.

Defining a program by *structural recursion* means to specify its meaning in terms of the meanings of its components. We will see that induction and recursion are very similar: for both induction and recursion we will need well-founded models.

If we take a program which is cyclic or recursive, then we have to express these notions at the level of the meanings, which presents some technical difficulties. A recursive program p contains a call to itself:

$$p = f(p). \tag{1}$$

We are looking for the meanings which satisfy this equation. In order to solve this problem we resort to the *fixpoint* theory of *complete partial orders with bottom* and of *continuous* functions.

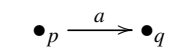
We will use these two paradigms for:

- an imperative language called **IMP**,
- and a functional language called **HOFL**.

For both of them we will define what are the programs and in the case of HOFL we will also define what are the *types*. Then we will define their operational semantics, their denotational semantics and finally, to some extent, we will prove the equivalence between the two. The fixpoint theory for HOFL will be more difficult because we are working on a more general situation where functions are first class citizens.

The models we use for IMP and HOFL are not appropriate for concurrent and interactive systems, like the very common network based applications: on the one hand we want their behavior not to depend as much as possible on the speed of processes, on the other hand we want to permit infinite computations. So we do not consider time explicitly, but we have to introduce nondeterminism to account for races between concurrent processes.

The typical models for nondeterminism and infinite computations are *transition systems*.



In the figure above, we have a transition system with two states p and q and a transition from p to q. However, from the outside we can just observe the action a associated to the transition. Equivalent processes are represented by states which have correspondent observable transitions. The language that we will employ in this setting is called **CCS** (*Calculus for Communicating Systems*), and its most used notion of observational equivalence is *bisimulation*.

Then we will study systems where we will be able to change the link structure during execution. These systems are called *open-ended*. As our case study, we will present the **π-calculus**, which extends CCS. The **π-calculus** is quite expressive, due to its ability to create and to transmit new names, which can represent ports, links, and also session names, passwords and so on in security applications.

Finally, in the last part of the course we will introduce probabilistic models, where we trade nondeterminism for probability distributions, which we associate to choice points. We will also present stochastic models, where actions take place in a continuous time setting, with an exponential distribution. Probabilistic/stochastic models find applications in many fields, e.g. performance evaluation, decision support systems and system biology.

2. Structure

The course comprises four main parts

- Computational models for imperative languages, exemplified over IMP
- Computational models for functional languages, exemplified over HOFL
- Computational models for concurrent / non-deterministic / interactive languages, exemplified over CCS and pi-calculus
- Computational models for probabilistic and stochastic process calculi

The first two models exemplify deterministic systems; the other two models non-deterministic ones. The difference will emerge clear during the course.

3. References

- Glynn Winskel, *The formal Semantics of Programming Languages*, MIT Press, 1993. Chapters: 1.3, 2, 3, 4, 5, 8, 11. (*La Semantica Formale dei Linguaggi di Programmazione*, traduzione italiana a cura di Franco Turini, UTET 1999).
- Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989. Chapters: 1-7, 10.

1. Preliminaries

1.1. Inference Rules

Inference rules are a key tool for defining syntax (e.g., which programs respect the syntax, which programs are well-typed) and semantics (e.g. to derive operational semantics by induction on the structure of the programs).

Definition 1.1 (Inference rule)
Let $x_1, x_2, \ldots, x_n, y$ be (well-formed) formulas. An inference rule *is written as*

$$r = \{ \underbrace{x_1, x_2, \ldots, x_n}_{\textit{premises}} \} / \underbrace{y}_{\textit{conclusion}}$$

using on-line notation. Letting $X = \{ x_1, x_2, \ldots, x_n \}$, equivalent notations are

$$r = \frac{X}{y} \qquad\qquad r = \frac{x_1 \quad \ldots \quad x_n}{y}$$

The idea is that if we can prove all the formulas $x_1, x_2, \ldots, x_n$ in our logic system, then by exploiting the inference rule r we can also derive the validity of the formula y.

Definition 1.2 (Axiom)
An axiom *is an inference rule with empty premise:*

$$r = \varnothing / y.$$

Equivalent notations are:

$$r = \frac{\varnothing}{y} \qquad\qquad r = \frac{}{y}$$

In other words, there are no preconditions for applying axiom r, hence there is nothing to prove in order to apply the rule: in this case we can assume y to hold.

Definition 1.3 (Logic system)
A logic system *is a set of inference rules $R = \{ r_i \mid i \in I \}$.*

In other words, having a set of rules available, we can start by deriving obvious facts using axioms and then derive new valid formulas applying the inference rules to the formulas that we know to hold (as premises). In turn, the new derived formulas can be used to prove other formulas.

Example 1.4 (Some inference rules)
The inference rule

$$\frac{a \in I \quad b \in I \quad a \oplus b = c}{c \in I}$$

means that, if *a and b are two elements that belongs to the set I* and *the result of applying the operator $\oplus$ to a and b gives c as a result,* then *c must also belong to the set I.*

The rule

$$\frac{}{5 \in I}$$

is an axiom, so we know that 5 belongs to the set I.

By composing inference rules, we build *derivations*, a fundamental concept for this course.

Definition 1.5 (Derivation)
Given a logic R, a derivation is written

$$d \Vdash_R y$$

where

- *either* $d = \varnothing / y$ *is an axiom of R, i.e.,* $(\varnothing / y) \in R$*;*
- *or* $d = (\{d_1, \ldots, d_n\} / y)$ *with* $d_1 \Vdash_R x_1, \ldots, d_n \Vdash_R x_n$ *and* $(\{x_1, \ldots, x_n\} / y) \in R$

The notion of derivation is obtained putting together different steps of reasoning according to the rules in R. We can see $d \Vdash_R y$ as the proof that in the formal system R we can derive y.

Let us look more closely at the two cases in Definition 1.5. The first case tells us that if we know that:

$$\left(\frac{\varnothing}{y}\right) \in R$$

i.e., we have an axiom for deriving y in our inference system R, then

$$\left(\frac{\varnothing}{y}\right) \Vdash_R y$$

is a derivation of y in R.

The second case tells us that if we have already proved x_1 with derivation d_1, x_2 with derivation d_2 and so on, i.e.

$$d_1 \Vdash_R x_1, \qquad d_2 \Vdash_R x_2, \qquad ..., \qquad d_n \Vdash_R x_n$$

and we have a rule for deriving y from $x_1, ..., x_n$ in our inference system, i.e.

$$\left(\frac{x_1, ..., x_n}{y}\right) \in R$$

then we can build a derivation for y:

$$\left(\frac{\{d_1, ..., d_n\}}{y}\right) \Vdash_R y$$

Summarizing all the above:

- $(\varnothing / y) \Vdash_R y$ if $(\varnothing / y) \in R$ (axiom)
- $(\{d_1, \ldots, d_n\} / y) \Vdash_R y$ if $(\{x_1, \ldots, x_n\} / y) \in R$ and $d_1 \Vdash_R x_1, \ldots, d_n \Vdash_R x_n$ (inference)

A derivation can roughly be seen as a tree whose root is the formula y we derive and whose leaves are the axioms we need.

Definition 1.6 (Theorem)
A theorem *(in R) is a well-formed formula y for which there exists a proof, and we write* $\Vdash_R y$*.*

In other words, y is a theorem if $\exists d. d \Vdash_R y$.

Definition 1.7 (Set of theorems in *R*)
We let $I_R = \{ y \mid \Vdash_R y \}$ *be the set of all theorems that can be proved in R.*

We mention two main approaches to prove theorems:

- *top-down* or *direct*
- *bottom-up* or *goal-oriented*

The *top-down* is the approach we used before in which we start from the axioms and we prove the theorems. However, quite often, we would work using *bottom-up* because we have already a goal and we want to prove that this is a theorem.

Example 1.8 (Grammars as sets of inference rules)
Every grammar can be presented equivalently as a set of inference rules. Let us consider the well-known grammar for strings of balanced parentheses. Recalling that λ denotes the empty string, we write:

```
S ::= SS | (S) | λ
```

We let L_S denote the set of strings generated by the grammar for the symbol S. The translation from production to inference rules is straightforward. The first production `S ::= SS` *says that given any two strings s_1 and s_2 of balanced parentheses, their juxtaposition is also a string of balanced parentheses. In other words:*

$$\frac{s_1 \in L_S \quad s_2 \in L_S}{s_1 s_2 \in L_S}\ (1)$$

Similarly, the second production `S ::= (S)` *says that we can surround with brackets any string s of balanced parentheses and get again a string of balanced parentheses. In other words:*

$$\frac{s \in L_S}{(s) \in L_S}\ (2)$$

Finally, the last rule says that the empty string λ is just a particular string of balanced parentheses. In other words we have an axiom:

$$\frac{}{\lambda \in L_S}\ (3)$$

Note the difference between the placeholders s, s_1, s_2 and the symbol λ appearing in the rules above: the former can be replaced by any string to obtain a specific instance of rules (1) and (2), while the latter denotes a given string (i.e. rules (1) and (2) define rule schemes with many instances, while there is a unique instance of rule (3)).
For example, the rule

$$\frac{)(\in L_S \quad ((\in L_S}{)(((\in L_S}\ (1)$$

is an instance of rule (1): it is obtained by replacing s_1 with `)(` *and s_2 with* `((`*. Of course the string* `)(((` *appearing in the conclusion does not belong to L_S, but the instance is perfectly valid, because it says that "*`)(((` $\in L_S$ if `)(` $\in L_S$ *and* `((` $\in L_S$*" and since the premises are false we cannot draw any conclusion.*
Let us see an example of valid derivation that uses some valid instances of rules (1) and (2).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\lambda \in L_S}\ (3)}{(\lambda) = () \in L_S}\ (2)}{(()) \in L_S}\ (2) \qquad \dfrac{\dfrac{}{\lambda \in L_S}\ (3)}{(\lambda) = () \in L_S}\ (2)}{(())() \in L_S}\ (1)$$

Reading the proof (from the leaves to the root): Since $\lambda \in L_S$ by axiom (3), then we know that `()` $\in L_S$ *by (2); if we apply again rule (2) we derive also* `(())` $\in L_S$ *and hence* `(())()` $\in L_S$ *by (1). In other words* `(())()` $\in L_S$ *is a theorem.*
Let us introduce a second formalization of the same language (balanced parentheses) without using inference rules. In the following we let a_i denote the ith symbol of the string a. Let

$$f(a_i) = \begin{cases} 1 & \text{if } a_i = (\\ -1 & \text{if } a_i =) \end{cases}$$

A string of n parentheses $a = a_1 a_2 ... a_n$ is balanced iff both the following properties hold:

Property 1 $\sum_{i=1}^{m} f(a_i) \geq 0 \qquad m = 0, 1...n$

Property 2 $\sum_{i=1}^{n} f(a_i) = 0$

Intuitively, $\sum_{i=1}^{m} f(a_i)$ counts the difference between the number of open parentheses and closed parentheses. Therefore, the first property requires that in any prefix of the string a the number of open parentheses exceeds the number of closed ones; the second property requires that the string a has as many open parentheses than closed ones.

An example is shown below for a = (())():

$m =$	1	2	3	4	5	6
$a_m =$	(	(	)	)	(	)
$f(a_m) =$	1	1	−1	−1	1	−1
$\sum_{i=1}^{m} f(a_i) =$	1	2	1	0	1	0

The two properties are easy to inspect for any string and therefore define an useful procedure to check *if a string belongs to our language or not.*

Next, we show that the two different characterizations of the language (by inference rules and by inspection) of balanced parentheses are equivalent.

Theorem 1.9

$$a \in L_S \iff \begin{cases} \sum_{i=1}^{m} f(a_i) \geq 0 & m = 0, 1...n \\ \sum_{i=1}^{n} f(a_i) = 0 \end{cases}$$

The proof is composed of two implications that we show separately:

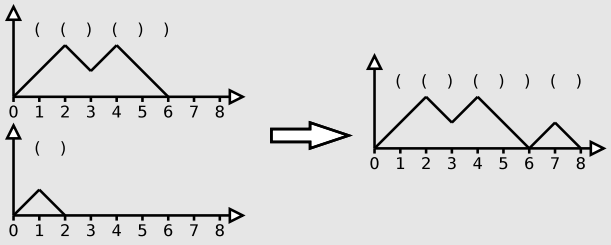
$\Longrightarrow$ *all the strings produced by the grammar satisfy the two properties;*

$\Longleftarrow$ *any string that satisfy the two properties can be generated by the grammar.*

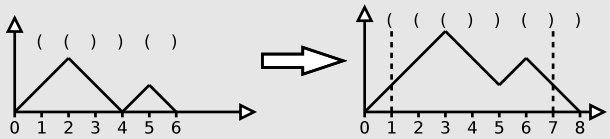
Proof of $\Longrightarrow$***)*** *To show the first implication, we proceed by* induction over the rules*: we assume that the implication is valid for the premises and we show that it holds for the conclusion:*

The two properties can be represented graphically over the cartesian plane by taking m over the x-axis and $\sum_{i=1}^{m} f(a_i)$ over the y-axis. Intuitively, the graph should start in the origin; it should never cross below the x-axis and it should end in $(n, 0)$.

Let us check that by applying any inference rule the properties 1 and 2 still hold. The first inference rule corresponds to the juxtaposition of the two graphs and therefore the result still satisfies both properties (when the original graphs do).



The second rule corresponds to translate the graph upward (by 1 unit) and therefore the result still satisfies both properties (when the original graph does).

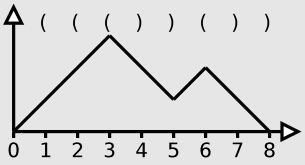


The third rule is just concerned with the empty string that trivially satisfies the two properties.

Since we have inspected all the inference rules, the proof of the first implication is concluded. □

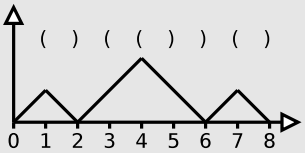
Proof of ⟸) *We need to find a derivation for any string that satisfies the two properties. Let a be such a generic string. (We only sketch this direction of the proof, that goes by induction over the length of the string a.) We proceed by case analysis, considering three cases:*

1. *If $n = 0$, $a = \lambda$. Then, by rule (3) we conclude that $a \in L_S$.*
2. *The second case is when the graph associated with a never touches the x-axis (except for its start and end points). An example is shown below:*



In this case we can apply rule (2), because we know that the parentheses opened at the beginning of a is only matched by the parenthesis at the very end of a.

3. *The third and last case is when the graph touches the x-axis (at least) once in a point $(k, 0)$ different from its start and its end. An example is shown below:*



In this case the substrings $a_1...a_k$ and $a_{k+1}...a_n$ are also balanced and we can apply the rule (1) to their derivations to prove that $a \in L_S$. □

*The last part of the proof outlines a goal-oriented strategy to build a derivation for a given string: We start by looking for a rule whose conclusion can match the goal we are after. If there are no alternatives, then we fail. If we have only one alternative we need to build a derivation for its premises. If there are more than one alternatives we can either explore all of them in parallel (*breadth-first *approach) or try one of them and back-track in case we fail (*depth-first*).*

Suppose we want to find a proof for $(())() \in L_S$. We use the notation $(())() \in L_S \nwarrow$ to mean that we look for a goal-oriented derivation.

- *Rule (1) can be applied in many different ways, by splitting the string $(())()$ in all possible ways. We use the notation $(())() \in L_S \nwarrow \lambda \in L_S, (())() \in L_S$ to mean that we reduce the proof of $(())() \in L_S$ to that of $\lambda \in L_S$ and $(())() \in L_S$. Then we have all the following alternatives to inspect:*
  1. $(())() \in L_S \nwarrow \lambda \in L_S, (())() \in L_S$
  2. $(())() \in L_S \nwarrow (\in L_S,))() \in L_S$
  3. $(())() \in L_S \nwarrow ((\in L_S,))() \in L_S$
  4. $(())() \in L_S \nwarrow (() \in L_S,)() \in L_S$
  5. $(())() \in L_S \nwarrow (()) \in L_S, () \in L_S$

6. $(())() \in L_S \nwarrow (())(\in L_S,) \in L_S$
7. $(())() \in L_S \nwarrow (())() \in L_S, \lambda \in L_S$

Note that some alternatives are identical except for the order in which they list subgoals (1 and 7) and may require to prove the same goal from which we started (1 and 7). For example, if option 1 is selected applying depth-first strategy without any additional check, the derivation procedure might diverge. Moreover, some alternatives lead to goals we won't be able to prove (2, 3, 4, 6).

- *Rule (2) can be applied in only one way:*

$$(())() \in L_S \nwarrow ())(\in L_S$$

- *Rule (3) cannot be applied.*

We show below a successful goal-oriented derivation.

$$\begin{array}{lll}
(())() \in L_S & \nwarrow & (()) \in L_S, () \in L_S \quad \textit{by applying (1)} \\
& \nwarrow & (()) \in L_S, \lambda \in L_S \quad \textit{by applying (2) to the second goal} \\
& \nwarrow & (()) \in L_S \quad \textit{by applying (3) to the second goal} \\
& \nwarrow & () \in L_S \quad \textit{by applying (2)} \\
& \nwarrow & \lambda \in L_S \quad \textit{by applying (2)} \\
& \nwarrow & \square \quad \textit{by applying (3)}
\end{array}$$

We remark that in general the problem to check if a certain formula is a theorem is only *semidecidable* (not necessarily *decidable*). In this case the breadth-first strategy for goal-oriented derivation offers a semidecision procedure.

1.2. Logic Programming

We end this chapter by mentioning a particularly relevant paradigm based on goal-oriented derivation: *logic programming* and its Prolog incarnation. Prolog exploits depth-first goal-oriented derivations with backtracking.

Let $X = \{x, y, ...\}$ be a set of variables, $\Sigma = \{f(.,.), g(.), ...\}$ a signature of function symbols (with given arities), $\Pi = \{\mathsf{p}(.), \mathsf{q}(.,.), ...\}$ a signature of predicate symbols (with given arities). We denote by Σ_n (respectively Π_n) the subset of function symbols (respectively predicate symbols) with arity n.

Definition 1.10 (Atomic formula)
An atomic formula *consists of a predicate symbol* p *of arity n applied to n terms with variables.*

For example, $\mathsf{p}(\, f(g(x), x)\,,\, g(y)\,)$ is an atomic formula (p has arity 2, f has arity 2, g has arity 1).

Definition 1.11 (Formula)
A formula *is the conjunction of atomic formulas.*

Definition 1.12 (Horn clause)
A Horn clause *is written* l: $-r$ *where* l *is an atomic formula, called the* head *of the clause, and* r *is a formula called the* body *of the clause.*

A logic program is a set of Horn clauses. The variables appearing in each clause can be instantiated with any term. The goal itself may have variables.

Unification is used to "match" the head of a clause to the goal we want to prove in the most general way (i.e. by instantiating the variables as little as possible). Before performing unification, the variables of the clause are renamed with fresh identifiers to avoid any clash with the variables already present in the goal. Unification itself may introduce new variables to represent placeholders that can be substituted by any term with no consequences for the matching.

Example 1.13 (Sum in Prolog)
Let us consider the logic program consisting of the clauses:

$$\begin{aligned} \mathsf{sum}(\,0\,,\,y\,,\,y\,) \quad &:- \\ \mathsf{sum}(\,s(x)\,,\,y\,,\,s(z)\,) \quad &:- \quad \mathsf{sum}(\,x\,,\,y\,,\,z\,) \end{aligned}$$

where $\mathsf{sum}(.,.,.) \in \Pi_3$, $s(.) \in \Sigma_1$, $0 \in \Sigma_0$ *and* $x, y, z \in X$.

Let us consider the goal $\mathsf{sum}(\,s(s(0))\,,\,s(s(0))\,,\,v\,)$ *with* $v \in X$.

There is no match against the head of the first clause, because 0 *is different from* $s(s(0))$.

We rename x, y, z *in the second clause to* x', y', z' *and compute the unification of* $\mathsf{sum}(\,s(s(0))\,,\,s(s(0))\,,\,v\,)$ *and* $\mathsf{sum}(\,s(x')\,,\,y'\,,\,s(z')\,)$. *The result is the substitution (called most general unifier)*

$$[\,x' = s(0) \quad y' = s(s(0)) \quad v = s(z')\,]$$

We then apply the substitution to the body of the clause, which will form the new goal to prove

$$\mathsf{sum}(\,x'\,,\,y'\,,\,z'\,)[\,x' = s(0) \quad y' = s(s(0)) \quad v = s(z')\,] = \mathsf{sum}(\,s(0)\,,\,s(s(0))\,,\,z'\,)$$

We write the derivation described above using the notation

$$\mathsf{sum}(\,s(s(0))\,,\,s(s(0))\,,\,v\,) \quad \nwarrow_{v=s(z')} \quad \mathsf{sum}(\,s(0)\,,\,s(s(0))\,,\,z'\,)$$

where we have recorded the substitution applied to the variables originally present in the goal (just v in the example), to record the least condition under which the derivation is possible.

The derivation can then be completed as follows:

$$\begin{aligned} \mathsf{sum}(\,s(s(0))\,,\,s(s(0))\,,\,v\,) \quad &\nwarrow_{v=s(z')} \quad \mathsf{sum}(\,s(0)\,,\,s(s(0))\,,\,z'\,) \\ &\nwarrow_{z'=s(z'')} \quad \mathsf{sum}(\,0\,,\,s(s(0))\,,\,z''\,) \\ &\nwarrow_{z''=s(s(0))} \quad \square \end{aligned}$$

By composing the computed substitutions we get

$$\begin{aligned} z' &= s(z'') = s(s(s(0))) \\ v &= s(z') = s(s(s(s(0)))) \end{aligned}$$

This gives us a proof of the theorem

$$\mathsf{sum}(\,s(s(0))\,,\,s(s(0))\,,\,s(s(s(s(0))))\,)$$

Part I.

IMP language

2. Operational Semantics of IMP

2.1. Syntax of IMP

The IMP programming language is a simple imperative language (a bare bone version of the C language) with three data types:

int: the set of integer numbers, ranged over by metavariables $m, n, m', n', m_0, n_0, m_1, n_1, ...$

$$\mathbb{N} = \{0, \pm 1, \pm 2, ...\}$$

bool: the set of boolean values, ranged over by metavariables $v, v', v_0, v_1, ...$

$$\mathbf{T} = \{ \textbf{true}, \textbf{false} \}$$

locations: the (denumerable) set of memory locations (we always assume there are enough locations available, i.e. our programs won't run out of memory), ranged over by metavariables $x, y, x', y', x_0, y_0, x_1, y_1, ...$

$$\mathbf{Loc} \quad \textit{locations}$$

Definition 2.1 (IMP: syntax)
The grammar for IMP comprises:

Aexp: *Arithmetic expressions, ranged over by* $a, a', a_0, a_1, ...$

Bexp: *Boolean expressions, ranged over by* $b, b', b_0, b_1, ...$

Com: *Commands, ranged over by* $c, c', c_0, c_1, ...$

The following productions define the syntax of IMP:

$$\begin{array}{rcl} a \in Aexp & ::= & n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \\ b \in Bexp & ::= & v \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \vee b_1 \mid b_0 \wedge b_1 \\ c \in Com & ::= & \textbf{skip} \mid x := a \mid c_0; c_1 \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b \textbf{ do } c \end{array}$$

where we recall that n is an integer number, v a boolean value and x a location,

IMP is a very simple imperative language and there are several constructs we deliberately omit. For example we omit other common conditional statements, like *switch*, and other cyclic constructs like *repeat*. Moreover IMP commands imposes a structured flow of control, i.e., IMP has no labels, no goto statements, no break statements, no continue statements. Other things which are missing and are difficult to model are those concerned with *modular programming*. In particular, we have no *procedures*, no *modules*, no *classes*, no *types*. Since IMP does not include variable declarations, procedures and blocks, memory allocation is essentially static. Of course, we have no *concurrent programming* construct.

2.1.1. Arithmetic Expressions

An arithmetic expression can be an integer number, or a location, a sum, a difference or a product. We notice that we do not have division because it can be undefined or give different values and other things we are not interested in.

2.1.2. Boolean Expressions

A boolean expression can be a logical value v; the equality of an arithmetic expression with another; an arithmetic expression less or equal than another one; the negation; the logical product or the logical sum.

To keep the notation compact, in the following we will take the liberty of writing boolean expressions such as $x \neq 0$, in place of $\neg(x = 0)$ and $x > 0$ in place of $1 \leq x$ or $(0 \leq x) \wedge \neg(x = 0)$.

2.1.3. Commands

A command can be **skip**, i.e. a command which is not doing anything, or an assignment where we have that an arithmetic expression is evaluated and the value is assigned to a location; we can also have the sequential execution of two commands (one after the other); an **if-then-else** with the obvious meaning: we evaluate a boolean b, if it is true we execute c_0 and if it is false we execute c_1. Finally we have a **while** which is a command that keeps executing c until b becomes false.

2.1.4. Abstract Syntax

The notation above gives the so-called *abstract syntax* in that it simply says how to build up new expressions and commands but it is ambiguous for parsing a string. It is the job of the *concrete syntax* to provide enough information through parentheses or orders of precedence between operation symbols for a string to parse uniquely. It is helpful to think of a term in the abstract syntax as a specific parse tree of the language.

Example 2.2 (Valid expressions)

(**while**	b	**do**	c_1)	;	c_2	*is a valid command*
while	b	**do**	(c_1	;	c_2)	*is a valid command*
while	b	**do**	c_1	;	c_2	*is not a valid command, because it is ambiguous*

In the following we will assume that enough parentheses have been added to resolve any ambiguity in the syntax. Then, given any formula of the form $a \in Aexp$, $b \in Bexp$, or $c \in Com$, the process to check if such formula is a "theorem" is deterministic (no backtracking is needed).

Example 2.3 (Validity check)
Let us consider the formula **if** $(x = 0)$ **then** (**skip**) **else** $(x := (x - 1)) \in Com$. *We can prove its validity by the following (deterministic) derivation*

$$
\begin{aligned}
&\textbf{if}\,(x = 0)\,\textbf{then}\,(\textbf{skip})\,\textbf{else}\,(x := (x - 1)) \in Com \\
&\quad\nwarrow\ x = 0 \in Bexp, \textbf{skip} \in Com, x := (x - 1) \in Com \\
&\quad\nwarrow\ x \in Aexp, 0 \in Aexp, \textbf{skip} \in Com, x := (x - 1) \in Com \\
&\quad\nwarrow\ x - 1 \in Aexp \\
&\quad\nwarrow\ x \in Aexp, 1 \in Aexp \\
&\quad\nwarrow\ \square
\end{aligned}
$$

2.2. Operational Semantics of IMP

2.2.1. Memory State

In order to define the evaluation of an expression or the execution of a command, we need to handle the state of the machine which is going to execute the IMP statements. Beside expressions to be evaluated and commands to be executed, we also need to record in the state some additional elements like values and memories. To this aim, we introduce the notion of *memory*:

$$\sigma : \Sigma = (Loc \to \mathbb{N})$$

The memory σ is an element of the set Σ which contains all the functions from locations to integer numbers. A particular σ is a particular function from locations to integer numbers so a *memory* is a function which associates to each location x the value $\sigma(x)$ that x stores.

Since **Loc** is an infinite set, things can be complicated: handling functions from an infinite set is not a good idea for a model of computation. Although **Loc** is large enough to store all the values that are manipulated by expressions and commands, the functions we are interested in are functions which are almost everywhere 0, except for a finite subset of memory locations.

If, for instance, we want to represent a memory we could write:

$$\sigma = (5/x,\ 10/y)$$

meaning that the location x contains the value 5 and the location y the value 10 and elsewhere 0. In this way we can represent any memory by a finite set of pairs.

Definition 2.4 (Zero memory)
We let $\sigma_0 = ()$ denote the memory such that $\forall x.\sigma_0(x) = 0$.

Definition 2.5 (Assignment)
Given a memory σ, we denote by $\sigma[^n/_x]$ the memory where the value of x is updated to n, i.e. such that

$$\sigma[^n/_x](y) = \begin{cases} n & \text{if } y = x \\ \sigma(y) & \text{if } y \neq x \end{cases}$$

Note that $\sigma[^n/_x][^m/_x](y) = \sigma[^m/_x](y)$. In fact:

$$\sigma[^n/_x][^m/_x](y) = \begin{cases} m & \text{if } y = x \\ \sigma[^n/_x](y) = \sigma(y) & \text{if } y \neq x \end{cases}$$

2.2.2. Inference Rules

Now we are going to give the *operational semantics* to IMP using an inference system like the one we saw before. It is called "big-step" semantics because it leads in one proof to the result.

We are interested in three kinds of *well formed formulas*:

Arithmetic expressions: The evaluation of an element of Aexp in a given σ results in an integer number.

$$\langle a, \sigma \rangle \to n$$

Boolean expressions: The evaluation of an element of Bexp in a given σ results in either **true** or **false**.

$$\langle b, \sigma \rangle \to v$$

Commands: The evaluation of an element of Com in a given σ leads to an updated final state σ'.

$$\langle c, \sigma \rangle \to \sigma'$$

Next we show each inference rule and comment on it. We start with the rules about arithmetic expressions.

$$\frac{}{\langle n, \sigma \rangle \to n}\ (\text{num}) \tag{2.1}$$

The axiom 2.1 is trivial: the evaluation of any numerical constant n (seen as syntax) results in the corresponding integer value n (read as an element of the semantic domain).

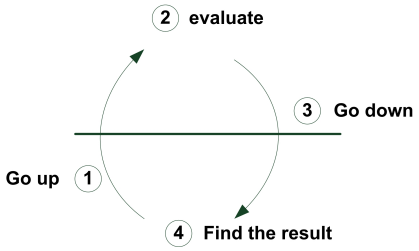
$$\frac{}{\langle x, \sigma\rangle \to \sigma(x)}\ (\text{ide}) \tag{2.2}$$

The axiom 2.2 is also quite intuitive: the evaluation of an identifier x in the memory σ results in the value stored in x.

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 + a_1, \sigma\rangle \to n}\ n = n_0 + n_1\ (\text{sum}) \qquad \frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 + a_1, \sigma\rangle \to n_0 + n_1}\ (\text{sum}) \tag{2.3}$$

The rule 2.3 for the sum has several premises: the evaluation of the syntactic expression $a_0 + a_1$ in σ returns a value n that corresponds to the arithmetic sum of the values n_0 and n_1 obtained after evaluating, respectively, a_0 and a_1 in σ. We present two equivalent versions of the rule: on the left, we exploit the side condition $n = n_0 + n_1$ to indicate the relation between the target n of the conclusion and the targets of the premises; on the right we use a more compact notation, where the target of the conclusion is obtained as the sum of the targets of the premises. In the following we shall adopt the second format. We remark the difference between the two occurrences of the symbol + in the rule: in the source of the conclusion it denotes a piece of syntax, in the target of the conclusion it denotes a semantic operation. To avoid any ambiguity we could have introduced different symbols in the two cases, but we have preferred to overload the symbol and keep the notation simpler. We hope the reader is expert enough to assign the right meaning to each occurrence of overloaded symbols by looking at the context in which they appear.

The way we read this rule is very interesting because, in general, if we want to evaluate the lower part we have to go up, evaluate the uppermost part and then compute the sum and finally go down again:



In this case we suppose we want to evaluate in memory σ the arithmetic expression $a_0 + a_1$. We have to evaluate a_0 in the same memory σ and get n_0, then we have to evaluate a_1 within the same memory σ and then the final result will be $n_0 + n_1$.

This kind of mechanism is very powerful because we deal with more proofs at once. First, we evaluate a_0. Second, we evaluate a_1. Then, we put all together. If we need to evaluate several expressions on a sequential machine we have to deal with the issue of fixing the order in which to proceed. On the other hand, in this case, using a logical language we just model the fact that we want to evaluate a tree (an expression) which is a tree of proofs in a very simple way and make explicit that the order is not important.

The rules for the remaining arithmetic expressions are similar to the one for the sum. We report them for completeness, but do not comment on them.

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 - a_1, \sigma\rangle \to n_0 - n_1}\ (\text{dif}) \tag{2.4}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \times a_1, \sigma\rangle \to n_0 \times n_1}\ (\text{prod}) \tag{2.5}$$

The rules for boolean expressions are also similar to the previous ones and need no comment.

$$\frac{}{\langle v, \sigma\rangle \to v}\ (\text{bool}) \tag{2.6}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 = a_1, \sigma\rangle \to (n_0 = n_1)}\ (\text{equ}) \tag{2.7}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \leq a_1, \sigma\rangle \to (n_0 \leq n_1)}\ (\text{leq}) \tag{2.8}$$

$$\frac{\langle b, \sigma\rangle \to v}{\langle \neg b, \sigma\rangle \to \neg v}\ (\text{not}) \tag{2.9}$$

$$\frac{\langle b_0, \sigma\rangle \to v_0 \quad \langle b_1, \sigma\rangle \to v_1}{\langle b_0 \vee b_1, \sigma\rangle \to (v_0 \vee v_1)}\ (\text{or}) \tag{2.10}$$

$$\frac{\langle b_0, \sigma\rangle \to v_0 \quad \langle b_1, \sigma\rangle \to v_1}{\langle b_0 \wedge b_1, \sigma\rangle \to (v_0 \wedge v_1)}\ (\text{and}) \tag{2.11}$$

Next, we move to the inference rules for commands.

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma}\ (\text{skip}) \tag{2.12}$$

The rule 2.12 is very simple: it leaves the memory σ unchanged.

$$\frac{\langle a, \sigma\rangle \to m}{\langle x := a, \sigma\rangle \to \sigma[{}^{m}/_{x}]}\ (\text{assign}) \tag{2.13}$$

The rule 2.13 exploits the assignment operation to update σ: we remind that $\sigma[{}^{m}/_{x}]$ is the same memory as σ except for the value assigned to x (m instead of $\sigma(x)$).

$$\frac{\langle c_0, \sigma\rangle \to \sigma'' \quad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c_0; c_1, \sigma\rangle \to \sigma'}\ (\text{seq}) \tag{2.14}$$

The rule 2.14 for the sequential composition (concatenation) of commands is quite interesting. We start by evaluating the first command c_0 in the memory σ. As a result we get an updated memory σ'' which we use for evaluating the second command c_1. In fact the order of evaluation of the two command is important and it would not make sense to evaluate c_1 in the original memory σ, because the effects of executing c_0 would be lost. Finally, the memory σ' obtained by evaluating c_1 in σ'' is returned as the result of evaluating $c_0; c_1$ in σ.

The conditional statement requires two different rules, that depend on the evaluation of the condition b (they are mutually exclusive).

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c_0, \sigma\rangle \to \sigma'}{\langle \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, \sigma\rangle \to \sigma'}\ (\text{iftt}) \tag{2.15}$$

$$\frac{\langle b, \sigma\rangle \to \textbf{false} \quad \langle c_1, \sigma\rangle \to \sigma'}{\langle \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, \sigma\rangle \to \sigma'}\ (\text{ifff}) \tag{2.16}$$

The rule 2.15 checks that b evaluated to true and then returns as result the memory σ' obtained by evaluating the command c_0 in σ. On the contrary, the rule 2.16 checks that b evaluated to false and then returns as result the memory σ' obtained by evaluating the command c_1 in σ.

Also the while statement requires two different rules, that depends on the evaluation of the guard b (they are mutually exclusive).

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma''\rangle \to \sigma'}{\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'}\ (\text{whtt}) \tag{2.17}$$

$$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma}\ (\text{whff}) \tag{2.18}$$

The rule 2.17 applies to the case where the guard evaluates to true: we need to compute the memory σ'' obtained by the evaluation of the body c in σ and then to iterate the evaluation of the cycle over σ''.

The rule 2.18 applies to the case where the guard evaluates to false: then we "exit" the cycle and return the memory σ unchanged.

Remark 2.6
There is one important difference between the rule 2.17 and all the other inference rules we have encountered so far. All the other rules takes as premises formulas that are "smaller in size" than their conclusions. This fact allows to decrease the complexity of the atomic goals to be proved as the derivation proceeds further, until having basic formulas to which axioms can be applied. The rule 2.17 is different because it recursively uses as a premise a formula as complex as its conclusion. This justifies the fact that a while command can cycle indefinitely, without terminating.

The set of all inference rules above defines the operational semantics of IMP. Formally, they induce a relation that contains all the pairs input-result, where the input is the expression / command together with the initial memory and the result is the corresponding evaluation:

$$\to \subseteq (Aexp \times \Sigma \times \mathbb{N}) \cup (Bexp \times \Sigma \times \mathbf{T}) \cup (Com \times \Sigma \times \Sigma)$$

2.2.3. Examples

Example 2.7 (Semantic evaluation of a command)
Let us consider the (extra-bracketed) command

$$c = (x := 0)\ ;\ (\ \textbf{while}\ (0 \le y)\ \ \textbf{do}\ (\ (x := ((x + (2 \times y)) + 1))\ ;\ (y := (y - 1))\ )\ )$$

To improve readability and without introducing too much ambiguity, we can write it as follows:

$$c = x := 0\ ;\ \textbf{while}\ \ 0 \le y\ \ \textbf{do}\ (\ x := x + (2 \times y) + 1\ ;\ y := y - 1\ )$$

or exploiting the usual convention for indented code, as:

$$\begin{aligned} c = \quad & x := 0\ ; \\ & \textbf{while}\ \ 0 \le y\ \ \textbf{do}\ (\\ & \qquad x := x + (2 \times y) + 1\ ; \\ & \qquad y := y - 1 \\ &) \end{aligned}$$

Without too much difficulties, the experienced reader can guess the relation between the value of y at the beginning of the execution and that of x at the end of the execution: The program computes the square of (the value initially stored in) y plus 1 (when $y \ge 0$) and stores it in x. In fact, by exploiting the well-known

equalities $0^2 = 0$ *and* $(n+1)^2 = n^2 + 2n + 1$*, the value of* $(y+1)^2$ *is computed as the sum of the first* $y+1$ *odd numbers* $\sum_{i=0}^{y}(2i+1)$*. For example, for* $y = 3$ *we have* $4^2 = 1+3+5+7 = 16$*.*

We report below the proof of well-formedness of the command, as a witness that c respects the syntax of IMP. (Of course the inference rules used in the derivation are those associated to the productions of the grammar of IMP.)

$$\dfrac{\dfrac{\overline{x}\quad 0}{x := 0}\quad \dfrac{\dfrac{\overline{0}\quad \overline{y}}{0 \le y}\quad \dfrac{\dfrac{\overline{x}\quad \dfrac{\dfrac{\overline{x}\quad \dfrac{\overline{2}\quad \overline{y}}{(2\times y)}}{a_1 = ((x + 2\times y))}\quad \overline{1}}{a = ((x + (2\times y)) + 1)}}{c_3 = (x := ((x + (2\times y)) + 1))}\quad \dfrac{\overline{y}\quad \dfrac{\overline{y}\quad \overline{1}}{(y-1)}}{c_4 = (y := (y-1))}}{c_2 = ((x := ((x + (2\times y)) + 1)); (y := y - 1))}}{c_1 = (\textbf{while}(0 \le y)\,\textbf{do}((x := ((x + 2\times y)) + 1)); (y := (y-1)))}}{c = ((x := 0); (\textbf{while}(0 \le y)\,\textbf{do}((x := ((x + (2\times y)) + 1)); (y := (y+1)))))}$$

We can summarize the above proof as follows, introducing several shorthands for referring to some subterms of c that will be useful later.

$$c = \underbrace{x := 0}; \textbf{while}\, \underbrace{0 \le y}\, \textbf{do}\, (\underbrace{x := \overbrace{\underbrace{x + (2\times y)}_{a_1} + 1}^{a}}_{c_3}; \underbrace{y := y - 1}_{c_4})$$
$$\underbrace{\qquad\qquad\qquad\qquad}_{c_2}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{c_1}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{c}$$

To find the semantics of c in a given memory we proceed in the goal-oriented fashion. For instance, we take the well-formed formula $\left\langle c, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\right\rangle \to \sigma$*, and check if there exists a memory* σ *such that the formula becomes a theorem. This is equivalent to find an answer to the following question: "given the initial memory* $({}^{27}/_{x}, {}^{2}/_{y})$ *and the command c, can we find a derivation that leads to some memory* σ*?" By answering in the affirmative, we would have a proof of termination for c and would establish the content of the memory* σ *at the end of the computation.*

We show the proof in the tree-like notation: the goal to prove is the root (situated at the bottom) and the "pieces" of derivation are added on top. As the tree grows immediately larger, we split the derivation in smaller pieces that are proved separately.

$$\dfrac{\dfrac{\dfrac{}{\langle 0, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to 0}\,\text{num}}{\langle x := 0, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \left({}^{27}/_{x}, {}^{2}/_{y}\right)\left[{}^{0}/_{x}\right] = \sigma'}\,\text{assign} \qquad \langle c_1, \sigma'\rangle \to \sigma}{\langle c, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma}\,\text{seq}$$

Note that c_1 *is a cycle, therefore we have two possible rules that can be applied, depending on the evaluation of the guard. We only show the successful derivation, recalling that* $\sigma' = \left({}^{0}/_{x}, {}^{2}/_{y}\right)$*.*

$$\dfrac{\dfrac{\dfrac{}{\langle 0, \sigma'\rangle \to 0}\,\text{num} \qquad \dfrac{}{\langle y, \sigma'\rangle \to \sigma'(y) = 2}\,\text{ide}}{\langle 0 \le y, \sigma'\rangle \to (0 \le 2) = \textbf{true}}\,\text{leq} \qquad \langle c_2, \sigma'\rangle \to \sigma'' \quad \langle c_1, \sigma''\rangle \to \sigma}{\langle c_1, \sigma'\rangle \to \sigma}\,\text{whtt}$$

Next we need to prove the goals $\langle c_2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma''$ *and* $\langle c_1, \sigma''\rangle \to \sigma$. *Let us focus on* $\langle c_2, \sigma'\rangle \to \sigma''$ *first:*

$$\dfrac{\dfrac{\langle a_1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m' \qquad \dfrac{}{\langle 1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to 1}\,\text{num}}{\langle a, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m = m' + 1}\,\text{sum}}{\langle c_3, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \left({}^{0}/_{x}, {}^{2}/_{y}\right)\left[{}^{m}/_{x}\right] = \sigma'''}\,\text{assign} \qquad \dfrac{\langle y - 1, \sigma'''\rangle \to m''}{\langle c_4, \sigma'''\rangle \to \sigma'''\left[{}^{m''}/_{y}\right] = \sigma''}\,\text{assign}}{\langle c_2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma''}\,\text{seq}$$

We show separately the derivations for $\langle a_1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m'$ *and* $\langle y - 1, \sigma'''\rangle \to m''$ *in full details:*

$$\dfrac{\dfrac{}{\langle x, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to 0}\,\text{ide} \qquad \dfrac{\dfrac{}{\langle 2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to 2}\,\text{num} \qquad \dfrac{}{\langle y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to 2}\,\text{ide}}{\langle 2 \times y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m''' = 2 \times 2 = 4}\,\text{prod}}{\langle a_1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m' = 0 + 4 = 4}\,\text{sum}$$

Since $m' = 4$, *then it means that* $m = m' + 1 = 5$ *and hence* $\sigma''' = \left({}^{0}/_{x}, {}^{2}/_{y}\right)\left[{}^{5}/_{x}\right] = \left({}^{5}/_{x}, {}^{2}/_{y}\right)$.

$$\dfrac{\dfrac{}{\langle y, \left({}^{5}/_{x}, {}^{2}/_{y}\right)\rangle \to 2}\,\text{ide} \qquad \dfrac{}{\langle 1, \left({}^{5}/_{x}, {}^{2}/_{y}\right)\rangle \to 1}\,\text{num}}{\langle y - 1, \left({}^{5}/_{x}, {}^{2}/_{y}\right)\rangle \to m'' = 2 - 1 = 1}\,\text{dif}$$

Since $m'' = 1$ *we know that* $\sigma'' = \left({}^{5}/_{x}, {}^{2}/_{y}\right)\left[{}^{m''}/_{y}\right] = \left({}^{5}/_{x}, {}^{2}/_{y}\right)\left[{}^{1}/_{y}\right] = \left({}^{5}/_{x}, {}^{1}/_{y}\right)$.

Next we prove $\langle c_1, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \sigma$, *this time omitting some details (the derivation is analogous to the one just seen).*

$$\dfrac{\dfrac{\vdots}{\langle 0 \le y, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \textbf{true}}\,\text{leq} \qquad \dfrac{\vdots}{\langle c_2, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \left({}^{5}/_{x}, {}^{1}/_{y}\right)\left[{}^{8}/_{x}\right]\left[{}^{0}/_{y}\right] = \sigma''''}\,\text{seq} \qquad \langle c_1, \sigma''''\rangle \to \sigma}{\langle c_1, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \sigma}\,\text{whtt}$$

Hence $\sigma'''' = \left({}^{8}/_{x}, {}^{0}/_{y}\right)$ *and next we prove* $\langle c_1, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \sigma$.

$$\dfrac{\dfrac{\vdots}{\langle 0 \le y, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \textbf{true}}\,\text{leq} \qquad \dfrac{\vdots}{\langle c_2, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \left({}^{8}/_{x}, {}^{0}/_{y}\right)\left[{}^{9}/_{x}\right]\left[{}^{-1}/_{y}\right] = \sigma'''''}\,\text{seq} \qquad \langle c_1, \sigma'''''\rangle \to \sigma}{\langle c_1, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \sigma}\,\text{whtt}$$

Hence $\sigma''''' = \left({}^{9}/_{x}, {}^{-1}/_{y}\right)$. *Finally:*

$$\dfrac{\dfrac{\vdots}{\langle 0 \le y, \left({}^{9}/_{x}, {}^{-1}/_{y}\right)\rangle \to \textbf{false}}\,\text{leq}}{\langle c_1, \left({}^{9}/_{x}, {}^{-1}/_{y}\right)\rangle \to \left({}^{9}/_{x}, {}^{-1}/_{y}\right) = \sigma}\,\text{whff}$$

Summing up all the above, we have proved the theorem $\langle c, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \left({}^{9}/_{x}, {}^{-1}/_{y}\right)$.

It is evident that as the proof tree grows larger it gets harder to paste the different pieces of the proof together. We now show the same proof as a goal-oriented derivation, which should be easier to follow. To this aim, we group several derivation steps into a single one omitting trivial steps.

$$
\begin{array}{rl}
\langle c, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma & \nwarrow \quad \langle x := 0, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma', \quad \langle c_1, \sigma'\rangle \to \sigma \\
\nwarrow_{\sigma' = ({}^{27}/_{x}, {}^{2}/_{y})[{}^{n}/_{x}]} & \langle 0, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to n, \quad \langle c_1, \left({}^{27}/_{x}, {}^{2}/_{y}\right)[{}^{n}/_{x}]\rangle \to \sigma \\
\nwarrow_{n=0\ \sigma' = ({}^{0}/_{x}, {}^{2}/_{y})} & \langle c_1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma \\
\nwarrow & \langle 0 \le y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \textbf{true}, \ \langle c_2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma'', \ \langle c_1, \sigma''\rangle \to \sigma \\
\nwarrow & \langle 0, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to n_1, \quad \langle y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to n_2, \quad n_1 \le n_2, \\
& \langle c_2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma'', \quad \langle c_1, \sigma''\rangle \to \sigma \\
\nwarrow^{*}_{n_1=0\ n_2=2} & \langle c_3, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma''', \quad \langle c_4, \sigma'''\rangle \to \sigma'', \quad \langle c_1, \sigma''\rangle \to \sigma \\
\nwarrow_{\sigma''' = ({}^{0}/_{x}, {}^{2}/_{y})[{}^{m}/_{x}]} & \langle x + (2 \times y) + 1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m, \ \langle c_4, \left({}^{0}/_{x}, {}^{2}/_{y}\right)[{}^{m}/_{x}]\rangle \to \sigma'', \\
& \langle c_1, \sigma''\rangle \to \sigma \\
\nwarrow^{*}_{m=0+(2\times 2)+1=5\ \sigma''' = ({}^{5}/_{x}, {}^{2}/_{y})} & \langle c_4, \left({}^{5}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma'', \quad \langle c_1, \sigma''\rangle \to \sigma \\
\nwarrow^{*}_{\sigma'' = ({}^{5}/_{x}, {}^{2}/_{y})[{}^{2-1}/_{y}] = ({}^{5}/_{x}, {}^{1}/_{y})} & \langle c_1, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \sigma \\
\nwarrow^{*}_{\sigma'''' = ({}^{5}/_{x}, {}^{1}/_{y})[{}^{5+2\times 1+1}/_{x}][{}^{0}/_{y}] = ({}^{8}/_{x}, {}^{0}/_{y})} & \langle c_1, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \sigma \\
\nwarrow^{*}_{\sigma''''' = ({}^{8}/_{x}, {}^{0}/_{y})[{}^{8+2\times 0+1}/_{x}][{}^{0-1}/_{y}] = ({}^{9}/_{x}, {}^{-1}/_{y})} & \langle c_1, \left({}^{9}/_{x}, {}^{-1}/_{y}\right)\rangle \to \sigma \\
\nwarrow_{\sigma = ({}^{9}/_{x}, {}^{-1}/_{y})} & \langle 0 \le y, \left({}^{9}/_{x}, {}^{-1}/_{y}\right)\rangle \to \textbf{false} \\
\nwarrow^{*} & \square
\end{array}
$$

There are commands c and memories σ such that there is no σ' for which we can find a proof of $\langle c, \sigma\rangle \to \sigma'$. We use the notation below to denote such cases:

$$\langle c, \sigma\rangle \nrightarrow \quad \text{iff} \quad \neg\exists \sigma'. \langle c, \sigma\rangle \to \sigma'$$

The condition $\neg\exists\sigma'.\langle c, \sigma\rangle \to \sigma'$ can be written equivalently as $\forall\sigma'.\langle c, \sigma\rangle \nrightarrow \sigma'$.

Example 2.8 (Non termination)
Let us consider the command

$$c = \textbf{while true do skip}$$

Given σ, the only possible derivation goes as follows:

$$
\begin{array}{rl}
\langle c, \sigma\rangle \to \sigma' & \nwarrow \quad \langle \textbf{true}, \sigma\rangle \to \textbf{true}, \ \langle \textbf{skip}, \sigma\rangle \to \sigma'', \ \langle c, \sigma''\rangle \to \sigma' \\
& \nwarrow \quad \langle \textbf{skip}, \sigma\rangle \to \sigma'', \ \langle c, \sigma''\rangle \to \sigma' \\
& \nwarrow_{\sigma'' = \sigma} \quad \langle c, \sigma\rangle \to \sigma'
\end{array}
$$

After a few steps of derivation we reach the same goal from which we started! There is no alternative to try!

We can prove that $\langle c, \sigma\rangle \nrightarrow$. We proceed by contradiction, assuming there exists σ' for which we can find a (finite) derivation d for $\langle c, \sigma\rangle \to \sigma'$. Let d be the derivation sketched below:

$$
\begin{array}{rcl}
\langle c, \sigma\rangle \to \sigma' & \nwarrow & \langle \textbf{true}, \sigma\rangle \to \textbf{true},\ \langle \textbf{skip}, \sigma\rangle \to \sigma'',\ \langle c, \sigma''\rangle \to \sigma' \\
 & \vdots & \\
(*) & \nwarrow & \langle c, \sigma\rangle \to \sigma' \\
 & \vdots & \\
 & \nwarrow & \square
\end{array}
$$

We have marked by $(*)$ *the last occurrence of the goal* $\langle c, \sigma\rangle \to \sigma'$*. But this leads to a contradiction, because the next step of the derivation can only be obtained by applying rule (whtt) and therefore it should lead to another instance of the original goal.*

2.3. Abstract Semantics: Equivalence of IMP Expressions and Commands

The same way as we can write different expressions denoting the same value, we can write different programs for solving the same problem. For example we are used not to distinguish between say $2+2$ and 2×2 because both evaluate to 4. Similarly, would you distinguish between say $x := 1; y := 0$ and $y := 0; x := y + 1$? So a natural question arise: when are two programs "equivalent"? Informally, two programs are equivalent if they *behave in the same way*. But can we make this idea more precise?

The equivalence between two commands is an important issue because we know that two commands are equivalent, then we can replace one for the other in any larger program without changing the overall behaviour. Since the evaluation of a command depends on the memory, two equivalent programs must behave the same *w.r.t. any initial memory*. For example the two commands $x := 1$ and $x := y + 1$ assign the same value to x only when evaluated in a memory σ such that $\sigma(y) = 0$, so that it wouldn't be safe to replace one for the other in any program. Moreover, we must take into account that commands may diverge when evaluated with certain memory state. We will call *abstract semantics* the notion of behaviour w.r.t. we will compare programs for equivalence.

The operational semantics offers a straightforward abstract semantics: two programs are equivalent if they result in the same memory when evaluated over the same initial memory.

Definition 2.9 (Equivalence of expressions and commands)
We say that the arithmetic expressions a_1 *and* a_2 *are* equivalent*, written* $a_1 \sim a_2$ *if and only if for any memory* σ *they evaluate in the same way. Formally:*

$$a_1 \sim a_2 \quad \textit{iff} \quad \forall \sigma, n.(\ \langle a_1, \sigma\rangle \to n \ \Leftrightarrow\ \langle a_2, \sigma\rangle \to n\ )$$

We say that the boolean expressions b_1 *and* b_2 *are* equivalent*, written* $b_1 \sim b_2$ *if and only if for any memory* σ *they evaluate in the same way. Formally:*

$$b_1 \sim b_2 \quad \textit{iff} \quad \forall \sigma, v.(\ \langle b_1, \sigma\rangle \to v \ \Leftrightarrow\ \langle b_2, \sigma\rangle \to v\ )$$

We say that the commands c_1 *and* c_2 *are* equivalent*, written* $c_1 \sim c_2$ *if and only if for any memory* σ *they evaluate in the same way. Formally:*

$$c_1 \sim c_2 \quad \textit{iff} \quad \forall \sigma, \sigma'.(\ \langle c_1, \sigma\rangle \to \sigma' \ \Leftrightarrow\ \langle c_2, \sigma\rangle \to \sigma'\ )$$

Notice that if the evaluation of $\langle c_1, \sigma\rangle$ diverges we have no σ' such that $\langle c_1, \sigma\rangle \to \sigma'$. Then, when $c_1 \sim c_2$, the double implication prevents $\langle c_2, \sigma\rangle$ to converge. As an easy consequence, any two programs that diverge for any σ are equivalent.

2.3.1. Examples: Simple Equivalence Proofs

The first example we show is concerned with a fully specified program that operates on an unspecified memory.

Example 2.10 (Equivalent commands)
Let us try to prove that the following two commands are equivalent:

$$c_1 = \textbf{while}\ x \neq 0\ \textbf{do}\ x := 0$$
$$c_2 = x := 0$$

It is immediate to prove that

$$\forall\sigma.\langle c_2, \sigma\rangle \to \sigma' = \sigma[0/x]$$

Hence σ and σ' can differ only for the value stored in x. In particular, if $\sigma(x) = 0$ then $\sigma' = \sigma$.
The evaluation of c_1 in σ depends on $\sigma(x)$: if $\sigma(x) = 0$ we must apply the rule 2.18 (whff)*, otherwise the rule 2.17* (whtt) *must be applied. Since we do not know the value of $\sigma(x)$, we consider the two cases separately.*

($\sigma(x) \neq 0$)

$$\begin{array}{lll}
\langle c_1, \sigma\rangle \to \sigma' & \nwarrow & \langle x \neq 0, \sigma\rangle \to \textbf{true}, \quad \langle x := 0, \sigma\rangle \to \sigma'', \quad \langle c_1, \sigma''\rangle \to \sigma' \\
& \nwarrow^{*}_{\sigma''=\sigma[0/x]} & \langle c_1, \sigma\left[0/x\right]\rangle \to \sigma' \\
& \nwarrow_{\sigma'=\sigma[0/x]} & \langle x \neq 0, \sigma\left[0/x\right]\rangle \to \textbf{false} \\
& \nwarrow^{*} & \sigma\left[0/x\right](x) = 0 \\
& \nwarrow & \square
\end{array}$$

($\sigma(x) = 0$)

$$\begin{array}{lll}
\langle c_1, \sigma\rangle \to \sigma' & \nwarrow_{\sigma'=\sigma} & \langle x \neq 0, \sigma\rangle \to \textbf{false} \\
& \nwarrow^{*} & \sigma(x) = 0 \\
& \nwarrow & \square
\end{array}$$

Finally, we observe the following:

- *If $\sigma(x) = 0$, then* $\begin{cases} \langle c_1, \sigma\rangle \to \sigma \\ \langle c_2, \sigma\rangle \to \sigma[0/x] = \sigma \end{cases}$
- *Otherwise, if $\sigma(x) \neq 0$, then* $\begin{cases} \langle c_1, \sigma\rangle \to \sigma[0/x] \\ \langle c_2, \sigma\rangle \to \sigma[0/x] \end{cases}$

Therefore $c_1 \sim c_2$ because for any σ they result in the same memory.

The general methodology should be clear by now: in case the computation terminates we need just to develop the derivation and check the results.

2.3.2. Examples: Parametric Equivalence Proofs

The programs considered so far were entirely spelled out: all the commands and expressions were given and the only unknown parameter was the initial memory σ. In this section we address equivalence proofs for programs that contain symbolic expressions a and b and symbolic commands c: we will need to prove that the equality holds for any such a, b and c.

This is not necessarily more complicated than what we have done already: the idea is that we can just carry the derivation with symbolic parameters.

Example 2.11 (Parametric proofs (1))
Let us consider the commands:

$$c_1 = \textbf{while } b \textbf{ do } c$$
$$c_2 = \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip} = \textbf{if } b \textbf{ then } (c; c_1) \textbf{ else skip}$$

Is it true that $\forall b, c.\ (c_1 \sim c_2)$*?*

We start by considering the derivation for c_1 *in a generic initial memory* σ*. The command* c_1 *is a cycle and there are two rules we can apply: either the rule 2.18* (whff)*, or the rule 2.17* (whtt)*. Which rule to use depends on the evaluation of b. Since we do not know what b is, we must take into account both possibilities and consider the two cases separately.*

($\langle b, \sigma\rangle \to \textbf{false}$***)***

$$\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma' \quad \nwarrow_{\sigma'=\sigma} \quad \langle b, \sigma\rangle \to \textbf{false}$$
$$\nwarrow \qquad \Box$$

$$\langle \textbf{if } b \textbf{ then } (c; c_1) \textbf{ else skip}, \sigma\rangle \to \sigma' \quad \nwarrow \qquad \langle b, \sigma\rangle \to \textbf{false}, \quad \langle \textbf{skip}, \sigma\rangle \to \sigma'$$
$$\nwarrow_{\sigma'=\sigma} \quad \Box$$

It is evident that if $\langle b, \sigma\rangle \to \textbf{false}$ *then the two derivations for* c_1 *and* c_2 *lead to the same result.*

($\langle b, \sigma\rangle \to \textbf{true}$***)***

$$\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma' \quad \nwarrow \quad \langle b, \sigma\rangle \to \textbf{true}, \quad \langle c, \sigma\rangle \to \sigma'', \quad \langle c_1, \sigma''\rangle \to \sigma'$$
$$\nwarrow \quad \langle c, \sigma\rangle \to \sigma'', \quad \langle c_1, \sigma''\rangle \to \sigma'$$

We find it convenient to stop here the derivation, because otherwise we should add further hypotheses on the evaluation of c and of the guard b after the execution of c. Instead, let us look at the derivation of c_2*:*

$$\langle \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip}, \sigma\rangle \to \sigma' \quad \nwarrow \quad \langle b, \sigma\rangle \to \textbf{true}, \quad \langle c; \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$
$$\nwarrow \quad \langle c; \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$
$$\nwarrow \quad \langle c, \sigma\rangle \to \sigma'', \quad \langle c_1, \sigma''\rangle \to \sigma'$$

Now we can stop again, because we have reached exactly the same subgoals that we have obtained evaluating c_1*! It is then obvious that if* $\langle b, \sigma\rangle \to \textbf{true}$ *then the two derivations for* c_1 *and* c_2 *will necessarily lead to the same result whenever they terminate, and if one diverge the other diverges too.*

Summing up the two cases, and since there are no more alternatives to try, we can conclude that $c_1 \sim c_2$*.*

Note that the equivalence proof technique that exploits reduction to the same subgoals is one of the most convenient methods for proving the equivalence of **while** commands, whose evaluation may diverge.

Example 2.12 (Parametric proofs (2))
Let us consider the commands:

$$c_1 = \textbf{while } b \textbf{ do } c$$
$$c_2 = \textbf{if } b \textbf{ then } c_1 \textbf{ else skip}$$

Is it true that $\forall b, c.\ c_1 \sim c_2$*?*

We have already examined the different derivations for c_1 *in the previous example. Moreover, the evaluation of* c_2 *when* $\langle b, \sigma\rangle \to \textbf{false}$ *is also analogous to that of the command* c_2 *in Example 2.11. Therefore we focus on the analysis of* c_2 *for the case* $\langle b, \sigma\rangle \to \textbf{true}$*. Trivially:*

$$\langle \textbf{if } b \textbf{ then}(\textbf{while } b \textbf{ do } c) \textbf{ else skip}, \sigma\rangle \to \sigma' \quad \nwarrow \quad \langle b, \sigma\rangle \to \textbf{true}, \quad \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$
$$\nwarrow \quad \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$

So we reduce to the subgoal identical to the evaluation of c_1, and we can conclude that $c_1 \sim c_2$.

2.3.3. Inequality Proofs

The next example deals with programs that can behave the same or exhibit different behaviours depending on the initial memory.

Example 2.13 (Inequality proof)
Let us consider the commands:

$$c_1 = (\textbf{while } x > 0 \textbf{ do } x := 1); x := 0$$
$$c_2 = x := 0$$

Let us prove that $c_1 \nsim c_2$.
For c_2 we have

$$\frac{\dfrac{n = 0}{\langle 0, \sigma\rangle \to n}}{\langle x := 0, \sigma\rangle \to \sigma[n/x]}$$

That is: $\forall\sigma.\ \langle x := 0, \sigma\rangle \to \sigma[0/x]$.
Next, we focus on the first part of c_1. Assuming $\sigma(x) \leq 0$ it is easy to check that

$$\langle \textbf{while } x > 0 \textbf{ do } x := 1, \sigma\rangle \to \sigma$$

The derivation is sketched below:

$$\frac{\dfrac{\dfrac{n = \sigma(x)}{\langle x, \sigma\rangle \to n} \qquad \dfrac{}{\langle 0, \sigma\rangle \to 0}}{\langle x > 0, \sigma\rangle \to \textbf{false}}\ (n > 0) = \textbf{false}}{\langle \textbf{while } x > 0 \textbf{ do } x := 1, \sigma\rangle \to \sigma}$$

Instead, if we assume $\sigma(x) > 0$, then:

$$\frac{\dfrac{\cdots}{\langle x > 0, \sigma\rangle \to \textbf{true}} \quad \dfrac{\sigma'' = \sigma[1/x]}{\langle x := 1, \sigma\rangle \to \sigma''} \quad \dfrac{?}{\langle \textbf{while } x > 0 \textbf{ do } x := 1, \sigma''\rangle \to \sigma'}}{\langle \textbf{while } x > 0 \textbf{ do } x := 1, \sigma\rangle \to \sigma'}$$

Let us expand the derivation for $\langle \textbf{while } x > 0 \textbf{ do } x := 1, \sigma[1/x]\rangle \to \sigma'$:

$$\frac{\dfrac{\cdots}{\langle x > 0, \sigma[1/x]\rangle \to \textbf{true}} \quad \dfrac{\cdots}{\langle x := 1, \sigma[1/x]\rangle \to \sigma[1/x]} \quad \dfrac{?}{\langle \textbf{while } x > 0 \textbf{ do } x := 1, \sigma[1/x]\rangle \to \sigma'}}{\langle \textbf{while } x > 0 \textbf{ do } x := 1, \sigma[1/x]\rangle \to \sigma'}$$

Now, note that we got the same subgoal $\langle \textbf{while } x > 0 \textbf{ do } x := 1, \sigma[1/x]\rangle \to \sigma'$ already inspected: hence it is not possible to conclude the derivation, which will loop.

Summing up all the above we conclude that:

$$\forall \sigma, \sigma'. \langle \textbf{while}\ x > 0\ \textbf{do}\ x := 1, \sigma\rangle \to \sigma' \Rightarrow \sigma(x) \leq 0 \wedge \sigma' = \sigma$$

We can now complete the reduction for the whole c_1 when $\sigma(x) \leq 0$ (the case $\sigma(x) > 0$ is discharged, because we know that there is no derivation).

$$\dfrac{\dfrac{\sigma(x) \leq 0}{\vdots} \quad \dfrac{\sigma' = \sigma[0/x]}{\vdots}}{\dfrac{\langle (\textbf{while}\ x > 0\ \textbf{do}\ x := 1), \sigma\rangle \to \sigma \quad \langle x := 0, \sigma\rangle \to \sigma'}{\langle (\textbf{while}\ x > 0\ \textbf{do}\ x := 1); x := 0, \sigma\rangle \to \sigma'}}$$

Therefore the evaluation ends with $\sigma' = \sigma[0/x]$.
By comparing c_1 and c_2 we have that:

- *there are memories for which the two commands behave the same (i.e., when $\sigma(x) \leq 0$)*

$$\exists \sigma, \sigma'. \begin{cases} \langle (\textbf{while}\ x > 0\ \textbf{do}\ x := 1); x := 0, \sigma\rangle \to \sigma' \\ \langle x := 0, \sigma\rangle \to \sigma' \end{cases}$$

- *there are also cases for which the two commands exhibit different behaviours:*

$$\exists \sigma, \sigma'. \begin{cases} \langle (\textbf{while}\ x > 0\ \textbf{do}\ x := 1); x := 0, \sigma\rangle \nrightarrow \\ \langle x := 0, \sigma\rangle \to \sigma' \end{cases}$$

As an example, take any σ with $\sigma(x) = 1$ and $\sigma' = \sigma[0/x]$.

Since we can find pairs (σ, σ') such that c_1 loops and c_2 terminates we have that $c_1 \nsim c_2$.
Note that in disproving the equivalence we have exploited a standard technique in logic: to show that a universally quantified formula is not valid we can exhibit one counterexample. Formally:

$$\neg \forall x.(P(x) \Leftrightarrow Q(x)) = \exists x.(P(x) \wedge \neg Q(x)) \vee (\neg P(x) \wedge Q(x))$$

2.3.4. Diverging Computations

What does it happen if the program has infinite looping situations when σ meets certain conditions? How should we handle the σ for which this happens?

Let us rephrase the definition of equivalence between commands:

$$\forall \sigma, \sigma' \quad \begin{cases} \langle c_1, \sigma\rangle \to \sigma' & \Leftrightarrow & \langle c_2, \sigma\rangle \to \sigma' \\ \langle c_1, \sigma\rangle \nrightarrow & \Leftrightarrow & \langle c_2, \sigma\rangle \nrightarrow \end{cases}$$

Next we see an example where this situation emerges.

Example 2.14 (Proofs of non-termination)
Let us consider the commands:

$$c_1 = \textbf{while}\ x > 0\ \textbf{do}\ x := 1$$
$$c_2 = \textbf{while}\ x > 0\ \textbf{do}\ x := x + 1$$

Is it true that $c_1 \sim c_2$? On the one hand, note that c_1 can only store 1 in x, whereas c_2 can keep incrementing the value stored in x, so one may lead to suspect that the two commands are not equivalent. On the other hand, we know that when the commands diverge, the values stored in the memory locations are inessential.

As already done in previous examples, let us focus on the possible derivation of c_1 by considering two separate cases that depends of the evaluation of the guard $x > 0$:

($\sigma(x) \leq 0$)

$$\langle c_1, \sigma\rangle \to \sigma' \quad \nwarrow_{\sigma'=\sigma} \quad \langle x > 0, \sigma\rangle \to \textbf{false}$$
$$\nwarrow \qquad\quad \square$$

*In this case, the body of the **while** is not executed and the resulting memory is left unchanged. We leave to the reader to fill the details for the analogous derivation of c_2, which behaves the same.*

($\sigma(x) > 0$)

$$\begin{array}{rl}
\langle c_1, \sigma\rangle \to \sigma' \quad \nwarrow & \langle x > 0, \sigma\rangle \to \textbf{true}, \quad \langle x := 1, \sigma\rangle \to \sigma'', \quad \langle c_1, \sigma''\rangle \to \sigma' \\
\nwarrow & \langle x := 1, \sigma\rangle \to \sigma'', \quad \langle c_1, \sigma''\rangle \to \sigma' \\
\nwarrow_{\sigma''=\sigma[^1/_x]} & \langle c_1, \sigma\left[^1/_x\right]\rangle \to \sigma' \\
\nwarrow & \langle x > 0, \sigma\left[^1/_x\right]\rangle \to \textbf{true}, \quad \langle x := 1, \sigma\left[^1/_x\right]\rangle \to \sigma''', \quad \langle c_1, \sigma'''\rangle \to \sigma' \\
\nwarrow & \langle x := 1, \sigma\left[^1/_x\right]\rangle \to \sigma''', \quad \langle c_1, \sigma'''\rangle \to \sigma' \\
\nwarrow_{\sigma'''=\sigma[^1/_x][^1/_x]=\sigma[^1/_x]} & \langle c_1, \sigma\left[^1/_x\right]\rangle \to \sigma' \\
\nwarrow & \ldots
\end{array}$$

Note that we reach the same subgoal $\langle c_1, \sigma\left[^1/_x\right]\rangle \to \sigma'$ already inspected, i.e., the derivation will loop.

Now we must check if c_2 diverges too when $\sigma(x) > 0$:

$$\begin{array}{rl}
\langle c_2, \sigma\rangle \to \sigma' \quad \nwarrow & \langle x > 0, \sigma\rangle \to \textbf{true}, \quad \langle x := x + 1, \sigma\rangle \to \sigma'', \quad \langle c_2, \sigma''\rangle \to \sigma' \\
\nwarrow & \langle x := x + 1, \sigma\rangle \to \sigma'', \quad \langle c_2, \sigma''\rangle \to \sigma' \\
\nwarrow_{\sigma''=\sigma[^{\sigma(x)+1}/_x]} & \langle c_2, \sigma\left[^{\sigma(x)+1}/_x\right]\rangle \to \sigma' \\
\nwarrow & \langle x > 0, \sigma\left[^{\sigma(x)+1}/_x\right]\rangle \to \textbf{true}, \quad \langle x := x + 1, \sigma\left[^{\sigma(x)+1}/_x\right]\rangle \to \sigma''', \\
 & \langle c_2, \sigma'''\rangle \to \sigma' \\
\nwarrow & \langle x := x + 1, \sigma\left[^{\sigma(x)+1}/_x\right]\rangle \to \sigma''', \quad \langle c_2, \sigma'''\rangle \to \sigma' \\
\nwarrow_{\sigma'''=\sigma''[^{\sigma''(x)+1}/_x]=\sigma[^{\sigma(x)+2}/_x]} & \ldots
\end{array}$$

Now the situation is more subtle: we keep looping, but without crossing the same subgoal twice, because the memory is updated with different values for x at each iteration. However, using induction, that will be the subject of Section 3.1.3, we can prove that the derivation will not terminate. Roughly, the idea is the following:

- *at step 0, i.e. at the first iteration, the cycle does not terminate;*
- *if at the ith step the cycle has not terminated yet, then it will not terminate at the $(i + 1)$th step, because $x > 0 \Rightarrow x + 1 > 0$.*

The formal proof would require to show that at the kth iteration the values stored in the memory at location x will be $\sigma(x) + k$, from which we can conclude that the expression $x > 0$ will hold true (since by assumption $\sigma(x) > 0$ and thus $\sigma(x) + k > 0$). Once the proof is completed, we can conclude that c_2 diverges and therefore $c_1 \sim c_2$.

Let us consider the command $w = \textbf{while}\, b\, \textbf{do}\, c$. As we have seen in the last example, to prove the non-termination of w we can exploit the induction hypotheses over memory states to define the following

inference rule:

$$\frac{\sigma \in S \quad \forall \sigma' \in S. \forall \sigma''. (\langle c, \sigma' \rangle \to \sigma'' \implies \sigma'' \in S) \quad \forall \sigma' \in S. (\langle b, \sigma' \rangle \to \mathbf{true})}{\langle w, \sigma \rangle \nrightarrow} \tag{2.19}$$

If we can find a set S of memories such that, for any $\sigma' \in S$, the guard b is evaluated to **true** and the execution of c leads to a memory which is also in S, then we can conclude that w diverges when evaluated in any of the memories $\sigma \in S$. Note that the condition

$$(\langle c, \sigma' \rangle \to \sigma'' \implies \sigma'' \in S)$$

is satisfied even when $\langle c, \sigma' \rangle \nrightarrow$, as the left-hand side of the implication is false and therefore the implication is true.

3. Induction and Recursion

In this chapter we presents some induction techniques that will turn out useful for proving formal properties of the languages and models presented in the course.

In the literature several different kinds of induction are defined, but they all rely on the so-called *Noether principle of well-founded induction*. We start by defining this important principle and will derive several induction methods.

3.1. Noether Principle of Well-founded Induction

3.1.1. Well-founded Relations

We recall some key mathematical notions and definitions.

Definition 3.1 (Binary relation)
A (binary) relation $\prec$ *over a set A is a subset of the cartesian product* $A \times A$.

$$\prec \subseteq A \times A$$

For $(a, b) \in \prec$ we use the infix notation $a \prec b$ and also write equivalently $b \succ a$. A relation $\prec \subseteq A \times A$ can be conveniently represented as an *oriented graph* whose nodes are the elements of A and whose arcs $n \to m$ represent the pairs $(n, m) \in \prec$ in the relation. For instance, the graph in Fig. 3.1 represents the relation $(\{a, b, c, d, e, f\}, \prec)$ with $a \prec b$, $b \prec c$, $c \prec d$, $c \prec e$, $e \prec f$, $e \prec b$.

Definition 3.2 (Infinite descending chain)
Given a relation $\prec$ *over the set A, an* infinite descending chain *is an infinite sequence* $\{a_i\}_{i \in \omega}$ *of elements in A such that*

$$\forall i \in \omega.\, a_{i+1} \prec a_i$$

An infinite descending chain can be represented as a function a from ω to A such that $a(i)$ decreases (according to $\prec$) as i grows:

$$a(0) \succ a(1) \succ a(2) \succ \cdots$$

Definition 3.3 (Well-founded relation)
A relation is well-founded *if it admits no infinite descending chains.*

Definition 3.4 (Transitive closure)
Let $\prec$ *be a relation over A. The* transitive closure *of* $\prec$*, written* $\prec^+$*, is defined by the following inference rules*

$$\frac{a \prec b}{a \prec^+ b} \qquad \frac{a \prec^+ b \quad b \prec^+ c}{a \prec^+ c}$$

Definition 3.5 (Transitive and reflexive closure)
Let $\prec$ *be a relation over A. The* transitive and reflexive closure *of* $\prec$*, written* $\prec^*$*, is defined by the following inference rules*

$$a \prec^* a \qquad \frac{a \prec^* b \qquad b \prec c}{a \prec^* c}$$

Theorem 3.6
Let $\prec$ be a relation over A. For any $x, y \in A$, $x \prec^+ y$ if and only if there exist a finite number of elements $z_0, z_1, ..., z_k \in A$ such that

$$x = z_0 \prec z_1 \prec \cdots \prec z_k = y.$$

Theorem 3.7 (Well-foundedness of $\prec^+$)
A relation $\prec$ is well-founded if and only if its transitive closure $\prec^+$ is well-founded.

Proof. One implication is trivial: if $\prec^+$ is well-founded then $\prec$ is obviously well-founded, because any descending chain for $\prec$ is also a descending chain for $\prec^+$ (and all such chains are finite by hypothesis).

For the other direction, let us assume $\prec^+$ is non well-founded and take any infinite descending chain

$$a_0 \succ^+ a_1 \succ^+ a_2 \cdots$$

But whenever $a_i \succ^+ a_{i+1}$ there must be a finite descending $\prec$-chain of elements between a_i and a_{i+1} and therefore we can build an infinite descending chain

$$a_0 \succ \cdots \succ a_1 \succ \cdots \succ a_2 \succ \cdots$$

leading to a contradiction. □

Definition 3.8 (Acyclic relation)
We say that $\prec$ has a cycle if $\exists a \in A.\, a \prec^+ a$. We say that $\prec$ is acyclic *if it has no cycle.*

Theorem 3.9 (Well-founded relations are acyclic)
If the relation $\prec$ is well-founded, then it is acyclic.

Proof. We need to prove that:

$$\forall x \in A.\, x \not\prec^+ x$$

By contradiction, we assume there is $x \in A$ such that $x \prec^+ x$. This means that there exist finitely many elements $x_1, x_2, \cdots, x_n \in A$ such that

$$x \prec x_1 \prec \ldots \prec x_n \prec x$$

Then, we can build an infinite sequence by cycling on such elements:

$$x \succ x_n \succ \ldots \succ x_1 \succ x \succ x_n \succ \ldots \succ x_1 \succ x \succ \ldots$$

The infinite sequence above is clearly an infinite descending chain, leading to a contradiction, because $\prec$ is well-founded by hypothesis. □

For example, the relation in Fig. 3.1 is not acyclic and thus it is not well-founded.

Theorem 3.10 (Well-founded relations over finite sets)
Let A be a finite set and let $\prec$ be acyclic, then $\prec$ is well-founded.

Proof. Since A is finite, any descending chain strictly longer than $\|A\|$ must contain (at least) two occurrences of a same element (by the so-called "pigeon hole principle") that form a cycle, but this is not possible because $\prec$ is acyclic by hypothesis. □

Definition 3.11 (Minimal element)
Let $\prec$ be a relation over the set A. Given a set $Q \subseteq A$, we say that $m \in Q$ is minimal *if there is no element $x \in Q$ such that $x \prec m$, i.e., $\forall x \in Q.\, x \not\prec m$.*

It follows that Q has no minimal element if $\forall m \in Q.\, \exists x \in Q.\, x \prec m$.

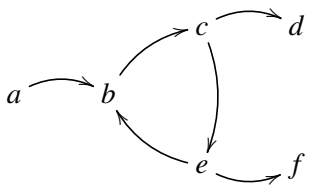


Figure 3.1.: Graph of a relation

Lemma 3.12 (Well-founded relation)
Let $\prec$ be a relation over the set A. The relation $\prec$ is well-founded if and only if every nonempty subset $Q \subseteq A$ contains a minimal element m.

Proof. The Lemma can be rephrased by saying that the relation $\prec$ has an infinite descending chain if and only if there exists a nonempty subset $Q \subseteq A$ with no minimal element.

As common when a double implication is involved in the statement, we prove each implication separately.

(⇐) We assume that every nonempty subset of A has a minimal element and we need to show that $\prec$ has no infinite descending chain. By contradiction, we assume that $\prec$ has an infinite descending chain $a_1 \succ a_2 \succ a_3 \succ \cdots$ and we let $Q = \{a_1, a_2, a_3, \ldots\}$ be the set of all the elements in the infinite descending chain. The set Q has no minimal element, because for any candidate $a_i \in Q$ we know there is one element $a_{i+1} \in Q$ with $a_i \succ a_{i+1}$. This contradicts the hypothesis, concluding the proof.

(⇒) We assume the relation $\prec$ is well-founded and we need to show that every nonempty subset Q of A has a minimal element. By contradiction, let Q be a nonempty subset of A with no minimal element. Since Q is nonempty, it must contain at least an element. We randomly pick an element $a_0 \in Q$. Since a_0 is not minimal there must exists an element $a_1 \in Q$ such that $a_0 \succ a_1$, and we can iterate the reasoning (i.e. a_1 is not minimal and there is $a_2 \in Q$ with $a_0 \succ a_1 \succ a_2$, etc.). But since $\prec$ is well-founded, we cannot build such an infinite descending chain.

□

Example 3.13 (Natural numbers)
Both $n \prec n + 1$ and $n \prec n + 1 + k$, with n and k in the set ω of natural numbers, are simple examples of well-founded relations. In fact, from every element $n \in \omega$ we can start a descending chain of at most length n.

Definition 3.14 (Terms over one-sorted signatures)
Let $\Sigma = \{\Sigma_n\}_{n\in\omega}$ a one-sorted signature, i.e., a set of ranked operators f such that $f \in \Sigma_n$ if f takes n arguments. We define the set of Σ-terms as the set T_Σ that is defined inductively by the following inference rule:

$$\frac{t_i \in T_\Sigma \quad i = 1, \ldots, n \quad f \in \Sigma_n}{f(t_1, \ldots, t_n) \in T_\Sigma}$$

Definition 3.15 (Terms over sorted signatures)
Let

- *S be a set of* sorts *(i.e. the set of the different data types we want to consider);*
- *$\Sigma = \{\Sigma_{s_1\ldots s_n,s}\}_{s_1\ldots s_n,s\in S}$ a* signature *over S, i.e. a set of typed operators ($f \in \Sigma_{s_1\ldots s_n,s}$ is an operator that takes n arguments, the ith argument being of type s_i, and then returns a result of type s).*

We define the set of Σ-terms *as the set*

$$T_\Sigma = \{T_{\Sigma,s}\}_{s\in S}$$

where, for $s \in S$, the set $T_{\Sigma,s}$ is the set of terms of sort s over the signature Σ, defined inductively by the following inference rule:

$$\frac{t_i \in T_{\Sigma,s_i} \quad i = 1,\dots,n \quad f \in \Sigma_{s_1,\dots,s_n,s}}{f(t_1,\dots,t_n) \in T_{\Sigma,s}}$$

(When S is a singleton, we write just Σ_n instead of $\Sigma_{w,s}$ with $w = \underbrace{s \dots s}_{n}$.)

Since the operators of the signature are known, we can specialize the above rule for each operator, i.e. we can consider the set of inference rules:

$$\left\{ \frac{t_i \in T_{\Sigma,s_i} \quad i = 1,\dots,n}{f(t_1,\dots,t_n) \in T_{\Sigma,s}} \right\}_{f \in \Sigma_{s_1,\dots,s_n,s}}$$

Note that, as special case of the above inference rule, for constants $a \in \Sigma_{\epsilon,s}$ we have:

$$\frac{}{a \in T_{\Sigma,s}}$$

Example 3.16 (IMP Signature)
In the case of IMP, we have $S = \{Aexp, Bexp, Com\}$ and then we have an operation for each production in the grammar.

For example, the sequential composition of commands ";" corresponds to the binary infix operator $(-;-) \in \Sigma_{ComCom,Com}$.

Similarly the equality expression is built using the operator $(- = -) \in \Sigma_{AexpAexp,Bexp}$.

By abusing the notation, we often write Com for $T_{\Sigma,Com}$ (respectively, $Aexp$ for $T_{\Sigma,Aexp}$ and $Bexp$ for $T_{\Sigma,Bexp}$).

Then, we have inference rules such as:

$$\frac{}{skip \in Com} \qquad \frac{skip \in Com \quad x := a \in Com}{skip\,;\, x := a \in Com}$$

We shall often exploit well-founded relations over terms of a signature.

Example 3.17 (Terms and subterms)
The programs we consider are (well-formed) terms over a suitable signature Σ (possibly many-sorted, with S the set of sorts). It is useful to define a well-founded containment relation between a term and its subterms. (We will exploit this relation when dealing with structural induction in Section 3.1.5). For any n-ary function symbol $f \in \Sigma_n$ and terms $t_1,\dots,t_n$, we let:

$$t_i \prec f(t_1,\dots,t_n) \qquad i = 1,\dots,n$$

The idea is that a term t_i precedes (according to $\prec$, i.e. it is less than) any term that contains it as a subterm (e.g. as an argument).

As a concrete example, let us consider the signature Σ with $\Sigma_0 = \{c\}$ and $\Sigma_2 = \{f\}$. Then, we have, e.g.:

$$c \prec f(c,c) \prec f(f(c,c),c) \prec f(f(f(c,c),c),f(c,c))$$

If we look at terms as trees (function symbols as nodes with one children for each argument and constants as leaves), then we can observe that whenever $s \prec t$ the depth of s is strictly less than the depth of t. Therefore any descending chain is finite (the length is at most the depth of the first term of the chain). Moreover, in the particular case above, c is the only constant and therefore the only minimal element.

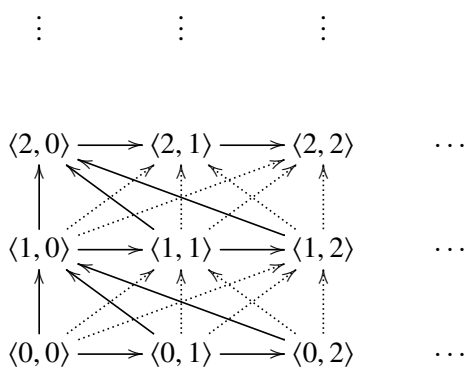


Figure 3.2.: Graph of the lexicographic order relation over pairs of natural numbers.

Example 3.18 (Lexicographic order)
A quite common (well-founded) relation is the so-called lexicographic order. The idea is to have elements that are strings over a given ordered alphabet and to compare them symbol-by-symbol, from the leftmost to the rightmost: as soon as we find a symbol in one string that precedes the symbol in the same position of the other string, then we assume that the former string precedes the latter (independently from the remaining symbols of the two strings).

As a concrete example, let us consider the set of all pairs $\langle n, m\rangle$ of natural numbers. The lexicographic order relation is defined as (see Figure 3.2):

- $\forall n, m, m'.\ (\langle n, m\rangle \prec \langle n + 1, m'\rangle)$
- $\forall n, m.\ (\langle n, m\rangle \prec \langle n, m + 1\rangle)$

Note that the relation has no cycle and any descending chain is bound by the only minimal element $\langle 0, 0\rangle$.
For example, we have:

$$\langle 5, 1\rangle \succ \langle 4, 25\rangle \succ \langle 3, 100\rangle \succ \langle 3, 14\rangle \succ \langle 2, 1\rangle \succ \langle 1, 1000\rangle \succ \langle 0, 0\rangle$$

It is worth to note that any element $\langle n, m\rangle$ with $n \geq 1$ is preceded by infinitely many elements (e.g., $\forall k.\ \langle 0, k\rangle \prec \langle 1, 0\rangle$) and it can be the first element of infinitely many (finite) descending chains (of unbounded length).

Still, given any nonempty set $Q \subseteq \omega \times \omega$, it is easy to find a minimal element $m \in Q$, namely such that $\forall b \prec m.\ b \notin Q$. In fact, we can just take $m = \langle m_1, m_2\rangle$, where m_1 is the minimum (w.r.t. the usual less-than relation over natural numbers) of the set $Q_1 = \{n_1 | \langle n_1, n_2\rangle \in Q\}$ and m_2 is the minimum of the set $Q_2 = \{n_2 | \langle m_1, n_2\rangle \in Q\}$. Note that Q_1 is nonempty because Q is such by hypothesis, and Q_2 is nonempty because $m_1 \in Q_1$ and therefore there must exists at least one pair $\langle m_1, n_2\rangle \in Q$ for some n_2. Thus

$$\langle m_1 = min\{n_1 | \langle n_1, n_2\rangle \in Q\}, min\{n_2 | \langle m_1, n_2\rangle \in Q\}\rangle$$

is a (the only) minimal element of Q. By Lemma 3.12 the relation is well-founded.

Example 3.19 (A counterample: Integer numbers)
The usual "strictly less than" relation $<$ over the set of integer numbers $\mathbb{N}$ is not well-founded. *In fact it is immediate to define infinite descending chains, such as:*

$$0 > -1 > -2 > -3 > \ldots$$

3.1.2. Noether Induction

Theorem 3.20
Let $\prec$ be a well-founded relation over the set A and let P be a unary predicate over A. Then:

$$\frac{\forall a \in A.\ (\forall b \prec a.\ P(b)) \Rightarrow P(a)}{\forall a \in A.P(a)}$$

Proof. We show that the inference rules is valid. We proceed by contradiction by assuming $\neg(\forall a \in A.P(a))$, i.e., that $\exists a \in A.\ \neg P(a)$. Let us consider the nonempty set $Q = \{\ a \in A\ \mid\ \neg P(a)\ \}$ of all those elements a in A for which $P(a)$ is false. Since $\prec$ is well-founded, we know by Lemma 3.12 that there is a minimal element $m \in Q$. Obviously $\neg P(m)$ (otherwise m cannot be in Q). Since m is minimal in Q, then $\forall b \prec m.b \notin Q$, i.e., $\forall b \prec m.P(b)$. But this leads to a contradiction, because by hypothesis we have $\forall a \in A.(\forall b \prec a.P(b)) \rightarrow P(a)$ and instead the predicate $(\forall b \prec m.P(b)) \rightarrow P(m)$ is false. Therefore Q must be empty and $\forall a \in A.P(a)$ must hold. □

We observe that if $\forall a.P(a)$ then $(\forall b \prec a.\ P(b)) \rightarrow P(a)$ is true for any a because the premise $(\forall b \prec a.\ P(b))$ is not relevant (the conclusion of the implication is true).

3.1.3. Weak Mathematical Induction

The principle of weak mathematical induction is a special case of Noether induction that is frequently used to prove formulas over the set on natural numbers: we take

$$A = \omega \qquad n \prec m \Leftrightarrow m = n + 1$$

Theorem 3.21 (Weak mathematical induction)

$$\frac{P(0) \quad \forall n \in \omega.(P(n) \Rightarrow P(n+1))}{\forall n \in \omega.P(n)}$$

In other words, to prove that $P(n)$ holds for any $n \in \omega$ we can just prove that:

- $P(0)$ holds (base case), and
- that, given a generic $n \in \omega$, $P(n+1)$ holds whenever $P(n)$ holds (inductive case).

The principle is helpful, because it allows us to exploit the hypothesis $P(n)$ when proving $P(n+1)$.

3.1.4. Strong Mathematical Induction

The principle of strong mathematical induction extends the weak one by strengthening the hypotheses under which $P(n+1)$ is proved to hold. We take:

$$n \prec m \Leftrightarrow \exists k.\ m = n + k + 1$$

Theorem 3.22 (Strong mathematical induction)

$$\frac{P(0) \quad \forall n \in \omega.(\forall i \leq n.P(i)) \Rightarrow P(n+1)}{\forall n \in \omega.P(n)}$$

In other words, to prove that $P(n)$ holds for any $n \in \omega$ we can just prove that:

- $P(0)$ holds, and
- that, given a generic $n \in \omega$, $P(n+1)$ holds whenever $P(i)$ holds for all $i = 0, ..., n$.

The principle is helpful, because it allows us to exploit the hypothesis $P(0) \wedge P(1) \wedge ... \wedge P(n)$ when proving $P(n+1)$.

3.1.5. Structural Induction

The principle of structural induction is a special instance of Noether induction for proving properties over the set of terms generated by a given signature. Here, the order relation binds a term to its subterms.

Structural induction takes T_Σ as set of elements and subterm-term relation as well-founded relation:

$$t_i < f(t_i, \ldots, t_n) \quad i = 1, \ldots, n$$

Definition 3.23 (Structural induction)

$$\frac{\forall t \in T_\Sigma.\ (\forall t' < t.\ P(t')) \Rightarrow P(t)}{\forall t \in T_\Sigma.\ P(t)}$$

By exploiting the definition of the well-founded subterm relation, we can expand the above principle as the rule

$$\frac{\forall f \in \Sigma_{s_1 \ldots s_n, s}. \forall t_i \in T_{\Sigma, s_i}\ i = 1, \ldots, n.\ (P(t_1) \wedge \ldots \wedge P(t_n)) \Rightarrow P(f(t_1, \ldots, t_n))}{\forall t \in T_\Sigma.\ P(t)}$$

An easy link can be established w.r.t. mathematical induction by taking a unique sort, a constant 0 and a unary operation *succ* (i.e., $\Sigma = \Sigma_0 \cup \Sigma_1$ with $\Sigma_0 = \{0\}$ and $\Sigma_1 = \{succ\}$). Then, the structural induction rule would become:

$$\frac{P(0) \quad \forall t.\ (\ P(t) \Rightarrow P(succ(t))\ )}{\forall t.\ P(t)}$$

Example 3.24
Let us consider the grammar of IMP arithmetic expressions:

$$a \quad ::= \quad n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

How do we exploit structural induction to prove that a property $P(\cdot)$ holds for all arithmetic expressions a? (Namely, we want to prove that $\forall a \in Aexp.\ P(a)$.) The structural induction rule is:

$$\frac{\begin{array}{c}\forall n.\ P(n) \qquad \forall x.\ P(x) \qquad \forall a_0, a_1.\ (P(a_0) \wedge P(a_1) \Rightarrow P(a_0 + a_1)) \\ \forall a_0, a_1.\ (P(a_0) \wedge P(a_1) \Rightarrow P(a_0 - a_1)) \qquad \forall a_0, a_1.\ (P(a_0) \wedge P(a_1) \Rightarrow P(a_0 \times a_1))\end{array}}{\forall a.\ P(a)}$$

Essentially, to prove that $\forall a \in Aexp.\ P(a)$, we just need to show that the property holds for any production, i.e. we need to prove that all of the following hold:

- *$P(n)$ holds for any integer n*
- *$P(x)$ holds for any identifier x*
- *$P(a_0 + a_1)$ holds whenever both $P(a_0)$ and $P(a_1)$ hold*
- *$P(a_0 - a_1)$ holds whenever both $P(a_0)$ and $P(a_1)$ hold*
- *$P(a_0 \times a_1)$ holds whenever both $P(a_0)$ and $P(a_1)$ hold*

Example 3.25 (Structural induction over arithmetic expressions)
Let us consider the case of arithmetic expressions seen above and prove that the evaluation of expressions always terminates:

$$\forall a \in Aexp, \forall \sigma \in \Sigma, \exists m \in \omega, \langle a, \sigma \rangle \rightarrow m$$

In this case we have $P(a) = \forall \sigma \in \Sigma, \exists m \in \omega, \langle a, \sigma \rangle \to m$. Therefore we need to prove that

- *$P(n) = \forall \sigma \in \Sigma, \exists m \in \omega, \langle n, \sigma \rangle \to m$, for any integer n. Trivially, by applying rule (num) we take $m = n$ and we are done.*
- *$P(x) = \forall \sigma \in \Sigma, \exists m \in \omega, \langle x, \sigma \rangle \to m$, for any location x. Trivially, by applying rule (ide) we take $m = \sigma(x)$ and we are done.*
- *$P(a_0) \wedge P(a_1) \Rightarrow P(a_0 + a_1)$ for any arithmetic expressions a_0 and a_1. We assume*

  $$P(a_0) = \forall \sigma \in \Sigma, \exists m_0 \in \omega, \langle a_0, \sigma \rangle \to m_0$$

  and

  $$P(a_1) = \forall \sigma \in \Sigma, \exists m_1 \in \omega, \langle a_1, \sigma \rangle \to m_1.$$

  We want to prove that

  $$P(a_0 + a_1) = \forall \sigma \in \Sigma, \exists m \in \omega, \langle a_0 + a_1, \sigma \rangle \to m.$$

  By applying rule (sum) we can take $m = m_0 + m_1$ if we prove that $\langle a_0, \sigma \rangle \to m_0$ and $\langle a_1, \sigma \rangle \to m_1$. But by inductive hypothesis we know that such m_0 and m_1 exist and we are done.
- *$P(a_0) \wedge P(a_1) \Rightarrow P(a_0 - a_1)$ for any arithmetic expressions a_0 and a_1. The proof is analogous to the previous case and thus omitted.*
- *$P(a_0) \wedge P(a_1) \Rightarrow P(a_0 \times a_1)$ for any arithmetic expressions a_0 and a_1. The proof is analogous to the previous case and thus omitted.*

Example 3.26 (Structural induction over arithmetic expressions)
Let us consider the case of arithmetic expressions seen above and prove that the evaluation of expressions is deterministic:

$$\forall a \in Aexp, \sigma \in \Sigma, m \in \omega, m' \in \omega. \langle a, \sigma \rangle \to m \wedge \langle a, \sigma \rangle \to m' \Rightarrow m = m'$$

In other words, we want to show that given any arithmetic expression a and any memory σ the evaluation of a in σ will always return one and only one value. We let

$$P(a) = \forall \sigma \in \Sigma, m \in \omega, m' \in \omega. \langle a, \sigma \rangle \to m \wedge \langle a, \sigma \rangle \to m' \Rightarrow m = m'$$

We proceed by structural induction.

($a \equiv n$): *there is only one rule that can be used to evaluate an integer number, and it always return the same value. Therefore $m = m'$.*

($a \equiv x$): *again, there is only one rule that can be applied, whose outcome depends on σ. Since σ is the same in both cases, $m = \sigma(x) = m'$.*

($a \equiv a_0 + a_1$): *we have $m = m_0 + m_1$ and $m' = m'_0 + m'_1$ for suitable m_0, m_1, m'_0, m'_1 such that $\langle a_0, \sigma \rangle \to m_0$, $\langle a_1, \sigma \rangle \to m_1$, $\langle a_0, \sigma \rangle \to m'_0$ and $\langle a_1, \sigma \rangle \to m_1$. By hypothesis for structural induction we can assume that $m_0 = m'_0$ (because $P(a_0)$ holds) and $m_1 = m'_1$ (because $P(a_1)$ holds). Thus, $m = m_0 + m_1 = m'_0 + m'_1 = m'$ and thus $P(a_0 + a_1)$ holds.*

The cases for $a \equiv a_0 - a_1$ and $a \equiv a_0 \times a_1$ follow exactly the same pattern as $a \equiv a_0 + a_1$.

3.1.6. Induction on Derivations

See Definitions 1.1 and 1.5 for the notion of inference rule and of derivation.

We can define an induction principle over the set of derivations.

Definition 3.27 (Immediate (sub-)derivation)
We say that d' is an immediate sub-derivation *of d, or simply a sub derivation of d, written $d' \prec d$, if and only if d has the form $(\{d_1, ..., d_n\} \,/\, y)$ with $d_1 \Vdash_R x_1, ..., d_n \Vdash_R x_n$ and $(\{x_1, ..., x_n\} \,/\, y) \in R$ (i.e. $d \Vdash_R y$) and $d' = d_i$ for some $1 \leq i \leq n$.*

Example 3.28 (Immediate (sub-)derivation)
Let us consider the derivation

$$\dfrac{\dfrac{}{\langle 1, \sigma\rangle \to 1}\ num \qquad \dfrac{}{\langle 2, \sigma\rangle \to 2}\ num}{\langle 1 + 2, \sigma\rangle \to 1 + 2 = 3}\ sum$$

the two derivations employing axiom (num) are immediate sub-derivations of the derivation that exploits rule (sum).

We can derive the notion of closed sub-derivations out of immediate ones.

Definition 3.29 (Proper sub-derivation)
We say that d' is a closed sub-derivation *of d if and only if $d' \prec^+ d$.*

Note that both $\prec$ and $\prec^+$ are well-founded and they can be used in proofs by induction.

For example, the induction principle based on immediate sub-derivation can be phrased as follows: Let R be a set of inference rules and D the set of derivations defined on R, then:

$$\dfrac{\forall \{x_1, \ldots, x_n\}/y \in R.\ (\forall d_i \Vdash_R x_i.\ P(d_1) \wedge \ldots \wedge P(d_n)) \Rightarrow P(\{d_1, \ldots, d_n\}/y)}{\forall d \in D.\ P(d)}$$

(Note that $d_1, \ldots, d_n$ are derivation for $x_1, \ldots, x_n$).

3.1.7. Rule Induction

The last kind of induction principle we shall consider applies to sets of elements that are defined by means of inference rules: we have a set of inference rules that establish which elements belong to the set (i.e are theorems) and we need to prove that the application of any such rule will not compromise the validity of the predicate we want to prove.

Formally, a rule has the form $(\varnothing/y)$ if it is an axiom, or $(\{x_1, \ldots, x_n\}/y)$ otherwise. Given a set R of such rules, the *set of theorems of R* is defined as

$$I_R = \{x \mid \Vdash_R x\}$$

The rule induction principle aims to show that the property P holds for all elements of I_R, which amounts to show that:

- for any axiom $(\varnothing/y)$, we have that $P(y)$ holds;
- for any other rule $(\{x_1, \ldots, x_n\}/y)$ we have that $(\forall 1 \leq i \leq n. x_i \in I_R \wedge P(x_i)) \Rightarrow P(y)$.

$$\dfrac{\forall (X/y) \in R \quad (X \subseteq I_R \quad \forall x \in X.\ P(x)) \Longrightarrow P(y)}{\forall x \in I_R. P(x)}$$

Note that in some cases we will use a simpler but less powerful rule

$$\frac{\forall (X/y) \in R \quad (\forall x \in X \cdot P(x)) \Longrightarrow P(y)}{\forall x \in I_R . P(x)}$$

In fact, if the latter applies, also the former does, since the implication in the premise must be proved in fewer cases: only for rules X/y such that all the formulas in X are theorems. However, usually it is difficult to take advantage of there restriction.

Example 3.30 (Proof by rule induction)
We have seen in Example 3.26 that structural induction can be conveniently used to prove that the evaluation of arithmetic expressions is deterministic. Formally, we were proving the predicate $P(\cdot)$ over arithmetic expressions defined as

$$P(a) \stackrel{\text{def}}{=} \forall \sigma . \forall m, m' . \langle a, \sigma \rangle \to m \wedge \langle a, \sigma \rangle \to m' \Longrightarrow m = m'$$

While the case of boolean expressions is completely analogous, for commands we cannot use the same proof strategy, because structural induction cannot deal with the rule (whtt). In this example, we show that rule induction provides a convenient strategy to solve the problem.

Let us consider the following predicate over "theorems":

$$P(\langle c, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1 . \langle c, \sigma \rangle \to \sigma_1 \Longrightarrow \sigma' = \sigma_1$$

(rule skip): *we want to show that*

$$P(\langle \mathbf{skip}, \sigma \rangle \to \sigma) \stackrel{\text{def}}{=} \forall \sigma_1 . \langle \mathbf{skip}, \sigma \rangle \to \sigma_1 \Longrightarrow \sigma_1 = \sigma$$

which is obvious because there is only one rule applicable to **skip***:*

$$\langle \mathbf{skip}, \sigma \rangle \to \sigma_1 \quad \nwarrow_{\sigma_1 = \sigma} \quad \square$$

(rule assign): *assuming*

$$\langle a, \sigma \rangle \to m$$

we want to show that

$$P(\langle x := a, \sigma \rangle \to \sigma[{}^{m}/_{x}]) \stackrel{\text{def}}{=} \forall \sigma_1 . \langle x := a, \sigma \rangle \to \sigma_1 \Longrightarrow \sigma_1 = \sigma[{}^{m}/_{x}]$$

Let us assume the premise of the implication we want to prove, and let us proceed goal oriented. We have:

$$\langle x := a, \sigma \rangle \to \sigma_1 \quad \nwarrow_{\sigma_1 = \sigma[{}^{m'}/_{x}]} \quad \langle a, \sigma \rangle \to m'$$

But we know that the evaluation of arithmetic expressions is deterministic and therefore $m' = m$ and $\sigma_1 = \sigma[{}^{m}/_{x}]$.

(rule seq): *assuming $P(\langle c_0, \sigma \rangle \to \sigma'')$ and $P(\langle c_1, \sigma'' \rangle \to \sigma')$ we want to show that*

$$P(\langle c_0; c_1, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1 . \langle c_0; c_1, \sigma \rangle \to \sigma_1 \Longrightarrow \sigma_1 = \sigma'$$

We have:

$$\langle c_0; c_1, \sigma \rangle \to \sigma_1 \quad \nwarrow \quad \langle c_0, \sigma \rangle \to \sigma_1'' \quad \langle c_1, \sigma_1'' \rangle \to \sigma_1$$

But now we can apply the first inductive hypotheses:

$$P(\langle c_0, \sigma \rangle \to \sigma'') \stackrel{\text{def}}{=} \forall \sigma_1'' . \langle c_0, \sigma \rangle \to \sigma_1'' \Longrightarrow \sigma_1'' = \sigma''$$

to conclude that $\sigma_1'' = \sigma''$*, which together with the second inductive hypothesis*

$$P(\langle c_1, \sigma'' \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\, \langle c_1, \sigma'' \rangle \to \sigma_1 \Longrightarrow \sigma_1 = \sigma'$$

allow us to conclude that $\sigma_1 = \sigma'$*.*

(rule ifft): *assuming* $\langle b, \sigma \rangle \to \mathbf{true}$ *and* $P(\langle c_0, \sigma \rangle \to \sigma')$ *we want to show that*

$$P(\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\, \langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma_1 \Longrightarrow \sigma_1 = \sigma'$$

Since $\langle b, \sigma \rangle \to \mathbf{true}$ *and the evaluation of boolean expressions is deterministic, we have:*

$$\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma_1 \nwarrow \langle c_0, \sigma \rangle \to \sigma_1$$

But then, exploiting the inductive hypothesis

$$P(\langle c_0, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\, \langle c_0, \sigma \rangle \to \sigma_1 \Longrightarrow \sigma_1 = \sigma'$$

we can conclude that $\sigma_1 = \sigma'$*.*

(rule ifff): *omitted (it is analogous to the previous case).*

(rule whileff): *assuming* $\langle b, \sigma \rangle \to \mathbf{false}$ *we want to show that*

$$P(\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma) \stackrel{\text{def}}{=} \forall \sigma_1.\, \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma_1 \Longrightarrow \sigma_1 = \sigma$$

Since $\langle b, \sigma \rangle \to \mathbf{false}$ *and the evaluation of boolean expressions is deterministic, we have:*

$$\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma_1 \nwarrow_{\sigma_1 = \sigma} \Box$$

(rule whilett): *assuming* $\langle b, \sigma \rangle \to \mathbf{true}$*,* $P(\langle c, \sigma \rangle \to \sigma'')$ *and* $P(\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma'' \rangle \to \sigma')$ *we want to show that*

$$P(\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\, \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma_1 \Longrightarrow \sigma_1 = \sigma'$$

Since $\langle b, \sigma \rangle \to \mathbf{true}$ *and the evaluation of boolean expressions is deterministic, we have:*

$$\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma_1 \nwarrow \quad \langle c, \sigma \rangle \to \sigma_1'' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma_1'' \rangle \to \sigma_1$$

But now we can apply the first inductive hypotheses:

$$P(\langle c, \sigma \rangle \to \sigma'') \stackrel{\text{def}}{=} \forall \sigma_1''.\, \langle c, \sigma \rangle \to \sigma_1'' \Longrightarrow \sigma_1'' = \sigma''$$

to conclude that $\sigma_1'' = \sigma''$*, which together with the second inductive hypothesis*

$$P(\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma'' \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\, \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma'' \rangle \to \sigma_1 \Longrightarrow \sigma_1 = \sigma'$$

allow us to conclude that $\sigma_1 = \sigma'$*.*

3.2. Well-founded Recursion

We conclude this chapter by presenting the concept of well-founded recursion. A recursive definition of a function f is *well-founded* when the recursive calls to f take as arguments values that are smaller w.r.t. the ones taken by the defined function (according to a suitable well-founded relation). The functions defined on natural numbers according to the principle of well-founded recursion are called *primitive recursive functions*.

Example 3.31 (Well-founded recursion)
Let us consider the Peano formula that defines the product of natural numbers

$$p(0, y) = 0$$
$$p(x + 1, y) = y + p(x, y)$$

Let us write the definition in a slightly different way

$$p_y(0) = 0$$
$$p_y(x + 1) = y + p_y(x)$$

It is immediate to see that $p_y(\cdot)$ is primitive recursive for every y.

Let us make the intuition more precise.

Definition 3.32 (Set of predecessors)
Given a well founded relation $\prec \subseteq B \times B$, the set of predecessors *of a set $I \subseteq B$ is the set*

$$\prec^{-1} I = \{\, b \in B \mid \exists b' \in I.\ b \prec b' \,\}$$

We recall that for $B' \subseteq B$ and $f : B \to C$, we denote by $f \restriction B'$ the restriction of f to values in B', i.e., $f \restriction B' : B' \to C$ and $(f \restriction B')(b) = f(b)$ for any $b \in B'$.

Theorem 3.33 (Well-founded recursion)
Let $(B, \prec)$ a well-founded relation over B. Let us consider a function F with $F(b, h) \in C$, where

- $b \in B$
- $h : \prec^{-1} \{b\} \to C$

Then, there exists one and only one function $f : B \to C$ which satisfies the equation

$$\forall b \in B.\ f(b) = F(b, f \restriction \prec^{-1} \{b\})$$

In other words, if we (recursively) define f over any b only in terms of the predecessors of b, then f is uniquely determined on all b. Notice that F has a *dependent* type, since the type of its second argument depends on the value of its first argument.

In the following chapters we will exploit partial orders fix-point theory to define the semantics of recursively defined functions. Well-founded recursion gives a simpler method, which however works only in the well-founded case.

Example 3.34 (Primitive recursion)
Let us recast the Peano formula seen above (Example 3.31) to the formal scheme of primitive recursion.

- $p_y(0) = F_y(0, p_y \restriction \varnothing) = 0$
- $p_y(x + 1) = F_y(x + 1, p_y \restriction \prec^{-1} \{x + 1\}) = y + p_y(x)$

Example 3.35 (Structural recursion)
Let us consider the signature Σ for binary trees $B = T_\Sigma$, where $\Sigma_0 = \{0, 1, ...\}$ and $\Sigma_2 = \mathsf{cons}$. Take the

well-founded relation $x_i \prec cons(x_1, x_2)$, $i = 1, 2$. Let $C = \omega$.

We want to compute the sum of the elements in the leaves of a binary tree. In Lisp-like notation:

$$sum(x) = \textbf{if } atom(x) \textbf{ then } x \textbf{ else } sum(car(x)) + sum(cdr(x))$$

where $atom(x)$ returns true if x is a leaf; $car(x)$ denotes the left subtree of x; $cdr(x)$ the right subtree of x and $cons(x, y)$ is the constructor for building a tree out of its left and right subtree. The same function defined in the structural recursion style is

$$\begin{cases} sum(n) = n \\ sum(cons(x, y)) = sum(x) + sum(y) \end{cases}$$

or more formally

$$\begin{cases} F(n, sum \restriction \emptyset) = n \\ F(cons(x, y), sum \restriction \{x, y\}) = sum(x) + sum(y) \end{cases}$$

Example 3.36 (Ackermann function)
The Ackermann function $ack(z, x, y) = ack_y(z, x)$ is defined by well-founded recursion (exploiting the lexicographic order over pair of natural numbers) by letting

$$\begin{cases} ack(\ 0\ ,\ 0\ ,\ y\ ) = y \\ ack(\ 0\ ,\ x+1\ ,\ y\ ) = ack(0, x, y) + 1 \\ ack(\ 1\ ,\ 0\ ,\ y\ ) = 0 \\ ack(\ z+2\ ,\ 0\ ,\ y\ ) = 1 \\ ack(\ z+1\ ,\ x+1\ ,\ y\ ) = ack(z, ack(z+1, x, y), y) \end{cases}$$

We have

$$\begin{cases} ack(0, 0, y) = y \\ ack(0, x+1, y) = ack(0, x, y) + 1 \end{cases} \implies ack(0, x, y) = y + x$$

$$\begin{cases} ack(1, 0, y) = 0 \\ ack(1, x+1, y) = ack(0, ack(1, x, y), y) = ack(1, x, y) + y \end{cases} \implies ack(1, x, y) = yx$$

$$\begin{cases} ack(2, 0, y) = 1 \\ ack(2, x+1, y) = ack(1, ack(2, x, y), y) = ack(2, x, y)y \end{cases} \implies ack(2, x, y) = y^x$$

and so on.

4. Partial Orders and Fixpoints

4.1. Orderings and Continuous Functions

This chapter is devoted to the introduction of the mathematical foundations of the denotational semantics of computer languages.

As we have seen, the operational semantics gives us a very concrete semantics, since the inference rules describe step by step the bare essential operations on the state required to reach the final state of computation. Unlike the operational semantics, the denotational one provides a more abstract view. Indeed, the denotational semantics gives us directly the meaning of the constructs of the language as particular functions over domains. Domains are sets whose structure will ensure the correctness of the constructions of the semantics.

As we will see, one of the most attractive features of the denotational semantics is that it is compositional, namely the meaning of a construct is given by combining the meanings of its components. The compositional property of denotational semantics is obtained by defining the semantics by structural recursion. Obviously there are particular issues in defining the "while" construct of the language, since the semantics of this construct, as we saw in the previous chapters, seems to be recursive. General recursion is not allowed in structural recursion, which allows only the use of sub-terms. The solution to this problem is given by solving equations of the type $f(x) = x$, namely by finding the fixpoint(s) of f. So on the one hand we would like to ensure that each recursive function that we will consider has at least a fixpoint. Therefore we will restrict our study to a particular class of functions: continuous functions. On the other hand, the aim of the theory we will develop, called domain theory, will be to identify one solution among the existing ones, and to provide an approximation method for it.

4.1.1. Orderings

We introduce the general theory of partial orders which will bring us to the concept of domain.

Definition 4.1 (Partial order)
A partial order *is a pair* $(P, \sqsubseteq)$ *where* P *is a set and* $\sqsubseteq \subseteq P \times P$ *is a relation on* P *(i.e. it is a set of pairs of elements of* P*) which is:*

- *reflexive:* $\forall p \in P.\ p \sqsubseteq p$
- *antisymmetric:* $\forall p, q \in P.\ p \sqsubseteq q \wedge q \sqsubseteq p \implies p = q$
- *transitive:* $\forall p, q, r \in P.\ p \sqsubseteq q \wedge q \sqsubseteq r \implies p \sqsubseteq r$

We call P *a* poset *(partially ordered set).*

Example 4.2 (Powerset)
Let $(2^S, \subseteq)$ *be a relation on the powerset of a set* S*. It is easy to see that* $(2^S, \subseteq)$ *is a partial order.*

- *reflexivity:* $\forall s \subseteq S.\ s \subseteq s$
- *antisymmetry:*$\forall s_1, s_2 \subseteq S.\ s_1 \subseteq s_2 \wedge s_2 \subseteq s_1 \implies s_1 = s_2$
- *transitivity:* $\forall s_1, s_2, s_3 \subseteq S.\ s_1 \subseteq s_2 \subseteq s_3 \implies s_1 \subseteq s_3$

Actually, partial orders are a generalization of the concept of powerset ordered by inclusion. Thus we should not be surprised by this result.

Definition 4.3 (Total order)
Let $(P, \sqsubseteq)$ *be a partial order such that:*

$$\forall x, y \in P.\ x \sqsubseteq y \ \vee \ y \sqsubseteq x$$

we call $(P, \sqsubseteq)$ total order.

Theorem 4.4 (Subsets of an order)
Let $(A, \sqsubseteq)$ *be a partial order and let* $B \subseteq A$. *Then* $(B, \sqsubseteq')$ *is a partial order, with* $\sqsubseteq' = \sqsubseteq \cap (B \times B)$. *Similarly, if* $(A, \sqsubseteq)$ *is a total order then* $(B, \sqsubseteq')$ *is a total order.*

Let us see some examples that will be very useful to understand the concepts of partial and total orders.

Example 4.5 (Natural Numbers)
Let $(\omega, \leq)$ *be the usual ordering on the set of natural numbers,* $(\omega, \leq)$ *is a total order.*

- *reflexivity:* $\forall n \in \omega . n \leq n$
- *antisymmetric:*$\forall n, m \in \omega . n \leq m \wedge m \leq n \Longrightarrow m = n$
- *transitivity:* $\forall n, m, z \in \omega . n \leq m \wedge m \leq z \Longrightarrow n \leq z$
- *total:*$\forall n, m \in \omega . n \leq m \vee m \leq n$

Example 4.6 (Discrete order)
Let $(P, \sqsubseteq)$ *be a partial order defined as follows:*

$$\forall p \in P . p \sqsubseteq p$$

We call $(P, \sqsubseteq)$ *a* discrete order. *Obviously* $(P, \sqsubseteq)$ *is a partial order.*

Example 4.7 (Flat order)
Let $(P, \sqsubseteq)$ *be a partial order defined as follows:*

- $\exists \bot \in P . \forall p \in P . \bot \sqsubseteq p$
- $\forall p \in P . p \sqsubseteq p$

We call $(P, \sqsubseteq)$ *a* flat order.

4.1.2. Hasse Diagrams

The aim of this section is to provide a tool that allows us to represent orders in a comfortable way.

First of all we could think to use graphs to represent an order. In this framework each element of the ordering is represented by a node of the graph and the order relation by the arrows (i.e. we would have an arrow from a to b if and only if $a \sqsubseteq b$).

This notation is not very comfortable, indeed we repeat many times the same information. For example in the usual natural numbers order we would have for each node $n + 1$ incoming arrows and infinite outgoing arrows, where n is the natural number which labels the node.

We need a more compact notation, which takes into account the redundant information. This notation is represented by the Hasse diagrams.

Definition 4.8 (Hasse Diagram)
Given a poset $(A, \sqsubseteq)$, *let R be a binary relation such that:*

$$\frac{x \sqsubseteq y \quad y \sqsubseteq z \quad x \neq y \neq z}{xRz} \quad , \quad \frac{\varnothing}{xRx}$$

We call Hasse diagram *the relation H defined as:*

$$H = \sqsubseteq - R$$

The Hasse diagram omits the information deducible by transitivity and reflexivity. A simple example of Hasse diagram is in Fig. 4.1.

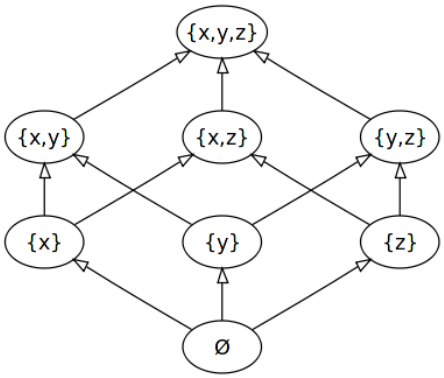


Figure 4.1.: Hasse diagram for the powerset over $\{x, y, z\}$ ordered by inclusion

To ensure that all the needed information is contained in the Hasse diagram we will use the following theorem.

Theorem 4.9 (Order relation, Hasse diagram Equivalence)
Let $(P, \sqsubseteq)$ a partial order with P finite set. Then the transitive and reflexive closure of its Hasse diagram is equal to $\sqsubseteq$:

$$\frac{\varnothing}{xH^*x} \qquad \frac{xH^*y \wedge yHz}{xH^*z}$$

We have:

$$H^* = \sqsubseteq$$

Note that the rule

$$\frac{yHz}{yH^*z}$$

is subsumed by the the fact that in the second rule one can use yH^*y (guaranteed by the first rule) and yHz as premises.

$$\frac{\dfrac{\varnothing}{yH^*y} \quad \wedge yHz}{yH^*z}$$

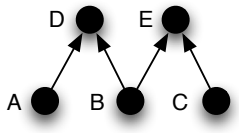
The above theorem only allows to represent finite orders.

Example 4.10 (Infinite order)
Let us see that the Hasse diagrams does not work with infinite orders. Let $(\omega \cup \{\infty\}, \leq)$ be the usual order on natural numbers extended by placing $n \leq \infty$ and $\infty \leq \infty$. The Hasse diagram eliminates all the arcs between each natural number and ∞. Now using the transitive and reflexive closure we would like to get back the order. Using the inference rules we obtain the usual order on natural numbers without any relation between ∞ and the natural numbers (recall that we only allow finite proofs).

In order to study the topology of the partial orders we introduce the following notions:

Definition 4.11 (Least element)
An element $m \in S$ is a least *element of $(S, \sqsubseteq)$ if: $\forall s \in S.\ m \sqsubseteq s$*

Let us consider the following order:



this order has no least element. As we will see the elements A, B and C are minimal since they have no smaller elements in the order.

Theorem 4.12 (Uniqueness of the least element)
Let $(A, \sqsubseteq)$ be a partial order, A has at most one least element.

Proof. Let $a, b \in A$ be both least elements of A, then $a \sqsubseteq b$ and $b \sqsubseteq a$. Now by using the antisymmetric property of A we obtain $a = b$. □

The counterpart of the least element is the concept of greatest element, we can obtain the greatest element as the least element of the reverse order (i.e. $x \sqsubseteq^{-1} y \iff y \sqsubseteq x$).

Definition 4.13 (Minimal element)
$m \in S$ is a minimal element *of $(S, \sqsubseteq)$ if: $\forall s \in S.\ s \sqsubseteq m \Rightarrow s = m$*

As for the least element we have the dual of minimal elements, maximal elements. Note that the definition of minimal and least element (maximal and greatest) are quite different. The least element must be smaller than all the elements of the order. A minimal element, instead, should not have elements smaller than it, no one guarantees that all the elements are in the order relation with a minimal element.

Remark 4.14
There is a difference between a least *and a* minimal *element of a set:*

- *the least element x is the smallest element of a set, i.e. x is such that $\forall a \in A.\ x \sqsubseteq a$.*
- *a minimal element y is just such that no smaller element can be found in the set, i.e. $\forall a \in A.\ a \not\sqsubset y$.*

So the least element of an order is obviously minimal, but a minimal element is not necessarily the least.

Definition 4.15 (Upper bound)
Let $(P, \sqsubseteq)$ be a partial order and $X \subseteq P$ be a subset of P, then $p \in P$ is an upper bound *of X iff*

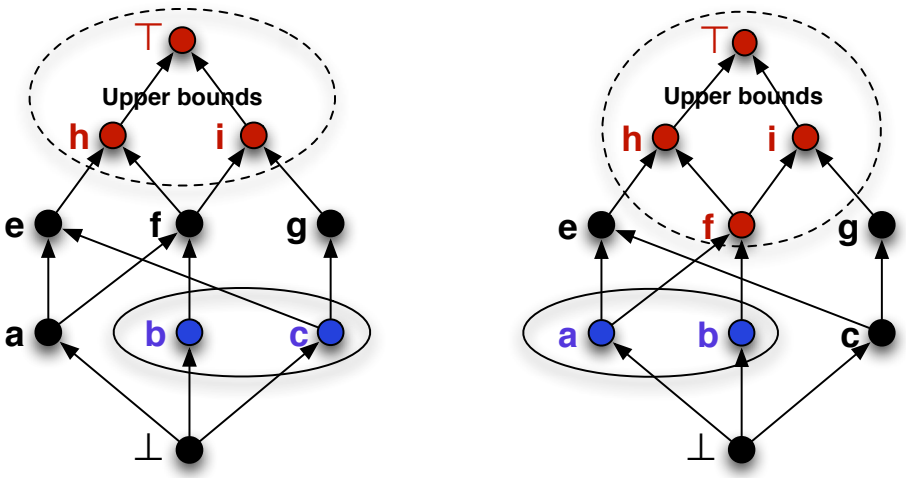
$$\forall q \in X.\ q \sqsubseteq p$$

Note that unlike the maximal element and the greatest element the upper bound does not necessarily belong to the subset.

Definition 4.16 (Least upper bound)
Let $(P, \sqsubseteq)$ be a partial order and $X \subseteq P$ be a subset of P then $p \in P$ is the least upper bound *of X if and only if p is the least element of the upper bounds of P. Formally:*

- *p is an upper bound of X*
- *$\forall q$ upper bound of X then $p \sqsubseteq q$*

and we write $lub(X) = p$.



(a) (b)

Figure 4.2.: Two subsets of a poset, the right one with LUB and the left one without LUB

Now we will clarify the concept of *LUB* with two examples. Let us consider the order represented in figure 4.2 (a). The set of upper bounds of the subset $\{b, c\}$ is the set $\{h, i, \top\}$. This set has no least element (i.e. h and i are not in the order relation) so the set $\{b, c\}$ has no LUB. In figure 4.2 (b) we see that the set of upper bounds of the set $\{a, b\}$ is the set $\{f, h, i, \top\}$. The least element of the latter set is f, which thus is the LUB of $\{a, b\}$.

4.1.3. Chains

One of the main concept in the study of order theory is that of *chain*.

Definition 4.17 (Chain)
Let $(P, \sqsubseteq)$ be a partial order, we call chain *a function $C : \omega \longrightarrow P$, (we will write $C = \{d_i\}_{i\in\omega}$) such that:*

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \ldots$$

i.e. where $\forall i \in \omega.\ d_i = C(i)$, we have:

$$\forall n \in \omega.\ C(n) \sqsubseteq C(n+1)$$

Definition 4.18 (Finite chain)
Let $C : \omega \longrightarrow P$ be a chain such that the codomain (range) of C is a finite set, then we say that C is a finite chain.

Definition 4.19 (Limit of a chain)
Let C be a chain. The LUB of the codomain of C, if it exists, is called the limit of C. *If d is the limit of the chain $C = \{d_i\}_{i\in\omega}$, we will write $d = \bigsqcup_{i\in\omega} d_i$.*

Note that each finite chain has a limit, indeed each finite chain has a finite totally ordered codomain, obviously this set has a LUB (the greatest element of the set).

Definition 4.20 (Depth of a poset)
Let $(P, \sqsubseteq)$ be a poset, we say that $(P, \sqsubseteq)$ has finite depth *if and only if each chain of the poset is finite.*

Lemma 4.21 (Prefix independence of the limit)
Let $n \in \omega$ and let C and C' be two chains such that $C = \{d_i\}_{i\in\omega}$ and $C' = \{d_{n+i}\}_{i\in\omega}$. Then C and C' have the

same limit, if any. This means that we can eliminate a proper prefix from a chain preserving the limit.

Proof. Let us show that the chains have the same set of upper bounds. Obviously if c is an upper bound of C, then c is an upper bound of C', since each element of C' is contained in C. On the other hand if c is an upper bound of C', we have $\forall j \in \omega.\ j \leq n \Rightarrow d_j \sqsubseteq d_n \wedge d_n \sqsubseteq c \Rightarrow d_j \sqsubseteq c$ by transitivity of $\sqsubseteq$ then c is an upper bound of C. Now since C and C' have the same set of upper bound elements, they have the same LUB, if it exists at all. □

4.1.4. Complete Partial Orders

As we said the aim of partial orders and continuous functions is to provide a framework that allows to ensure the correctness of the denotational semantics. Complete partial orders extend the concept of partial orders to support the limit operation on chains, which is a generalization of the countable union operation on a powerset. Limits will have a key role in finding fixpoints.

Definition 4.22 (Complete partial orders)
Let $(P, \sqsubseteq)$ be a partial order, we say that $(P, \sqsubseteq)$ is complete *(CPO) if each chain has a limit (i.e. each chain has a LUB).*

Definition 4.23 (CPO with bottom)
Let $(D, \sqsubseteq)$ be a CPO, we say that $(D, \sqsubseteq)$ is a CPO with bottom *($CPO_\perp$) if it has a least element $\perp$ (called* bottom*).*

Let us see some examples, that will clarify the concept of CPO.

Example 4.24 (Powerset completeness)
Let us consider again the previous example of powerset (ex.4.2), we can now show that the partial order $(2^S, \subseteq)$ is complete.

$$lub(s_0 \subseteq s_1 \subseteq s_2 \ldots) = \{d \mid \exists k.\ d \in s_k\} = \bigcup_{i \in \omega} s_i \in 2^S$$

Example 4.25 (Partial order without completeness)
Now let us take the usual order on natural numbers

$$(\omega, \leq)$$

Obviously all the finite chains have a limit (i.e. the greatest element of the chain). On the other hand infinite chains have no limits (i.e. there is no natural number greater than infinitely many natural numbers). To make the order a CPO all we have to do is to add an element ∞ greater than all the natural numbers. Now each infinite chain has a limit (∞), and the order is a CPO.

Example 4.26 (Partial order without completeness conclusion)
Let us define the partial order $(\omega \cup \{\infty_1, \infty_2\}, \sqsubseteq)$ as follows:

$$\sqsubseteq\restriction \omega = \leq\ ,\ \forall n \in \omega.\ n \sqsubseteq \infty_1 \wedge n \sqsubseteq \infty_2\ ,\ \infty_1 \sqsubseteq \infty_1\ ,\ \infty_2 \sqsubseteq \infty_2$$

Where $\sqsubseteq\restriction \omega$ is the restriction of $\sqsubseteq$ to natural numbers. This partial order is not complete, indeed each infinite chain has two upper bounds (i.e. ∞_1 and ∞_2) which have no least element.

Example 4.27 (Partial functions)
Let us consider a set that we will meet again during the course: the set of partial functions on natural numbers:

$$P = \omega \rightharpoonup \omega$$

Recall that a partial function is a relation f on $\omega \times \omega$ with the functional *property:*

$$\forall n, m, m' \in \omega.\ nfm \wedge nfm' \Longrightarrow m = m'$$

So the set P can be viewed as:

$$P = \{\ f \subseteq \omega \times \omega \mid \forall n, m, m' \in \omega.\ nfm \wedge nfm' \Longrightarrow m = m'\ \}$$

Let us denote by $f(n)\downarrow$ the predicate $\exists m.\ nfm$ (i.e., $f(n)\downarrow$ holds when the function f is defined on n). Now it is easy to define a partial order on P:

$$f \sqsubseteq g \iff (\forall n.\ f(n)\downarrow \Rightarrow\ f(n) = g(n))$$

That is g is greater than f when both are seen as relations and g includes f. This order has the empty relation as bottom element, and each infinite chain has as limit the countable union of the relations of the chain. Next we show that the limits of the infinite chains are not only relations, but are functions of our domain.

Let $R_0 \subseteq R_1 \subseteq R_2 \subseteq \ldots$ be a chain of P, with $\forall i.\ nR_im \wedge nR_im' \Longrightarrow m = m'$. We will show that their union is still a function, that is the relation $\cup_{i\in\omega}R_i$ has the functional property:

$$\forall n, m, m' \in \omega.\ n(\cup_{i\in\omega}R_i)m \wedge n(\cup_{i\in\omega}R_i)m' \Longrightarrow m = m'.$$

Let us assume $n(\cup_{i\in\omega}R_i)m \wedge n(\cup_{i\in\omega}R_i)m'$ so we have $\exists k, k'.\ nR_km \wedge nR_{k'}m'$, we take $k'' = max\{k, k'\}$ then it holds $nR_{k''}m \wedge nR_{k''}m'$ then by hypothesis $m = m'$.

Let us show a second way to define a CPO on the partial functions on natural numbers. We add a bottom element to the co-domain and we consider the following set of total functions:

$$P' = \{\ f \subseteq \omega \times \omega_\perp \mid \forall n, m, m' \in \omega.\ nfm \wedge nfm' \Longrightarrow m = m' \wedge\ \forall n \in \omega.\ \exists m.\ nfm\ \}$$

where we see $(\omega_\perp, \sqsubseteq_{\omega_\perp})$ as the flat order obtained by adding $\perp$ to the discrete order of the natural numbers. We define the following order on P'

$$f \sqsubseteq g \Leftrightarrow \forall x. f(x) \sqsubseteq_{\omega_\perp} g(x)$$

That is, if $f(x) = \perp$ then $g(x)$ can assume any value otherwise $f(x) = g(x)$. The bottom element of the order is the function $\lambda x.\ \perp$, which returns $\perp$ for every value of x. Note that the above ordering is complete, In fact, the limit of a chain obviously exists as a relation, and it is easy to show, analogously to the partial function case, that it is in addition a total function.

Example 4.28 (Limit of a chain of partial functions)
Let $\{f_i\}_{i\in\omega}$ be a chain on P' such that:

$$f_k(n) = \begin{cases} 3 \text{ if } \exists\ m \in \omega.\ n = 2m \wedge n \leq k \\ \perp \text{ otherwise} \end{cases}$$

Let us consider some application of the functions:

$$\begin{array}{lll} f_0(0) = 3 & & \\ f_1(0) = 3 & & \\ f_2(0) = 3 & f_2(2) = 3 & \\ f_3(0) = 3 & f_3(2) = 3 & \\ f_4(0) = 3 & f_4(2) = 3 & f_4(4) = 3 \end{array}$$

This chain has as limit the function that returns 3 *on the even numbers.*

$$f(n) = \begin{cases} 3 \text{ if } \exists\, m \in \omega.\ n = 2m \\ \bot \text{ otherwise} \end{cases}$$

So the limit of an infinite chain is still a partial function.

4.2. Continuity and Fixpoints

4.2.1. Monotone and Continuous Functions

In order to define a class of functions over domains which ensures the existence of their fixpoints we will introduce two general properties of functions: monotonicity and continuity.

Definition 4.29 (monotonicity)
Let f be a function over a CPO $(D, \sqsubseteq)$, we say that $f : D \longrightarrow D$ is monotone *if and only if*

$$\forall d, e \in D.\ d \sqsubseteq e \Rightarrow f(d) \sqsubseteq f(e)$$

We say that a monotone function preserves the order. So if $\{d_i\}_{i\in\omega}$ is a chain on $(D, \sqsubseteq)$ and f is a monotone function then $\{f(d_i)\}_{i\in\omega}$ is a chain on $(D, \sqsubseteq)$.

Example 4.30 (Not monotone function)
Let us define a CPO $(\{a, b, c\}, \sqsubseteq)$ where $\sqsubseteq$ is defined as $b \sqsubseteq a$, $b \sqsubseteq c$ and $x \sqsubseteq x$ for any $x \in \{a, b, c\}$. Now define a function f on $(\{a, b, c\}, \sqsubseteq)$ as follows:

$$f(a) = a \quad f(b) = a \quad f(c) = c$$

This function is not monotone, indeed $b \sqsubseteq c \not\Rightarrow f(b) \sqsubseteq f(c)$ (i.e., $a = f(b)$ and $c = f(b)$ are not related), so the function does not preserve the order.

Definition 4.31 (Continuity)
Let f be a monotone function on a CPO $(D, \sqsubseteq)$, we say that f is a continuous function if and only if for each chain in $(D, \sqsubseteq)$ we have:

$$f(\bigsqcup_{i\in\omega} d_i) = \bigsqcup_{i\in\omega} f(d_i)$$

Note that, as it is the case for most definitions of function continuity, the operations of applying function and taking the limit can be exchanged.

We will say that a continuous function preserves limits.

Example 4.32 (A monotone function which is not continuous)
Let $(\omega \cup \{\infty\}, \leq)$ be a CPO, define a function f:

$$\forall n \in \omega.\ f(n) = 0\ ,\ f(\infty) = 1$$

Let us consider the chain:

$$0 \leq 2 \leq 4 \leq 6 \leq \ldots$$

so we have

$$f(\bigsqcup_{i\in\omega} d_i) = f(\infty) = 1 \quad \neq \quad \bigsqcup_{i\in\omega} f(d_i) = 0$$

then the function does not preserve the limits.

4.2.2. Fixpoints

Now we are ready to study fixpoints of continuous functions.

Definition 4.33 (Fixpoint)
Let f a continuous function over a $CPO_\perp$ $(D, \sqsubseteq)$. An element d of D is a fixpoint *of f if and only if:*

$$f(d) = d$$

Definition 4.34 (Pre-fixpoint)
Let f a continuous function on a $CPO_\perp$ $(D, \sqsubseteq)$. We say that an element d is a pre-fixpoint *if and only if:*

$$f(d) \sqsubseteq d$$

Of course any fixpoint of f is also a pre-fixpoint of f.
We will denote $\text{gfp}(f)$ the greatest fixpoint of f and with $\text{lfp}(f)$ the least fixpoint of f.

4.2.3. Fixpoint Theorem

Let $(D, \sqsubseteq)$ be a complete partial order with bottom ($\text{CPO}_\perp$), $f : D \to D$ be a continuous function and $d \in D$. We denote by $f^n(d)$ the repeated application of f to d for n times, i.e., $f^0(d) = d$ and $f^{n+1}(d) = f^n(d)$. Note that $\{f^n(\perp)\}_{n\in\omega}$ defines a chain in D. In fact, the property $\forall n.\ f^n(\perp) \sqsubseteq f^{n+1}(\perp)$ can be readily proved by mathematical induction on n. For the base case ($n = 0$) we have $f^0(\perp) = \perp \sqsubseteq f^1(\perp) = f(\perp)$, as $\perp$ is the least element of D. For the inductive case, let us assume that the property holds for n (i.e., $f^n(\perp) \sqsubseteq f^{n+1}(\perp)$). We want to prove that $f^{n+1}(\perp) \sqsubseteq f^{n+2}(\perp)$. Since f is monotone and by the inductive hypothesis we have:

$$f^{n+1}(\perp) = f(f^n(\perp)) \sqsubseteq f(f^{n+1}(\perp)) = f^{n+2}(\perp).$$

Since D is complete, then the chain $\{f^n(\perp)\}_{n\in\omega}$ has a limit $\bigsqcup_{n\in\omega} f^n(\perp)$.

Next theorem ensures that the least fixpoint of a continuous function always exists and that it can be found by exploiting the above limit.

Theorem 4.35 (Fixpoint theorem)
Let $f : D \to D$ be a continuous function on the complete partial order with bottom D. Then, let

$$\text{fix}(f) = \bigsqcup_{n\in\omega} f^n(\perp).$$

The element $\text{fix}(f) \in D$ has the following properties:

1. $\text{fix}(f)$ *is a fixpoint of f*

$$f(\text{fix}(f)) = \text{fix}(f)$$

2. $\text{fix}(f)$ *is the least pre-fixpoint of f*

$$f(d) \sqsubseteq d \Rightarrow \text{fix}(f) \sqsubseteq d$$

so $\text{fix}(f)$ is the least fixpoint of f.

Proof. 1. By continuity we will show that $\text{fix}(f)$ is a fixpoint of f:

$$\begin{aligned} f(\text{fix}(f)) &= f(\bigsqcup_{n\in\omega} f^n(\perp)) \\ &= \bigsqcup_{n\in\omega} f(f^n(\perp)) \\ &= \bigsqcup_{n\in\omega} f^{n+1}(\perp) \end{aligned} \tag{4.2}$$

Now we have a new chain :

$$f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \dots$$

Now we have:

$$\begin{aligned} \bigsqcup_{n\in\omega} f^{n+1}(\bot) &= \left(\bigsqcup_{n\in\omega} f^{n+1}(\bot)\right) \sqcup \{\bot\} \\ &= \bigsqcup_{n\in\omega} f^{n}(\bot) \\ &= \text{fix}(f). \end{aligned} \tag{4.4}$$

2. We will prove that $\text{fix}(f)$ is the least fixpoint. Let us assume that d is a pre-fixpoint of f, that is $f(d) \sqsubseteq d$. By induction we show that $\forall n \in \omega. f^n(\bot) \sqsubseteq d$ (i.e., d is an upper bound for the chain $\{f^n(\bot)\}_{n\in\omega}$):

- base case: obviously $\bot \sqsubseteq d$
- inductive step: let us assume $f^n(\bot) \sqsubseteq d$

$$\begin{aligned} f^{n+1}(\bot) &= f(f^n(\bot)) && \text{by definition} \\ &\sqsubseteq f(d) && \text{by monotonicity of } f \text{ and inductive hypothesis} \\ &\sqsubseteq d && \text{because } d \text{ is a pre-fixpoint} \end{aligned}$$

So $\text{fix}(f) \sqsubseteq d$ is the least pre-fixpoint of f, and in particular the least fixpoint of f. □

Now let us make two examples which show that bottom and continuity are required to compute $\text{fix}(f)$.

Example 4.36 (Bottom is necessary)
Let $(\{True, False\}, \sqsubseteq)$ be the partial order of boolean values (i.e. a discrete order with two elements) obviously it is complete. The identity function Id is monotone. In fact there is no least fixpoint. To ensure the existence of the $\text{lfp}(Id)$ *we need a bottom element in the CPO.*

Example 4.37 (Continuity is necessary)
Let us consider the CPO $(\omega \cup \{\infty_1, \infty_2\}, \sqsubseteq)$ where $\sqsubseteq\restriction_{\omega} = \leq$, $\forall d \in \omega \cup \{\infty_1\}.\ d \sqsubseteq \infty_1$ and $\forall d \in \omega \cup \{\infty_1, \infty_2\}.\ d \sqsubseteq \infty_2$. We define a monotone function f as follows:

$$f(n) = n + 1 \quad f(\infty_1) = \infty_2 \quad f(\infty_2) = \infty_2$$

note that f is not continue. Let us consider $C = \{d_i\}_{i\in\omega}$ be the even numbers chain we have:

$$\bigsqcup_{i\in\omega} d_i = \infty_1 \quad f(\bigsqcup_{i\in\omega} d_i) = \infty_2 \quad \bigsqcup_{i\in\omega} f(d_i) = \infty_1$$

Note that f has at least one fixpoint, indeed:

$$f(\infty_2) = \infty_2$$

But nothing ensures that the fixpoint is reachable.

4.3. Immediate Consequence Operator

4.3.1. The $\hat{R}$ Operator

In this section we compare for the first time two different approaches for defining semantics: inference rules and fixpoint theory. We will see that the set of theorems of a generic logical system can be defined as a fixpoint of a suitable operator.

Let us consider an inference rule system, and the set $\mathscr{F}$ of the well-formed formulas of the language handled by the rules. We define an operator on the $CPO_\bot$ of sets of well-formed formulas ordered by inclusion $\left(2^{\mathscr{F}}, \subseteq\right)$.

Definition 4.38 (Immediate consequence operator $\hat{R}$)
Let R be a set of inference rules and let $B \subseteq \mathscr{F}$ be a set of well-formed formulas, we define:

$$\hat{R}(B) = \left\{ y | \exists ({}^{X}/_{y}) \in R . X \subseteq B \right\}$$

$\hat{R} : 2^{\mathscr{F}} \to 2^{\mathscr{F}}$ *is called* immediate consequence operator.

The operator $\hat{R}$ when applied to a set of formulas B calculates a new set of formulas by using the inference rules as shown in Chapter 1. Now we will show that the set of theorems proved by the rule system R is equal to the least fixpoint of the immediate consequence operator $\hat{R}$.

To apply the fixpoint theorem, we need monotonicity and continuity of $\hat{R}$.

Theorem 4.39 (Monotonicity of $\hat{R}$)
$\hat{R}$ is a monotone function.

Proof. We want to show that $\hat{R}(B_1) \subseteq \hat{R}(B_2)$.

Let $B_1 \subseteq B_2$. Let us assume $y \in \hat{R}(B_1)$, then there exists a rule $({}^{X}/_{y})$ in R, and $X \subseteq B_1$. So we have $X \subseteq B_2$ and $y \in \hat{R}(B_2)$. □

Theorem 4.40 (Continuity of $\hat{R}$)
Let R be a set of rules of the form $({}^{X}/_{y})$, with X a finite set, then $\hat{R}$ is continuous.

Proof. We will prove that $\bigcup_{n\in\omega} \hat{R}(B_n) = \hat{R}(\bigcup_{n\in\omega} B_n)$.
So as usual we prove:

1. $\bigcup_{n\in\omega} \hat{R}(B_n) \subseteq \hat{R}(\bigcup_{n\in\omega} B_n)$
2. $\bigcup_{n\in\omega} \hat{R}(B_n) \supseteq \hat{R}(\bigcup_{n\in\omega} B_n)$

1. Let y be an element of $\bigcup_{n\in\omega} \hat{R}(B_n)$ so there exists a natural number m such that $y \in \hat{R}(B_m)$. Since $B_m \subseteq \bigcup_{n\in\omega} B_n$ by monotonicity $\hat{R}(B_m) \subseteq \hat{R}(\bigcup_{n\in\omega} B_n)$ so $y \in \hat{R}(\bigcup_{n\in\omega} B_n)$.
2. Let y be an element of $\hat{R}(\bigcup_{n\in\omega} B_n)$ so there exists a rule X/y with $X \subseteq \bigcup_{n\in\omega} B_n$.
   Since X is finite there exists a natural number k such that $X \subseteq B_k$, otherwise $X \not\subseteq \bigcup_{n\in\omega} B_n$. In fact if $X = \{x_i\}_{i=1,\ldots,N}$, then $k_i = \min\{j \mid x_i \in B_j\}$ and $k = max\{k_i\}_{i=1,\ldots N}$. So $y \in \hat{R}(B_k)$ then $y \in \bigcup_{n\in\omega} \hat{R}(B_n)$ as required.

□

4.3.2. Fixpoint of $\hat{R}$

Now we are ready to present the fixpoint of $\hat{R}$. For this purpose let us define I_R the set of theorems provable from the set of rules R:

$$I_R = \bigcup_{i\in\omega} I_R^i$$

where

$$I_R^0 \stackrel{\text{def}}{=} \varnothing$$
$$I_R^{n+1} \stackrel{\text{def}}{=} \hat{R}(I_R^n) \cup I_R^n$$

Note that the generic I_R^n contains all theorems provable with derivations of depth at most n, and I_R contains all theorems provable by using the rule system R.

Theorem 4.41
Let R a rule system, it holds:

$$\forall n \in \omega.\ I_R^n = \hat{R}^n(\varnothing)$$

Proof. By induction on n

Basis) $I_R^0 = \hat{R}^0(\varnothing) = \varnothing$.

Ind.) We assume $I_R^n = \hat{R}^n(\varnothing)$. Then:

$$\begin{aligned} I_R^{n+1} &= \hat{R}(I_R^n) \cup I_R^n \\ &= \hat{R}(\hat{R}^n(\varnothing)) \cup I_R^n \\ &= \hat{R}^{n+1}(\varnothing) \cup I_R^n \\ &= \hat{R}^{n+1}(\varnothing) \cup \hat{R}^n(\varnothing) \\ &= \hat{R}^{n+1}(\varnothing) \end{aligned}$$

In the last step of the proof we have exploited the property $\hat{R}^{n+1}(\varnothing) \supseteq \hat{R}^n(\varnothing)$, which can be readily proved by mathematical induction (the base case amounts to $\hat{R}(\varnothing) \supseteq \varnothing$ that trivially holds and the inductive case follows by monotonicity of $\hat{R}$, as $\hat{R}^{n+1}(\varnothing) \supseteq \hat{R}^n(\varnothing)$ implies $\hat{R}^{n+2}(\varnothing) = \hat{R}(\hat{R}^{n+1}(\varnothing)) \supseteq \hat{R}(\hat{R}^n(\varnothing)) = \hat{R}^{n+1}(\varnothing)$). □

Theorem 4.42 (Fixpoint of $\hat{R}$)
Let R a rule system, it holds:

$$fix(\hat{R}) = I_R$$

Proof. By using the fixpoint theorem we have that there exists $\mathrm{lfp}(\hat{R})$ and that it is equal to $\bigcup_{n\in\omega} \hat{R}^n(\varnothing) \stackrel{\text{def}}{=} \mathrm{fix}(\hat{R})$, then:

$$I_R \stackrel{\text{def}}{=} \bigcup_{n\in\omega} I_R^n = \bigcup_{n\in\omega} \hat{R}^n(\varnothing) \stackrel{\text{def}}{=} \mathrm{fix}(\hat{R})$$

as required.

□

Example 4.43 (Rule system with discontinuous $\hat{R}$)

$$\frac{\varnothing}{P(1)} \qquad \frac{P(x)}{P(x+1)} \qquad \frac{\forall x \text{ odd. } P(x)}{P(0)}$$

To ensure the continuity of $\hat{R}$ the theorem 4.40 requires that the system has only rules with finitely many premises. The third rule of our system instead has infinitely many premises.
The continuity of $\hat{R}$, namely $\forall \{B_n\}_{n\in\omega}.\ \cup_{n\in\omega} \hat{R}(B_n) = \hat{R}(\cup_{n\in\omega} B_n)$, does not hold in this case. Indeed if we take the chain

$$\{P(1)\} \subseteq \{P(1), P(3)\} \subseteq \{P(1), P(3), P(5)\} \ldots$$

We have :

B_i	$\{P(1)\}$	$\subseteq \{P(1), P(3)\}$	$\subseteq \{P(1), P(3), P(5)\}$
$\hat{R}(B_i)$	$\{P(1), P(2)\}$	$\subseteq \{P(1), P(2), P(4)\}$	$\subseteq \{P(1), P(2), P(4), P(6)\}$

Then we have:

$$\bigcup_{i\in\omega} B_i = \{P(1), P(3), P(5), \ldots\}$$
$$\bigcup_{i\in\omega} \hat{R}(B_i) = \{P(1), P(2), P(4), \ldots\}$$

$$\hat{R}(\bigcup_{i\in\omega} B_i) = \{P(1), P(2), P(4) \ldots \underbrace{P(0)}_{\text{3rd rule}}\}$$

since the third rule applies only when the predicate is proved for all the odd numbers.

$$\text{fix}(\hat{R}) = \bigcup_{n\in\omega} \hat{R}^n(\varnothing) = \{P(1), P(2), P(3), P(4), \ldots\}$$

In fact, we have:

$$\hat{R}^0(\varnothing) = \varnothing \qquad \hat{R}^1(\varnothing) = \{P(1)\} \qquad \hat{R}^2(\varnothing) = \{P(1), P(2)\} \qquad \hat{R}^3(\varnothing) = \{P(1), P(2), P(3)\}$$

Thus,

$$\hat{R}(\text{fix}(\hat{R})) = \{P(0), P(1), P(2), P(3), P(4), \ldots\}$$

Then we can not use the fixpoint theorem, and $\text{fix}(\hat{R})$ *is not a fixpoint of* $\hat{R}$ *since* $\text{fix}(\hat{R}) \neq \hat{R}(\text{fix}(\hat{R}))$.

Example 4.44 (String)
Let us consider the grammar

$$S ::= \lambda \mid (S) \mid SS$$

Obviously using the inference rule formalism we can write:

$$\frac{\varnothing}{\lambda \in S} \qquad \frac{s \in S}{(s) \in S} \qquad \frac{s_1 \in S \quad s_2 \in S}{s_1 s_2 \in S}$$

So we can use the $\hat{R}$ *operator and the fixpoint theorem to find all the strings generated by the grammar:*

$$S_0 = \hat{R}^0(\varnothing) = \varnothing$$
$$S_1 = \hat{R}(S_0) = \lambda + (\varnothing) + \varnothing\varnothing = \{\lambda\}$$
$$S_2 = \hat{R}(S_1) = \lambda + (\lambda) + \lambda\lambda = \{\lambda,\ (\,)\}$$
$$S_3 = \hat{R}(S_2) = \lambda + (\lambda) + ((\,)) + (\,)(\,) = \{\lambda,\ (\,),\ ((\,)),\ (\,)(\,)\}$$
$$\ldots$$

So the language generated by the grammar is $\text{fix}(\hat{R})$.

5. Denotational Semantics of IMP

The same language can be assigned different kinds of semantics, depending on the properties under study or the aspects we are interested in representing. The operational semantics is closer to the memory-based, executable machine-like view: given a program and a state, we derive the state obtained after the execution of that program. We now give a more abstract, purely mathematical semantics, called *denotational semantics*, that takes a program and returns the transformation function over memories associated with that program: it takes an initial state as argument and returns the final state as result. Since functions will be written in some fixed mathematical notation, i.e. they can also be regarded as "programs" of a suitable formalism, we can say that, to some extent, the operational semantics defines an "interpreter" of the language (given a program *and* the initial state it returns the final state obtained by executing the program), while the denotational semantics defines a "compiler" for the language (from programs to functions, i.e. programs written in a more abstract language).

The *(meta-)language* we shall rely on for writing functions is called λ**-notation**.

5.1. λ-notation

The lambda calculus was introduced by Alonzo Church in 1930 in order to answer one of the questions in the Hilbert's program. As we said we will use lambda notation as meta-language, this means that we will express the semantics of IMP by lambda terms.

Definition 5.1 (Lambda terms)
We define lambda terms *as the terms generated by the grammar:*

$$t \ ::= \ x \ \mid \ \lambda x.t \ \mid \ tt \ \mid \ t \to t, t$$

Where x is a variable.

As we can see the lambda notation is very simple, it has four constructs:

- x: is a simple *variable*.
- $\lambda x.t$: is the *lambda abstraction* which allows to define anonymous functions.
- tt': is the *application* of a function t to its argument t'.
- $t \to t', t''$ is the conditional operator, i.e. the "If-Then-Else" construct in lambda notation.

All the notions used in this definition, like "true" and "false" can be formalized in lambda notation only by using lambda abstraction, but this development is beyond the scope of the course. In the following we will take the liberty to assume that data types such as integers and booleans are available in the lambda-notation as well as the usual operations on them.

Definition 5.2
Let t, t_0 and t_1 be three lambda terms, we define:

$$t \to t_0, t_1 = \begin{cases} t_0 & \text{if } t = true \\ t_1 & \text{if } t = false \end{cases}$$

Lambda abstraction $\lambda x.t$ is the main feature. It allows to define functions, where x represents the parameter of the function and t is the lambda term which represents the body of the function. For example the term $\lambda x.x$

is the identity function. Usually we equip the lambda notation with some equations. Before introducing these equations we will present the notions of *substitution* and *free variable*. Substitution allows to systematically substitute a lambda term for a variable.

Definition 5.3 (Free variables)
Let t and t' be two lambda terms, we define:

$$\begin{aligned} fv(x) &= \{x\} \\ fv(\lambda x.t) &= fv(t) \setminus \{x\} \\ fv(tt') &= fv(t) \cup fv(t') \\ fv(t \to t', t'') &= fv(t) \cup fv(t') \cup fv(t'') \end{aligned}$$

The second equation highlights that the lambda abstraction is a binding operator.

Definition 5.4 (Capture-avoiding substitution)
Let t, t', t'' and t''' be four lambda terms, we define:

$$\begin{aligned} y[t/x] &= \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \\ (\lambda y.t')[t/x] &= \lambda z.t'[z/y][t/x] \ \text{ if } z \notin fv(\lambda y.t') \cup fv(t) \cup \{x\} \\ (t_0 t_1)[t/x] &= t_0[t/x]t_1[t/x] \\ (t' \to t_0, t_1)[t/x] &= t'[t/x] \to t_0[t/x], t_1[t/x] \end{aligned}$$

Note that the matter of names is not so trivial. In the second equation we first rename y with a fresh name z, than go ahead with the substitution. This solution is motivated by the fact that y might not be free in t, but it introduces some non-determinism in the equations due to the arbitrary nature of the new name z. This non-determinism immediately disappear if we regard the terms up to the name equivalence defined as follows:

Definition 5.5 (Alpha-conversion)
Let t be a lambda term we define:

$$\lambda x.t = \lambda y.t[y/x] \ \text{ if } y \notin fv(t)$$

this property is called α-conversion.

Obviously α-conversion and substitution should be defined at the same time to avoid circularity. By using the α-conversion we can prove statements like $\lambda x.x = \lambda y.y$.

Definition 5.6 (Beta-conversion)
Let t, t' be two lambda terms we define:

$$(\lambda x.t')t = t'[t/x]$$

this property is called β-conversion.

This second equation allows to understand how to interpret function applications.
Finally we introduce some syntactic sugar and conventions.

- $tt't'' = (tt')t''$ (i.e., application is left-associative)
- $\lambda x.tt' = \lambda x.(tt')$
- $A \to B \times C = A \to (B \times C)$
- $A \times B \to C = (A \times B) \to C$
- $A \times B \times C = (A \times B) \times C$
- $A \to B \to C = A \to (B \to C)$

In the following, we will often omit parentheses.

5.2. Denotational Semantics of IMP

The denotational semantics of IMP consists of three separate *interpretation* functions, one for each syntax category (Aexp, Bexp, Com):

- each arithmetic expression is mapped to a function from states to integers: $\mathscr{A} : Aexp \to (\Sigma \to \mathbb{N})$;
- each boolean expression is mapped to a function from states to booleans: $\mathscr{B} : Bexp \to (\Sigma \to \mathbb{B})$;
- each command is mapped to a (partial) function from states to states: $\mathscr{C} : Com \to (\Sigma \rightharpoonup \Sigma)$.

5.2.1. Function $\mathscr{A}$

The denotational semantics of arithmetic expressions is defined as the function:

$$\mathscr{A} : Aexp \to \Sigma \to \mathbb{N}$$

We shall define $\mathscr{A}$ by exploiting *structural recursion* over the syntax of arithmetic expressions. Let us fix some notation:

We will rely on definitions of the form

$$\mathscr{A}\,[\![n]\!] = \lambda\sigma.n$$

with the following meaning:

- $\mathscr{A}$ is the interpretation function, typed in the functional space $Aexp \to \Sigma \to \mathbb{N}$
- n is an arithmetic expression (i.e. a term in Aexp). The surrounding brackets $[\![$ and $]\!]$ emphasize that it is a piece of syntax rather then part of the metalanguage.
- the expression $\mathscr{A}\,[\![n]\!]$ is a function $\Sigma \to \mathbb{N}$. Notice that also the right part of the equation must be of the same type $\Sigma \to \mathbb{N}$.

We shall often define the interpretation function $\mathscr{A}$ by writing equalities such as:

$$\mathscr{A}\,[\![n]\!]\,\sigma = n$$

instead of

$$\mathscr{A}\,[\![n]\!] = \lambda\sigma.n$$

In this way, we simplify the notation in the right-hand side. Notice that both sides of the equation ($\mathscr{A}\,[\![n]\!]\,\sigma$ and n) have the type $\mathbb{N}$.

Definition 5.7 (Denotational semantics of arithmetic expressions)
The denotational semantics of arithmetic expressions is defined by structural recursion as follows:

$$\begin{aligned}
\mathscr{A}\,[\![n]\!]\,\sigma &= n \\
\mathscr{A}\,[\![x]\!]\,\sigma &= \sigma x \\
\mathscr{A}\,[\![a_0 + a_1]\!]\,\sigma &= (\mathscr{A}\,[\![a_0]\!]\,\sigma) + (\mathscr{A}\,[\![a_1]\!]\,\sigma) \\
\mathscr{A}\,[\![a_0 - a_1]\!]\,\sigma &= (\mathscr{A}\,[\![a_0]\!]\,\sigma) - (\mathscr{A}\,[\![a_1]\!]\,\sigma) \\
\mathscr{A}\,[\![a_0 \times a_1]\!]\,\sigma &= (\mathscr{A}\,[\![a_0]\!]\,\sigma) \times (\mathscr{A}\,[\![a_1]\!]\,\sigma)
\end{aligned}$$

Let us briefly comment the above definitions.

Constants The denotational semantics of any constant n is just the constant function that always returns n for any σ.

Variables The denotational semantics of any variable x is the function that takes a memory σ and returns the value of x in σ.

Binary expressions The denotational semantics of any binary expression evaluates the arguments (with the same given σ) and combines the results by exploiting the corresponding arithmetic operation.

Note that the symbols "+","-" and "×" are overloaded: in the left hand side they represent elements of the syntax, while in the right hand side they represent operators of the metalanguage. Similarly for "n" in the first definition.

5.2.2. Function $\mathscr{B}$

Function $\mathscr{B}$ is defined in a very similar way. The only difference is that the values to be returned are elements of $\mathbb{B}$ and not of $\mathbb{N}$.

Definition 5.8 (Denotational semantics of boolean expressions)
The denotational semantics of boolean expressions is defined by structural recursion as follows:

$$\begin{aligned}
\mathscr{B}[\![v]\!]\sigma &= v \\
\mathscr{B}[\![a_0 = a_1]\!]\sigma &= (\mathscr{A}[\![a_0]\!]\sigma) = (\mathscr{A}[\![a_1]\!]\sigma) \\
\mathscr{B}[\![a_0 \leq a_1]\!]\sigma &= (\mathscr{A}[\![a_0]\!]\sigma) \leq (\mathscr{A}[\![a_1]\!]\sigma) \\
\mathscr{B}[\![\neg b_0]\!]\sigma &= \neg(\mathscr{B}[\![b]\!]\sigma) \\
\mathscr{B}[\![b_0 \vee b_1]\!]\sigma &= (\mathscr{B}[\![b_0]\!]\sigma) \vee (\mathscr{B}[\![b_1]\!]\sigma) \\
\mathscr{B}[\![b_0 \wedge b_1]\!]\sigma &= (\mathscr{B}[\![b_0]\!]\sigma) \wedge (\mathscr{B}[\![b_1]\!]\sigma)
\end{aligned}$$

5.2.3. Function $\mathscr{C}$

We are now ready to present the denotational semantics of commands. As we might expect the interpretation function of commands is the most complex. It has the following type:

$$\mathscr{C} : Com \to (\Sigma \rightharpoonup \Sigma)$$

As we saw we can define an equivalent total function. So we will employ a function of the type:

$$\mathscr{C} : Com \to (\Sigma \to \Sigma_\perp)$$

This will simplify our work. We start from the simplest commands:

$$\mathscr{C}[\![\textbf{skip}]\!]\sigma = \sigma$$

$$\mathscr{C}[\![x := a]\!]\sigma = \sigma\left[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}\right]$$

We see that "skip" does not modify the memory, while "$x := a$" evaluates the arithmetic expression "a" with its function $\mathscr{A}$ and then modifies the memory assigning the corresponding value to "x" .

Let us now consider the concatenation of two commands. In interpreting "$c_0; c_1$" we will interpret "c_0" from the starting state and then "c_1" from the state obtained from "c_0". Then from the first application of $\mathscr{C}[\![c_0]\!]$ we obtain a value in $\Sigma_\perp$ so we can not apply $\mathscr{C}[\![c_1]\!]$. To work this problem out we introduce a *lifting* operator $\_^*$.

Definition 5.9 (Lifting)
Let $f : \Sigma \to \Sigma_\perp$, we define a function $f^* : \Sigma_\perp \to \Sigma_\perp$ as follows:

$$f^*(x) = \begin{cases} \perp & \text{if } x = \perp \\ f(x) & \text{Otherwise} \end{cases}$$

Note that $(\cdot)^*$ is an operator of the type $(\Sigma \to \Sigma_\perp) \to (\Sigma_\perp \to \Sigma_\perp)$.

So the definition of the interpretation function for "$c_0; c_1$" is:

$$\mathscr{C}[\![c_0; c_1]\!]\sigma = \mathscr{C}[\![c_1]\!]^*(\mathscr{C}[\![c_0]\!]\sigma)$$

Let us now consider the conditional command "if". Recall that the λ-calculus provides a conditional operator, then we have simply:

$$\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma$$

Now we present the semantics of the "while" command. We could think to define the semantics of "while" simply as:

$$\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma$$

Obviously this definition is not a structural recursion definition rule. Indeed structural recursion allows only for subterms, while here "**while** b **do** c" appears on both sides. To solve this issue we will reduce the problem of defining the semantics of "while" to a fixpoint calculation. Let us define a function $\Gamma_{b,c}$.

$$\Gamma_{b,c} = \lambda\varphi.\,\lambda\sigma.\,\underbrace{\underbrace{\underbrace{\mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![c]\!]\sigma), \sigma}_{\Sigma_\perp}}_{\Sigma\to\Sigma_\perp}}_{(\Sigma\to\Sigma_\perp)\to\Sigma\to\Sigma_\perp}$$

As we can see the function $\Gamma_{b,c}$ is of type $(\Sigma \to \Sigma_\perp) \to \Sigma \to \Sigma_\perp$ and contains only subterms of "**while** b **do** c". Clearly we require that:

$$\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!] = \Gamma_{b,c}\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]$$

We want to define the semantics of the "while" command as the least fixpoint of $\Gamma_{b,c}$. Now we will show that $\Gamma_{b,c}$ is a monotone continuous function, so that we can prove that $\Gamma_{b,c}$ has a least fixpoint and that:

$$\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!] = \text{fix } \Gamma_{b,c} = \bigsqcup_{n\in\omega} \Gamma_{b,c}^n(\perp_{\Sigma\to\Sigma_\perp})$$

To prove continuity we will consider $\Gamma_{b,c}$ as applied to partial functions: $\Gamma_{b,c} : (\Sigma \rightharpoonup \Sigma) \longrightarrow (\Sigma \rightharpoonup \Sigma)$. Partial functions in $\Sigma \rightharpoonup \Sigma$ can be represented as sets of (well formed) formulas $\sigma \longrightarrow \sigma'$. Then the effect of $\Gamma_{b,c}$ can be represented by the immediate consequence operators for the following (infinite) set of rules.

$$R_{\Gamma_{b,c}} = \left\{ \frac{\mathscr{B}[\![b]\!]\sigma \quad \mathscr{C}[\![c]\!]\sigma = \sigma'' \quad \sigma'' \to \sigma'}{\sigma \to \sigma'}, \qquad \frac{\neg\mathscr{B}[\![b]\!]\sigma}{\sigma \to \sigma} \right\} \text{ with } \hat{R}_{\Gamma_{b,c}} = \Gamma_{b,c}$$

Notice that here the only well formed formulas are $\sigma'' \longrightarrow \sigma'$, $\sigma \longrightarrow \sigma'$ and $\sigma \longrightarrow \sigma$. An instance of the first rule schema is obtained by picking up a value of σ such that $\mathscr{B}[\![b]\!]\sigma$ is true, and a (the) value of σ'' such that $\mathscr{C}[\![c]\!]\sigma = \sigma''$. Then for every σ' such that $\sigma'' \to \sigma'$ we can imply $\sigma \to \sigma'$. Similarly for the second rule schema.

Since all the rules obtained in this way have a finite number of premises (actually one or none), we can apply Theorem 4.40, which ensures the continuity of $\hat{R}_{\Gamma_{b,c}}$. Now by using Theorem 4.42 we have:

$$\text{fix } \hat{R}_{\Gamma_{b,c}} = I_{R_{\Gamma_{b,c}}} = \text{fix } \Gamma_{b,c}$$

Let us conclude this section with three examples which explain how to use the definitions we have given.

Example 5.10
Let us consider the command:

$$w = \textbf{while true do skip}$$

now we will see how to calculate its semantics. We have $\mathscr{C}[\![w]\!] = fix\,\Gamma_{\textbf{true,skip}}$ *where*

$$\Gamma_{\textbf{true,skip}}\varphi\sigma \quad = \quad \mathscr{B}[\![\textbf{true}]\!]\sigma \to \varphi^*(\mathscr{C}[\![\textbf{skip}]\!]\sigma), \sigma \quad = \quad \varphi^*(\mathscr{C}[\![\textbf{skip}]\!]\sigma) \quad = \quad \varphi^*\sigma \quad = \quad \varphi\sigma$$

So we have $\Gamma_{\textbf{true,skip}}\varphi = \varphi$*, that is* $\Gamma_{\textbf{true,skip}}$ *is the identity function. Then each function* φ *is a fixpoint of* $\Gamma_{\textbf{true,skip}}$*, but we are looking for the least fixpoint. This means that the sought solution is the least function in the* $CPO_\perp$ *of functions. Then we have* fix $\Gamma_{\textbf{true,skip}} = \lambda\sigma.\perp_{\Sigma_\perp}$.

In the following we will often write just Γ when the subscripts b and c are obvious from the context.

Example 5.11
Now we will see the equivalence between $w = \textbf{while}\, b\, \textbf{do}\, c$ *and* $\textbf{if}\, b\, \textbf{then}\, c; w\, \textbf{else skip}$ *for any b and c. Since* $\mathscr{C}[\![w]\!]$ *is a fixpoint we have:*

$$\mathscr{C}[\![w]\!] = \Gamma(\mathscr{C}[\![w]\!]) = \lambda\sigma.\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![w]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma$$

Moreover we have:

$$\begin{aligned}&\mathscr{C}[\![\textbf{if}\, b\, \textbf{then}\, c; w\, \textbf{else skip}]\!] = \lambda\sigma.\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c;w]\!]\sigma, \mathscr{C}[\![\textbf{skip}]\!]\sigma\\ &= \lambda\sigma.\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![w]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma\end{aligned}$$

Example 5.12
Let us consider the command:

$$c = \textbf{while}\, x \neq 0\, \textbf{do}\, x := x - 1$$

we have:

$$\mathscr{C}[\![c]\!] = \text{fix}\ \Gamma$$

where $\Gamma\varphi = (\lambda\, \sigma.\sigma x \neq 0 \to \varphi\, \sigma[\sigma x - 1/x], \sigma)$. *Let us see some iterations:*

$$\begin{aligned}&\varphi_0 = \Gamma^0\, \bot_{\Sigma\to\Sigma_\bot} = \bot_{\Sigma\to\Sigma_\bot} = \lambda\, \sigma.\bot_{\Sigma_\bot}\\ &\varphi_1 = \Gamma\, \varphi_0 = \lambda\, \sigma.\sigma x \neq 0 \to \underbrace{\bot_{\Sigma\to\Sigma_\bot}}_{\varphi_0}\, \sigma[\sigma x - 1/x], \sigma = \Gamma\, \varphi_0 = \lambda\, \sigma.\sigma x \neq 0 \to \bot_{\Sigma_\bot}, \sigma\\ &\varphi_2 = \Gamma\, \varphi_1 = \lambda\, \sigma.\sigma x \neq 0 \to \underbrace{(\lambda\, \sigma'.\, \sigma' x \neq 0 \to \bot_{\Sigma_\bot}, \sigma')}_{\varphi_1}\, \sigma[\sigma x - 1/x], \sigma\end{aligned}$$

Now we have the following possibilities:

$$\frac{\sigma x < 0}{\varphi_2\sigma = \bot} \qquad \frac{\sigma x = 0}{\varphi_2\sigma = \sigma} \qquad \frac{\sigma x = 1}{\varphi_2\sigma = \sigma[\sigma x - 1/x]} \qquad \frac{\sigma x > 1}{\varphi_2\sigma = \bot}$$

So we have:

$$\varphi_2 = \lambda\, \sigma.\sigma x < 0 \to \bot, 0 \leqslant \sigma x < 2 \to \sigma[0/x], \bot$$

We can now attempt to formulate a conjecture:

$$\forall n \in \omega.\varphi_n = \lambda\, \sigma.\sigma x < 0 \to \bot, 0 \leqslant \sigma x < n \to \sigma[0/x], \bot$$

We are now ready to prove our conjecture by mathematical induction on n.

The base case is trivial, indeed $\varphi_0 = \Gamma^0\, \bot = \lambda\, \sigma.\bot_{\Sigma_\bot}$.

For the inductive case. let us now assume the predicate for n and prove it for $n + 1$. *So as usual we assume:*

$$P(n) = (\varphi_n = \lambda\, \sigma.\sigma x < 0 \to \bot, 0 \leqslant \sigma x < n \to \sigma[0/x], \bot)$$

and prove:

$$P(n + 1) = (\varphi_{n+1} = \lambda\, \sigma.\sigma x < 0 \to \bot, 0 \leqslant \sigma x < n + 1 \to \sigma[0/x], \bot)$$

By definition:

$$\varphi_{n+1} = \Gamma\, \varphi_n = \lambda\, \sigma.\sigma\, x \neq 0 \to \underbrace{(\lambda\, \sigma'.\sigma' x < 0 \to \bot, 0 \leqslant \sigma' x < n \to \sigma'[0/x], \bot)}_{\varphi_n}\sigma[\sigma x - 1/x], \sigma$$

we have:

$$\frac{\sigma x < 0}{\varphi_{n+1}\sigma = \bot} \qquad \frac{\sigma x = 0}{\varphi_{n+1}\sigma = \sigma = \sigma[0/x]}$$

$$\frac{1 \leq \sigma x < n+1}{\varphi_{n+1}\sigma = \sigma[\sigma x - 1/x][0/x] = \sigma[0/x]} \qquad \frac{\sigma x \geq n+1}{\varphi_{n+1}\sigma = \bot}$$

Then:

$$\varphi_{n+1} = \lambda\, \sigma.\sigma x < 0 \to \bot, 0 \leqslant \sigma x < n+1 \to \sigma[0/x], \bot$$

Finally we have:

$$\mathscr{C}\,[\![c]\!] = \text{fix}\ \Gamma = \bigsqcup_{i\in\omega} \varphi^i \bot = \lambda\sigma.\sigma x < 0 \to \bot, \sigma[0/x]$$

5.3. Equivalence Between Operational and Denotational Semantics

This section deals with the issue of equivalence between the two semantics of IMP introduced up to now. As we will show the denotational and operational semantics agree. As usual we will handle first arithmetic and boolean expressions, then assuming the proved equivalences we will show that operational and denotational semantics agree also on commands.

5.3.1. Equivalence Proofs for $\mathscr{A}$ and $\mathscr{B}$

We start from the arithmetic expressions. The property which we will prove is the following:

$$P(a) \stackrel{\text{def}}{=} \langle a, \sigma\rangle \to \mathscr{A}\,[\![a]\!]\,\sigma \quad \forall a \in Aexpr$$

That is, the results of evaluating an arithmetic expression both by operational and denotational semantics are the same. If we regard the operational semantics as an interpreter and the denotational semantics as a compiler we are proving that interpreting an IMP program and executing its compiled version starting from the same memory leads to the same result. The proof is by structural induction on the arithmetic expressions.

Constants $P(n) \stackrel{\text{def}}{=} \langle n, \sigma\rangle \to \mathscr{A}\,[\![n]\!]\,\sigma$ Since $\mathscr{A}\,[\![n]\!]\,\sigma = n$ and $\langle n, \sigma\rangle \to n$.

Variables $P(x) \stackrel{\text{def}}{=} \langle x, \sigma\rangle \to \mathscr{A}\,[\![x]\!]\,\sigma$ Since $\mathscr{A}\,[\![x]\!]\,\sigma = \sigma(x)$ and $\langle x, \sigma\rangle \to \sigma(x)$.

Binary operators Let us generalize the proof for the binary operations of arithmetic expressions. Consider two arithmetic expressions a_0 and a_1 and a binary operator $\circ$ of IMP, whose corresponding semantic operator is $\odot$. We would like to prove

$$P(a_0) \wedge P(a_1) \Rightarrow P(a_0 \circ a_1)$$

So as usual we assume $P(a_0) \stackrel{\text{def}}{=} \langle a_0, \sigma\rangle \to \mathscr{A}\,[\![a_0]\!]\,\sigma$ and $P(a_1) \stackrel{\text{def}}{=} \langle a_1, \sigma\rangle \to \mathscr{A}\,[\![a_1]\!]\,\sigma$ and we prove $P(a_0 \circ a_1) \stackrel{\text{def}}{=} \langle a_0 \circ a_1, \sigma\rangle \to \mathscr{A}\,[\![a_0 \circ a_1]\!]\,\sigma$.
We have:

$\langle a_0 \circ a_1, \sigma\rangle \to \mathscr{A}\,[\![a_0]\!]\,\sigma \odot \mathscr{A}\,[\![a_1]\!]\,\sigma$	by operational semantics definition and using the preconditions
$\mathscr{A}\,[\![a_0]\!]\,\sigma \odot \mathscr{A}\,[\![a_1]\!]\,\sigma = \mathscr{A}\,[\![a_0 \circ a_1]\!]\,\sigma$	by denotational semantics definition

So we have $P(a_0 \circ a_1)$.

The boolean expressions case is completely similar to that of arithmetic expressions, so we leave the proofs as an exercise. From now on we will assume the equivalence between denotational and operational semantics for boolean and arithmetic expressions.

5.3.2. Equivalence of $\mathscr{C}$

Central to the proof of equivalence between denotational and operational semantics is the case of commands. Operational and denotational semantics are defined in very different formalisms: on the one hand we have an inference rule system which allows to calculate the execution of each command, on the other hand we have a function which associates to each command its functional meaning. So to prove the equivalence between the two semantics we will prove the following:

$$\forall c \in Com.\ \forall \sigma, \sigma' \in \Sigma.\ \langle c, \sigma\rangle \to \sigma' \iff \mathscr{C}[\![c]\!]\sigma = \sigma'$$

As usual we will divide the proof in two parts:

- $P(\langle c, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma'$ (Completeness)
- $P(c) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \in \Sigma.\ \mathscr{C}[\![c]\!]\sigma = \sigma' \Rightarrow \langle c, \sigma\rangle \to \sigma'$ (Correctness)

Notice that in this way it also guaranteed equivalence for the non defined cases: for instance we have $\langle c, \sigma\rangle \nrightarrow \Rightarrow\ \mathscr{C}[\![c]\!]\sigma = \perp_{\Sigma_\perp}$ since otherwise $\mathscr{C}[\![c]\!]\sigma = \sigma'$ would imply $\langle c, \sigma\rangle \to \sigma'$. Similarly in the other direction.

5.3.2.1. Completeness of the Denotational Semantics

Let us prove the first part of the theorem: $P(\langle c, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma'$.
Since the operational semantic is defined by a rule system we will proceed by rule induction. So for each rule we will assume the property on the premises and we will prove the property on the consequence.

Skip Let us consider the **skip** command. The corresponding operational rule is

$$\frac{}{\langle \mathbf{skip}, \sigma\rangle \to \sigma}$$

We would like to prove:

- $P(\langle \mathbf{skip}, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \mathscr{C}[\![\mathbf{skip}]\!]\sigma = \sigma$

Obviously the proposition is true by definition of denotational semantic.

Assignment Let us consider the assignment command:

$$\frac{\langle a, \sigma\rangle \to m}{\langle x := a, \sigma\rangle \to \sigma\left[{}^{m}/_{x}\right]}$$

We assume:

- $\langle a, \sigma\rangle \to m$ and therefore $\mathscr{A}[\![a]\!]\sigma = m$ by the correspondence between the operational and denotational semantics of arithmetic expressions.

We will prove:

- $P(\langle x := a, \sigma\rangle \to \sigma\left[{}^{m}/_{x}\right]) \stackrel{\text{def}}{=} \mathscr{C}[\![x := a]\!]\sigma = \sigma\left[{}^{m}/_{x}\right]$

We have by the definition of denotational semantics:

$$\mathscr{C}[\![x := a]\!]\sigma = \sigma\left[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}\right] = \sigma\left[{}^{m}/_{x}\right]$$

Concatenation Let us consider the concatenation rule:

$$\frac{\langle c_0, \sigma\rangle \to \sigma'' \quad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c_0; c_1, \sigma\rangle \to \sigma'}$$

We assume:

- $P(\langle c_0, \sigma\rangle \to \sigma'') \stackrel{\text{def}}{=} \mathscr{C}[\![c_0]\!]\sigma = \sigma''$
- $P(\langle c_1, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c_1]\!]\sigma'' = \sigma'$.

We will prove:

- $P(\langle c_0; c_1, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c_0; c_1]\!]\sigma = \sigma'$

We have by definition:

$$\mathscr{C}[\![c_0; c_1]\!]\sigma = \mathscr{C}[\![c_1]\!]^*(\mathscr{C}[\![c_0]\!]\sigma) = \mathscr{C}[\![c_1]\!]^*\sigma'' = \mathscr{C}[\![c_1]\!]\sigma'' = \sigma'$$

Note that the lifting operator can be deleted because $\sigma'' \neq \perp$ by inductive hypothesis.

Conditional For the conditional command we have two rules:

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c_0, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'} \qquad \frac{\langle b, \sigma\rangle \to \textbf{false} \quad \langle c_1, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}$$

We will prove only the first case, the second proof is completely analogous, so we leave that as an exercise. We assume:

- $\langle b, \sigma\rangle \to \textbf{true}$ and therefore $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$ by the correspondence between operational and denotational semantics for boolean expressions;
- $P(\langle c_0, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c_0]\!]\sigma = \sigma'$.

And we will prove:

- $P(\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \sigma'$

We have:

$$\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma = \mathscr{C}[\![c_0]\!]\sigma = \sigma'$$

While As for the conditional we have two rules also for the "while" command:

$$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma} \qquad \frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'}$$

Let us consider the first rule. We assume:

- $\langle b, \sigma\rangle \to \textbf{false}$ and therefore $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$

We will prove:

- $P(\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \sigma$

We have by the fixpoint property of the denotational semantics:

$$\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma = \sigma$$

For the second rule we assume:

- $\langle b, \sigma\rangle \to \mathbf{true}$ and therefore $\mathscr{B}[\![b]\!]\sigma = \mathbf{true}$
- $P(\langle c, \sigma\rangle \to \sigma'') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma''$
- $P(\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![\mathbf{while}\, b\, \mathbf{do}\, c]\!]\sigma'' = \sigma'$

We will prove:

- $P(\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![\mathbf{while}\, b\, \mathbf{do}\, c]\!]\sigma = \sigma'$

By the definition of the denotational semantics:

$$\begin{aligned}\mathscr{C}[\![\mathbf{while}\, b\, \mathbf{do}\, c]\!]\sigma &= \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![\mathbf{while}\, b\, \mathbf{do}\, c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma\\ &= \mathscr{C}[\![\mathbf{while}\, b\, \mathbf{do}\, c]\!]^*(\mathscr{C}[\![c]\!]\sigma)\\ &= \mathscr{C}[\![\mathbf{while}\, b\, \mathbf{do}\, c]\!]\sigma''\\ &= \sigma'\end{aligned}$$

Note that the lifting operator can be deleted since $\sigma'' \neq \bot$ by inductive hypothesis.

5.3.2.2. Correctness of the Denotational Semantics

Since the denotational semantics is given by structural recursion we will proceed by induction on the structure of commands. We will prove (for all $c \in Com$):

$$P(c) \stackrel{\text{def}}{=} \forall \sigma, \sigma'\ \mathscr{C}[\![c]\!]\sigma = \sigma' \Rightarrow \langle c, \sigma\rangle \to \sigma'$$

Skip We need to prove:

- $P(\mathbf{skip}) \stackrel{\text{def}}{=} \forall \sigma, \sigma'\ \mathscr{C}[\![\mathbf{skip}]\!]\sigma = \sigma' \Rightarrow \langle \mathbf{skip}, \sigma\rangle \to \sigma'$

By definition we have $\mathscr{C}[\![\mathbf{skip}]\!]\sigma = \sigma$ and $\langle \mathbf{skip}, \sigma\rangle \to \sigma$ is an axiom of the operational semantics.

Assignment We need to prove:

- $P(x := a) \stackrel{\text{def}}{=} \forall \sigma, \sigma'\ \mathscr{C}[\![x := a]\!]\sigma = \sigma' \Rightarrow \langle x := a, \sigma\rangle \to \sigma'$

By definition we have $\sigma' = \sigma[^n/_x]$ where $\mathscr{A}[\![a]\!]\sigma = n$ so we have $\langle a, \sigma\rangle \to n$ and thus we can apply the rule:

$$\frac{\langle a, \sigma\rangle \to n}{\langle x := a, \sigma\rangle \to \sigma[^n/_x]}$$

Concatenation We want to prove:

- $P(c_0; c_1) \stackrel{\text{def}}{=} \forall \sigma, \sigma'\ \mathscr{C}[\![c_0; c_1]\!]\sigma = \sigma' \Rightarrow \langle c_0; c_1, \sigma\rangle \to \sigma'$

We can assume:

- $P(c_0) \stackrel{\text{def}}{=} \forall \sigma, \sigma''\ \mathscr{C}[\![c_0]\!]\sigma = \sigma'' \Rightarrow \langle c_0, \sigma\rangle \to \sigma''$
- $P(c_1) \stackrel{\text{def}}{=} \forall \sigma'', \sigma'\ \mathscr{C}[\![c_1]\!]\sigma'' = \sigma' \Rightarrow \langle c_1, \sigma''\rangle \to \sigma'$

Assuming the premise of the implication we want to prove, we have $\mathscr{C}[\![c_0; c_1]\!]\sigma = \mathscr{C}[\![c_1]\!]^*(\mathscr{C}[\![c_0]\!]\sigma) = \sigma'$. Since $\sigma' \neq \bot$, it must be $\mathscr{C}[\![c_0]\!]\sigma \neq \bot$, i.e., we can assume the termination of c_0 we can omit the lifting operator:

$$\mathscr{C}[\![c_0; c_1]\!]\sigma = \mathscr{C}[\![c_1]\!](\mathscr{C}[\![c_0]\!]\sigma) = \sigma'$$

Let $\mathscr{C}[\![c_0]\!]\sigma = \sigma''$. We have $\mathscr{C}[\![c_1]\!]\sigma'' = \sigma'$. Then we can apply modus ponens to the inductive assumptions, obtaining $\langle c_0, \sigma\rangle \to \sigma''$ and $\langle c_1, \sigma''\rangle \to \sigma'$. Thus we can apply the inference rule:

$$\frac{\langle c_0, \sigma\rangle \to \sigma'' \qquad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c_0; c_1, \sigma\rangle \to \sigma'}$$

Conditional We will prove:

- $P(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \; \mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \sigma' \Rightarrow \langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'$

As usual we must distinguish two cases, let us consider only the case with $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$, namely $\langle b, \sigma\rangle \to \textbf{false}$
We assume:

- $P(c_0) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \; \mathscr{C}[\![c_0]\!]\sigma = \sigma' \Rightarrow \langle c_0, \sigma\rangle \to \sigma'$
- $P(c_1) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \; \mathscr{C}[\![c_1]\!]\sigma = \sigma' \Rightarrow \langle c_1, \sigma\rangle \to \sigma'$
- $\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \sigma'$

By definition and since $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$ we have

$$\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma = \mathscr{C}[\![c_1]\!]\sigma = \sigma'$$

this implies by hypothesis $\langle c_1, \sigma\rangle \to \sigma'$. Thus we can apply the rule:

$$\frac{\langle c_1, \sigma\rangle \to \sigma' \qquad \langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}$$

While We will prove:

$$P(\textbf{while } b \textbf{ do } c) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \; \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$

We assume by structural induction:

- $P(c) \stackrel{\text{def}}{=} \forall \sigma, \sigma'' \; \mathscr{C}[\![c]\!]\sigma = \sigma'' \Rightarrow \langle c, \sigma\rangle \to \sigma''$

By definition $\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \text{fix } \Gamma_{b,c}\,\sigma$ so:

$$\begin{aligned}\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma' &= \text{fix } \Gamma_{b,c}\,\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma' \\ &= \left(\bigsqcup_{n\in\omega} \Gamma^n_{b,c}\bot\right)\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma' \\ &= \forall n.(\Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma')\end{aligned}$$

Notice that the last two properties are equivalent. In fact, if there is a pair $\sigma \to \sigma'$ in the limit, it must also occur in $\Gamma^n_{b,c}\bot$ for some n. Vice versa, if it belongs to $\Gamma^n_{b,c}\bot$ for some n then it belongs also to the limit. Let

$$A(n) \stackrel{\text{def}}{=} \Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$

We prove that $\forall n.\, A(n)$ by mathematical induction.

A(0) We have to prove:

$$\Gamma^0_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$

Since $\Gamma^0_{b,c}\bot\sigma = \bot\sigma = \bot$, obviously:

$$\bot = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$

Since the premise $\bot = \sigma'$ is false and hence the implication is true.

A(n) ⇒ A(n+1) Let us assume

$$A(n) \stackrel{\text{def}}{=} \Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'.$$

We want to show that

$$A(n+1) \stackrel{\text{def}}{=} \Gamma^{n+1}_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'.$$

We assume the premise of the implication, i.e. $\Gamma_{b,c}\left(\Gamma^n_{b,c}\bot\right)\sigma = \sigma'$, that is

$$\mathscr{B}[\![b]\!]\sigma \to \left(\Gamma^n_{b,c}\bot\right)^*\left(\mathscr{C}[\![c]\!]\sigma\right), \sigma = \sigma'$$

Now we distinguish two cases $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$ and $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$.

- if $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$, we have $\sigma' = \sigma$. By the correspondence between the denotational semantics and the operational semantics of boolean expressions, we know

  $$\mathscr{B}[\![b]\!]\sigma = \textbf{false} \Leftrightarrow \langle b, \sigma\rangle \to \textbf{false}\,.$$

  Now by using the rule:

  $$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma}$$

  we have $\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma$ as required.
- if $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$ as for the previous case we know that $\langle b, \sigma\rangle \to \textbf{true}$. The premise of the implication becomes $\left(\Gamma^n_{b,c}\bot\right)(\mathscr{C}[\![c]\!]\sigma) = \sigma'$. Note that we can omit the lifting operator because $\sigma' \neq \bot$. So we have $\mathscr{C}[\![c]\!]\sigma = \sigma''$ and by structural induction $\langle c, \sigma\rangle \to \sigma''$. Since $\left(\Gamma^n_{b,c}\bot\right)\sigma'' = \sigma'$ we have by mathematical induction hypothesis

  $$A(n) = \Gamma^n_{b,c}\bot\sigma'' = \sigma' \Rightarrow \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma''\rangle \to \sigma'$$

  that $\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma''\rangle \to \sigma'$. Finally by using the rule:

  $$\frac{\langle b, \sigma\rangle \to \textbf{true} \qquad \langle c, \sigma\rangle \to \sigma'' \qquad \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma''\rangle \to \sigma'}{\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'}$$

  we have $\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'$ as required.

5.4. Computational Induction

Up to this time the denotational semantics is less powerful than the operational, since we are not able to prove properties on fixpoints. To fill this gap we introduce *computational induction*, which applies to a class of properties corresponding to inclusive sets.

Definition 5.13 (Inclusive set)
Let $(D, \sqsubseteq)$ be a CPO, let P be a set, we say that P is an inclusive *set if and only if:*

$$(\forall n \in \omega, d_n \in P) \Rightarrow \bigsqcup_{n\in\omega} d_n \in P$$

A property is inclusive if the set of values on which it holds is inclusive.

Notice that if we consider a subset P of D as equipped with the same partial ordering $\sqsubseteq$ of D, then P is inclusive iff it is complete.

Intuitively, a set P is inclusive if whenever we form a chain out of elements in P, then the limit of the chain is also in P.

Example 5.14 (Non inclusive property)
Let $(\{a,b\}^* \cup \{a,b\}^\infty, \sqsubseteq)$ be a CPO where $x \sqsubseteq y \Leftrightarrow \exists z.\, y = xz$. So the elements of the CPO are sequences of a and b and $x \sqsubseteq y$ if x is a finite prefix of y. Let us now define the following property:

- *x is fair iff $\nexists y \in \{a,b\}^*.\ x = ya^\infty \ \vee\ x = yb^\infty$*

Fairness is the property of an arbiter which does not favor one of two competitors all the times from some point on. Fairness is not inclusive, indeed,

- *a^n is finite and thus fair for any $n \in \omega$;*
- *a^∞ is not fair; and*
- *$\bigsqcup_{n\in\omega} a^n = a^\infty$.*

Theorem 5.15 (Computational Induction)
Let P be a property, $(D, \sqsubseteq)$ a $CPO_\perp$ and f a monotone, continuous function on it. Then the inference rule:

$$\frac{P \text{ inclusive} \qquad \perp \in P \qquad \forall d \in D.(d \in P \Longrightarrow f(d) \in P)}{\mathrm{fix}(f) \in P}$$

is sound.

Proof. Given the second and the third premises, it is easy to prove by mathematical induction that $\forall n.\ f^n(\perp) \in P$. Then also $\bigsqcup_{n\in\omega} f^n(\perp) = \mathrm{fix}(f) \in P$ since P is inclusive. □

Example 5.16 (Computational induction)
Let us consider the command

$$w = \textbf{while}\ x \neq 0\ \textbf{do}\ x := x - 1$$

from Example 5.12. We want to prove the property

$$\mathscr{C}[\![\textbf{while}\ x \neq 0\ \textbf{do}\ x := x - 1]\!]\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]$$

By definition:

$$\mathscr{C}[\![w]\!] = \mathrm{fix}(\Gamma) \quad \textit{where} \quad \Gamma\varphi\sigma \stackrel{\text{def}}{=} \sigma x \neq 0 \rightarrow \varphi\sigma[^{\sigma x - 1}/_x], \sigma$$

Let us define the property:

$$P(\varphi) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.(\varphi\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x])$$

we will show that the property is inclusive, that is, taken a chain $\{\varphi_i\}_{i\in\omega}$ we have:

$$(\forall i \in \omega, P(\varphi_i)) \Rightarrow P(\bigsqcup_{i\in\omega} \varphi_i)$$

By expanding the property P in the above formula:

$$\Big(\forall i, \sigma, \sigma'.\Big(\varphi_i\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]\Big)\Big) \Rightarrow \forall \sigma, \sigma'.\left((\bigsqcup_{i\in\omega} \varphi_i)\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]\right)$$

Let us assume $\forall i \in \omega, P(\varphi_i)$ and $(\bigsqcup_{i\in\omega} \varphi_i)\sigma = \sigma'$. We want to prove that $\sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]$. By $(\bigsqcup_{i\in\omega} \varphi_i)\sigma = \sigma'$ we have:

$$\exists k.\varphi_k\sigma = \sigma'$$

then we can conclude the thesis by $P(\varphi_l)$.
We can now use the computational induction:

$$\frac{P \text{ inclusive} \quad P(\bot) \quad \forall\varphi.P(\varphi) \Rightarrow P(\Gamma\,\varphi)}{P(\mathrm{fix}\,\Gamma)}$$

as $P(\mathrm{fix}\,\Gamma) = P(\mathscr{C}\,[\![w]\!])$.

- $P(\bot)$ *obviously since* $\bot\sigma = \sigma'$ *is always false.*
- $P(\varphi) \overset{?}{\Longrightarrow} P(\Gamma\varphi)$
  Let us assume
  $$P(\varphi) = \left(\forall\sigma.(\varphi\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[{}^{0}/_{x}])\right)$$
  we will show
  $$P(\Gamma\varphi) = \left(\Gamma\varphi\sigma = \sigma' \overset{?}{\Longrightarrow} \sigma x \geq 0 \wedge \sigma' = \sigma[{}^{0}/_{x}]\right)$$
  we assume the premise $\Gamma\varphi\sigma = \sigma'$ *and prove that* $\sigma x \geq 0 \wedge \sigma' = \sigma[{}^{0}/_{x}]$.
  $$\Gamma\varphi\sigma = \left(\sigma x \neq 0 \rightarrow \varphi\sigma[{}^{\sigma x-1}/_{x}], \sigma\right)$$
  if $\sigma x = 0$*: We have*
  $$(\sigma x \neq 0 \rightarrow \varphi\sigma[{}^{\sigma x-1}/_{x}], \sigma) = \sigma$$
  therefore $\sigma' = \sigma$ *and trivially*
  $$\sigma x = 0 \geq 0 \quad \sigma' = \sigma = \sigma[{}^{0}/_{x}].$$
  if $\sigma x \neq 0$*, We have*
  $$(\sigma x \neq 0 \rightarrow \varphi\sigma[{}^{\sigma x-1}/_{x}], \sigma) = \varphi\sigma[{}^{\sigma x-1}/_{x}].$$
  Let $\sigma'' = \sigma[{}^{\sigma x-1}/_{x}]$*. We exploit* $P(\varphi)$ *for the argument* σ''*:*
  $$\varphi\underbrace{\sigma[{}^{\sigma x-1}/_{x}]}_{\sigma''} = \sigma' \Rightarrow \sigma'' x \geq 0 \wedge \sigma' = \sigma''[{}^{0}/_{x}]$$
  we have:
  $$\sigma'' x \geq 0 \Leftrightarrow \sigma[{}^{\sigma x-1}/_{x}]x \geq 0 \Leftrightarrow \sigma x - 1 \geq 0 \Leftrightarrow \sigma x \geq 1 \Rightarrow \sigma x \geq 0$$
  $$\sigma' = \sigma''[{}^{0}/_{x}] = \sigma[{}^{\sigma x-1}/_{x}][{}^{0}/_{x}] = \sigma[{}^{0}/_{x}]$$
  And the consequence holds also in this case.
  Since we have proved that P is inclusive and that $P(\bot)$ *and* $P(\varphi) \Longrightarrow P(\Gamma\varphi)$*, by computational induction the property P holds for the fixpoint* $\mathrm{fix}(\Gamma)$ *and thus for the semantics of the command w as* $\mathscr{C}\,[\![w]\!] = \mathrm{fix}(\Gamma)$.

Part II.

HOFL language

6. Operational Semantics of HOFL

In the previous chapters we studied an imperative language called IMP. In this chapter we focus our attention on functional languages. In particular, we introduce HOFL, a simple higher-order functional language which allows the explicit construction of types. We adopt a *lazy* evaluation semantics, which corresponds to a *call-by-name* strategy, namely parameters are passed without evaluating them.

6.1. HOFL

As done for IMP we will start by introducing the syntax of HOFL. In IMP we have only three types: Aexp, Bexp and Com. Since IMP does not allow to construct other types explicitly, we embedded these types in its syntax. HOFL, instead, provides a method to define a variety of types, so we define the *pre-terms* first, then we introduce the concept of typed terms, namely the well-formed terms of our language. Due to the context-sensitive constraints induced by the types, it is possible to see that well-formed terms could not be defined by a syntax expressed in a context-free format.

$t ::=$	$x \mid$	variables
	$n \mid$	constants
	$t_0 + t_1 \mid t_0 - t_1 \mid t_0 \times t_1 \mid$	arithmetic operators
	$\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \mid$	conditional (it reads $\textbf{if } t = 0 \textbf{ then } t_0 \textbf{ else } t_1$)
	$(t_0, t_1) \mid \textbf{fst}(t) \mid \textbf{snd}(t) \mid$	pairing and projection operators
	$\lambda x.t \mid (t_0\ t_1) \mid$	function abstraction and application
	$\textbf{rec } x.t$	recursive definition

As usually we have variables, constants, arithmetic operators, conditional operator and function application and definition. Moreover in HOFL we have the constructs of pair and of recursion. Furthermore we have the operations which allow to project the pair on a single component: **fst**, which extracts the first component and **snd** which extracts the second component. Recursion allows to define recursive terms, namely $\textbf{rec } x.t$ defines a term t which can contain variable x, which in turn can be replaced by t.

6.1.1. Typed Terms

Using the definition of pre-term given in the previous section, we can construct ill-formed terms that make little sense (for example we can construct terms like $(\lambda x.0) + 1$). To avoid these constructions we introduce the concepts of *type* and *typed term*. A type is a term constructed by using the following grammar:

$$\tau ::= int \mid \tau * \tau \mid \tau \to \tau$$

So we allow constant type *int*, the pair type and the function type. Using these constructors we are allowed to define infinitely many types, like $(int * int) \to int$ and $int \to (int * (int \to int))$. Now we define the rule system which allows to say if a pre-term of HOFL is well-formed (i.e. if we can associate a type to the pre-term).

Variables $x : type(x) = \hat{x}$ where *type* is a function which assigns a type to each variable.

Arithmetic operators and conditional

$$n : int \qquad \frac{t_0 : int \quad t_1 : int}{t_0 \text{ op } t_1 : int} \text{ with op} = +, -, \times \qquad \frac{t : int \quad t_0 : \tau \quad t_1 : \tau}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 : \tau}$$

Pairings

$$\frac{t_0 : \tau_0 \quad t_1 : \tau_1}{(t_0, t_1) : \tau_0 * \tau_1} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{fst}(t) : \tau_0} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{snd}(t) : \tau_1}$$

Functions

$$\frac{x : \tau_0 \quad t : \tau_1}{\lambda x.t : \tau_0 \to \tau_1} \qquad \frac{t_1 : \tau_0 \to \tau_1 \qquad t_0 : \tau_0}{(t_1\ t_0) : \tau_1}$$

Recursion

$$\frac{x : \tau \quad t : \tau}{\mathbf{rec}\ x.t : \tau}$$

Definition 6.1 (Well-Formed Terms of HOFL)
Let t be a pre-term of HOFL, we say that t is well-formed *if there exists a type τ such that $t : \tau$.*

In Section 5.1 we defined the concepts of free variables and substitution for λ-calculus. Now we define the same concepts for HOFL. So by structural recursion we define the set of free-variables of HOFL terms as follows:

$$\begin{aligned}
&fv(n) = \varnothing \\
&fv(x) = \{x\} \\
&fv(t_0 \text{ op } t_1) = fv(t_0) \cup fv(t_1) \\
&fv(\mathbf{if}\ t\ \mathbf{then}\ t_0\ \mathbf{else}\ t_1) = fv(t) \cup fv(t_0) \cup fv(t_1) \\
&fv((t_0, t_1)) = fv(t_0) \cup fv(t_1) \\
&fv(\mathbf{fst}(t)) = fv(\mathbf{snd}(t)) = fv(t) \\
&fv(\lambda x.t) = fv(t) \backslash \{x\} \\
&fv((t_0\ t_1)) = fv(t_0) \cup fv(t_1) \\
&fv(\mathbf{rec}\ x.t) = fv(t) \backslash \{x\}
\end{aligned}$$

Finally as done for λ-calculus we define the substitution operator on HOFL:

$$\begin{aligned}
n[^t/_x] &= n \\
y[^t/_x] &= \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \\
(t_0 \text{ op } t_1)[^t/_x] &= t_0[^t/_x] \text{ op } t_1[^t/_x] \qquad \text{(Analogously for } \mathbf{if\ then\ else}\text{, pairing, } \mathbf{fst}, \mathbf{snd}\text{)} \\
(\lambda y.t')[^t/_x] &= \lambda z.t'[^z/_y][^t/_x] \qquad \text{where } z \notin fv(\lambda y.t') \cup fv(t) \cup \{x\} \qquad \text{(Analogously for recursion)}
\end{aligned}$$

Note that in the last rule we performed an α-conversion before the substitution. This ensures that the variables in t' are not bound by the substitution operation. As discussed in Section 5.1 the substitution is well-defined if we consider the terms up to α-equivalence (i.e. up to the name equivalence). Obviously we would like to extend these concepts to typed terms. So we are interested in understanding how substitution and α-conversion interact with typing. We have the following results:

Theorem 6.2 (Substitution Respects Types)
Let $t : \tau$ a term of type τ and let $t', y : \tau'$ be two terms of type τ' then we have

$$t[^{t'}/_y] : \tau$$

Proof. We leave as an exercise to prove the property by induction on the derivation, after having proved that for the special case where $t' = z : \tau'$ we have $t[^z/_y] : \tau$ with a derivation that has the same length as a derivation of $t : \tau$. □

Note that our type system is very simple. Indeed it does not allow to construct useful types, such as recursive,

parametric, polymorphic or abstract types. These limitations imply that we cannot construct many useful terms. For instance, lists of integer numbers of variable length are not typable in our system. Here we show an interesting term with recursion which is not typable.

Example 6.3
Now we define a pre-term t which, when applied to 0*, should define the list of all even numbers, where:*

$$t = \textbf{rec}\ \ p.\lambda x.\ (x, (p\ (x+2)))$$

Intuitively, the term t 0 *should represent the infinite list of even numbers:*

$$(0, (2, (4, \ldots)))$$

Let us show that this term is not typable:

$$\begin{aligned}
t = \textbf{rec}\ \ p.\lambda x.\ (x, (p\ (x+2))) : \tau \quad & \nwarrow_{\hat{p}=\tau} \\
\lambda x.\ (x, (p\ (x+2))) : \tau \quad & \nwarrow_{\tau=\tau_1\to\tau_2, \hat{x}=\tau_1} \\
(x, (p\ (x+2))) : \tau_2 \quad & \nwarrow_{\tau_2=\tau_3*\tau_4} \\
x : \tau_3,\ (p\ (x+2)) : \tau_4 \quad & \nwarrow_{\hat{x}=\tau_3} \\
(p\ (x+2)) : \tau_4 \quad & \nwarrow \\
p : \tau_5 \to \tau_4,\ (x+2) : \tau_5 \quad & \nwarrow_{\hat{p}=\tau_5\to\tau_4} \\
(x+2) : \tau_5 \quad & \nwarrow_{\tau_5=int} \\
x : int \quad & \nwarrow_{\hat{x}=int}
\end{aligned}$$

So we have:

$$\hat{x} = \tau_1 = \tau_3 = int \qquad \tau_2 = \tau_3 * \tau_4 = int * \tau_4 \qquad \tau = \tau_1 \to \tau_2 = int \to (int * \tau_4) \qquad \tau_5 = int$$

That is:

$$\hat{p} = \tau = int \to (int * \tau_4)$$

and

$$\hat{p} = \tau_5 \to \tau_4 = int \to \tau_4$$

Thus:

$$int * \tau_4 = \tau_4 \text{ which is absurd}$$

The above argument is represented more concisely below:

$$t = \textbf{rec}\ \ p.\lambda \underset{int}{x}.\ (\underset{int}{x}, (\underset{int\to\tau}{p}\ \ (\underset{int}{x}+2)))$$

$$\underbrace{\underbrace{\underbrace{\underbrace{}_{int}}_{int\to\tau}}_{\tau}}_{int*\tau}\ \ \underbrace{\qquad\qquad}_{int\to int*\tau = int\to\tau \Rightarrow \tau = int*\tau}$$

So we have no solutions, and the term is not an HOFL term, no matter what is the value of $\hat{p}$ and $\hat{x}$.

6.1.2. Typability and Typechecking

As we said in the last section we will give semantics only to well-formed terms, namely terms which have a type in our type system. Therefore we need an algorithm to say if a term is well-formed. In this section we will present two different solutions to the typability problem, introduced by Church and by Curry, respectively.

6.1.2.1. Church Type Theory

In Church type theory we explicitly associate a type to each variable and deduce the type of each term by structural recursion (i.e. by using the rules in a bottom-up fashion).

Let us show how it works by typing the factorial function:

Example 6.4 (Factorial with Church Types)
Let $x : int$ and $f : int \to int$. So we can type all the subterms:

$$\mathbf{fact} \stackrel{\text{def}}{=} \mathbf{rec}\ \underset{int\to int}{f}\ .\lambda\ \underset{int}{x}\ .\mathbf{if}\ \underset{int}{x}\ \mathbf{then}\ \underset{int}{1}\ \mathbf{else}\ \underset{int}{x} \times (\ \underbrace{\underset{int\to int}{f}\ (\underbrace{\underset{int}{x} - \underset{int}{1}}_{int})}_{int}\ ) \qquad : int \to int$$

6.1.2.2. Curry Type Theory

In Curry types we do not need to explicitly declare the type of each variable. We will use the inference rules to calculate type equations (i.e. equations which have types as variables) whose solutions will be all the possible types of the term. This means that the result will be a set of types associated to the typed term. The surprising fact is that this set can be represented as all the instantiation of a single term with variables, where one instantiation is obtained by replacing each variable with any type. We call this term with variables the *principal type* of the term. This construction is made by using the rules in a goal oriented fashion.

Example 6.5 (Identity)
Let us consider the identity function:

$$\lambda x.\ x$$

By using the type system we have:

$$\begin{aligned} \lambda x.\ x : \tau \quad & \nwarrow_{\tau=\tau_1\to\tau_2,\ \hat{x}=\tau_1} \\ x : \tau_2 \quad & \nwarrow_{\hat{x}=\tau_2} \\ \square & \end{aligned}$$

So we have $\hat{x} = \tau_1 = \tau_2$ and the principal type is $\lambda x.\ x : \tau_1 \to \tau_1$. Now each solution of the type equation will be an identity function for a specified type. For example if we set $\tau_1 = int$ we have $\tau = int \to int$.

As we will see in the next example and as we already saw in example 6.3, the typing problem reduces to that of resolving a system of type equations. We will also introduce a standard way to find a solution of the system (if it exists).

Example 6.6 (Non-typable term of HOFL)
Let us consider the following function, which computes the factorial without using recursion.

begin
$\quad$ **fact**$(f, x) \stackrel{\text{def}}{=}$ **if** $x = 0$ **then** 1 **else** $x \times f(f, x - 1)$
$\quad$ **fact**(**fact**, n)
end

It can be written in HOFL as follows:

$$\mathbf{fact} \stackrel{\text{def}}{=} \lambda\ \underset{\tau*int}{y}\ .\ \mathbf{if}\ \underbrace{\mathbf{snd}(\ \underset{\tau*int}{y}\ )}_{int}\ \mathbf{then}\ \underset{int}{1}\ \mathbf{else}\ \underbrace{\mathbf{snd}(\ \underset{\tau*int}{y}\ )}_{int} \times (\ \underbrace{\mathbf{fst}(\ \underset{\tau*int}{y}\ )}_{\tau=\tau*int\to int}\ (\underbrace{\mathbf{fst}(\ \underset{\tau*int}{y}\ )}_{\tau},\ \underbrace{\mathbf{snd}(\ \underset{\tau*int}{y}\ )}_{int})\ - \underset{int}{1}\ )$$

We conclude $\mathbf{fst}(y) : \tau$ *and* $\mathbf{fst}(y) : \tau * int \to int$. *Thus we have* $\tau = \tau * int \to int$ *which is an equation which has no solution.*

We present an algorithm, called *unification*, to solve general systems of type equations.
We start from a system of equation of the type:

$$\begin{cases} t_1 = t'_1 \\ t_2 = t'_2 \\ \dots \end{cases}$$

where t's are type terms, with type variables denoted by τ's, then we apply iteratively in any order the following steps:

1) We eliminate all the equations like $\tau = \tau$.

2) For each equation of the form $f(t_1, \dots, t_n) = f'(t'_1, \dots, t'_m)$: if $f \neq f'$, then the system has no solutions. if $f = f'$ then $m = n$ so we must have:

$$t_1 = t'_1, t_2 = t'_2, \dots, t_n = t'_n$$

Then we replace the original equation with these.

3) For each equation of the type $\tau = t$, $t \neq \tau$: we let $\tau \stackrel{\text{def}}{=} t$ and we replace each occurrence of τ with t in all the other equations. If $\tau \in t$ then the system has no solutions.

Eventually, either the system is recognized as unsolvable, or all the variables in the original equations are assigned to solution terms. Note that the order of the step executions can affect the complexity of the algorithm but not the solution. The best execution strategies yield a complexity linear or quasi linear with the size of the original system of equations.

Example 6.7
Let us now apply the algorithm to the previous example:

$$\tau = \tau * int \to int$$

step 3 fails since type variable τ *appears in the right hand side.*

6.2. Operational Semantics of HOFL

We are now ready to present the (lazy) operational semantics of HOFL. Unlike IMP the operational semantics of HOFL is a simple manipulation of terms. This means that the operational semantics of HOFL defines a method to calculate the *canonical form* of a given closed term of HOFL. Canonical forms are closed terms, which we will consider the results of calculations (i.e. values). For each type we will define a set of terms in canonical form by taking a subset of terms which reasonably represent the values of that type. As shown in the previous section, HOFL has three type constructors: integer, pair and arrow. Obviously, terms which represent the integers are the canonical forms for the integer type. For pair type we will take pairs of terms (note that this choice is arbitrary, for example we could take pairs of terms in canonical form). Finally, since HOFL is a higher-order language, functions are values. So is quite natural to take all abstractions as canonical forms for the arrow type.

Definition 6.8 (Canonical forms)
Let us define a set C_τ *of canonical forms for each type* τ *as follows:*

$$\frac{}{n \in C_{int}}$$

$$\frac{t_0 : \tau_0 \quad t_1 : \tau_1 \quad t_0, t_1 \text{ closed}}{(t_0, t_1) \in C_{\tau_0 * \tau_1}}$$

$$\frac{\lambda x.t : \tau_0 \to \tau_1 \quad \lambda x.t \text{ closed}}{\lambda x.t \in C_{\tau_0 \to \tau_1}}$$

We now define the rules of the operational semantics, these rules define an evaluation relation:

$$t \longrightarrow c$$

where t is a well-formed closed term of HOFL and c is its canonical form. So we define:

$$\frac{}{c \to c} \qquad \frac{t_0 \to n_0 \quad t_1 \to n_1}{t_0 \text{ op } t_1 \to n_0 \underline{\text{op}}\, n_1} \qquad \frac{t \to 0 \quad t_0 \to c_0}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_0} \qquad \frac{t \to n \quad n \neq 0 \quad t_1 \to c_1}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_1}$$

The first rule is an axiom which allows to evaluate terms already in canonical form. For the arithmetic operators the semantics is obviously the simple application of the correspondent meta-operator as well as in IMP. Only, here we distinguish between operator and meta operator by underlying the latter. For the "if-then-else", since we have no boolean values, we use the convention that **if** t **then** t_0 **else** t_1 stands for **if** $t = 0$ **then** t_0 **else** t_1, so $t \neq 0$ means t is false and $t = 0$ means t is true.

Let us now consider the pairing. Obviously, since we consider pairs as values, we have no rules for the simple pair. We have instead two rules for projections:

$$\frac{t \to (t_0, t_1) \quad t_0 \to c_0}{\textbf{fst}(t) \to c_0} \qquad \frac{t \to (t_0, t_1) \quad t_1 \to c_1}{\textbf{snd}(t) \to c_1}$$

For function application, we give a lazy operational semantics, this means that we do not evaluate the parameters before replacing them with the copy rule in the function body. If the argument is actually needed it may be later evaluated several times (every times it is used).

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[{}^{t_0}/_x] \to c}{(t_1 \quad t_0) \to c} \quad \text{(lazy)}$$

So we replace each occurrence of x in t_1' with a copy of the parameter.
Let us consider the *eager* counterpart of this rule. Unlike the lazy semantics, the eager semantics evaluates the parameters only once and during the application. Note that these two types of evaluation are not equivalent. If the evaluation of the argument does not terminate, and it is not needed, the lazy rule will guarantee convergence, while the eager rule will diverge.

$$\frac{t_1 \to \lambda x.t_1' \quad t_0 \to c_0 \quad t_1'[{}^{c_0}/_x] \to c}{(t_1 \quad t_0) \to c} \quad \text{(eager)}$$

Finally we have a rule for recursive terms:

$$\frac{t[{}^{\textbf{rec } x.t}/_x] \to c}{\textbf{rec } x.t \to c}$$

Let us see an example which illustrates how rules are used to evaluate a function application.

Example 6.9 (Factorial)
Let us consider the factorial function in HOFL:

$$\textbf{fact} = \textbf{rec } f.\lambda x.\, \textbf{if } x \textbf{ then } 1 \textbf{ else } x \times (f(x-1))$$

As we said in the previous examples $fact$ is closed and typable. So we can calculate its canonical form by using the last rule:

$$\textbf{fact} \longrightarrow \lambda x.\,\textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x \times (\textbf{fact}(x-1))$$

Now we can apply this function to a specific value and calculate the result as usual:

$$\begin{array}{rll}
(\textbf{fact}\,2) \to c & \nwarrow & \textbf{fact} \to \lambda x.t \quad t[^2/_x] \to c \\
& \nwarrow & \lambda x.\,\textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x \times (\textbf{fact}(x-1)) \to \lambda x.t \quad t[^2/_x] \to c \\
\nwarrow_{t=\textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x\times\textbf{fact}(x-1)} & & \textbf{if}\ 2\ \textbf{then}\ 1\ \textbf{else}\ 2 \times (\textbf{fact}(2-1)) \to c \\
& \nwarrow & 2 \times (\textbf{fact}(2-1)) \to c \\
\nwarrow_{c=c_1\times c_2} & & 2 \to c_1 \quad (\textbf{fact}(2-1)) \to c_2 \\
\nwarrow_{c_1=2} & & \textbf{fact} \to \lambda x.t \quad \underbrace{t[^{2-1}/_x]}_{2-1\ \textit{not evaluated}} \to c_2 \\
& \nwarrow & \textbf{if}(2-1)\ \textbf{then}\ 1\ \textbf{else}(2-1) \times (\textbf{fact}((2-1)-1)) \to c_2 \\
& \nwarrow & 2-1 \to n \quad n \neq 0 \quad (2-1) \times \textbf{fact}((2-1)-1) \to c_2 \\
\nwarrow_{n=n_1-n_2} & & 2 \to n_1 \quad 1 \to n_2 \quad 2-1 \to n_1 - n_2 \quad n_1 - n_2 \neq 0 \\
& & (2-1) \times \textbf{fact}((2-1)-1) \to c_2 \\
\nwarrow_{n_1=2 \quad n_2=1} & & 2-1 \to 1 \quad 1 \neq 0 \quad (2-1) \times \textbf{fact}((2-1)-1) \to c_2 \\
\nwarrow_{c_2=c_3\times c_4} & & 2-1 \to c_3 \quad \textbf{fact}((2-1)-1) \to c_4 \\
\nwarrow_{c_3=1} & & \textbf{if}(2-1)-1\ \textbf{then}\ 1\ \textbf{else}((2-1)-1) \times (\textbf{fact}(((2-1)-1)-1)) \to c_4 \\
\nwarrow^{*} & & (2-1)-1 \to 0 \qquad 1 \to c_4 \\
\nwarrow_{c_4=1\ c_3=1\ c_2=c_3\times c_4=1\ c=c_1\times c_2\times 1=2} & & \square
\end{array}$$

Example 6.10 (Non-equivalence of lazy and eager evaluations)
The aim of this example is to illustrate the difference between lazy and eager semantics.

- ***Lazy evaluation***
  Lazy evaluation evaluates the terms only if needed: if a parameter is never used in a function or in a specific instance of a function it will never be evaluated. Let us show an example:

$$\begin{array}{rll}
((\lambda x : int.3)\,\textbf{rec}\,y : int.y) : int \to c & \nwarrow & \lambda x : int.3 \to \lambda x.t \quad t[^{\textbf{rec}\,y.y}/_x] \to c \\
& \nwarrow_{t=3} & 3[^{\textbf{rec}\,y.y}/_x] \to c \\
& \nwarrow_{c=3} & \square
\end{array}$$

  So although the term $\textbf{rec}\,y.y$ *as no canonical form the application can be evaluated.*

- ***Eager evaluation***
  On the contrary in the eager semantics this term has no canonical form since the parameter is evaluated before the application:

$$\begin{array}{rll}
((\lambda x : int.3)\,\textbf{rec}\,y : int.y) : int \to c & \nwarrow & \lambda x : int.3 \to \lambda x.t \quad \textbf{rec}\,y.y \to c_1 \quad t[^{c_1}/_x] \to c \\
& \nwarrow_{t=3} & \textbf{rec}\,y.y \to c_1 \quad 3[^{c_1}/_x] \to c \\
& \nwarrow & \textbf{rec}\,y.y \to c_1 \quad 3[^{c_1}/_x] \to c \\
& \nwarrow & \ldots
\end{array}$$

  So the evaluation does not terminate. However if the parameter of a function is used n times, the parameter would be evaluated n times (at most) in the lazy semantics and only once in the eager case.

Theorem 6.11

i) If $t \to c$ and $t \to c'$ then $c = c'$ (if a canonical form exists it is unique)

ii) if $t \to c$ and $t : \tau$ then $c : \tau$ (the evaluation relation respects the types)

Proof. We prove the property i)

$$P(t \to c) \stackrel{\text{def}}{=} \forall c'\ t \to c' \Rightarrow c = c'$$

by rule induction. Let us show only the function rule, the remainder of the proof of the theorem is left as exercise. We have the rule:

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[{}^{t_0}/_x] \to c}{(t_1 \quad t_0) \to c}$$

We will show:

$$P((t_1 \quad t_0) \longrightarrow c) \stackrel{\text{def}}{=} \forall c'\ (t_1 \quad t_0) \longrightarrow c' \Rightarrow c = c'$$

As usually let us assume the premise:

$$(t_1 \quad t_0) \longrightarrow c'$$

And the inductive hypothesis:

- $P(t_1 \to \lambda x.t_1') \stackrel{\text{def}}{=} \forall c'\ t_1 \longrightarrow c' \Rightarrow \lambda x.t_1' = c'$
- $P(t_1'[{}^{t_0}/_x] \to c) \stackrel{\text{def}}{=} \forall c'\ t_1'[{}^{t_0}/_x] \longrightarrow c' \Rightarrow c = c'$

From $(t_1 \quad t_0) \to c'$ by goal reduction:

- $t_1 \to \lambda \overline{x}.\overline{t_1}'$
- $\overline{t_1}'[{}^{t_0}/_{\overline{x}}] \to c'$

then we have by inductive hypothesis:

- $\lambda x.t_1' = \lambda \overline{x}.\overline{t_1}'$
- $t_1'[{}^{t_0}/_x] = \overline{t_1}'[{}^{t_0}/_{\overline{x}}]$

So we have $c = c'$ □

Part V.

Appendices

A. Summary

A.1. Induction rules 3.1.2

A.1.1. Noether

Let $\prec$ be a well-founded relation over the set A and let P be a unary predicate over A. Then:

$$\frac{\forall a \in A.\ (\forall b \prec a.\ P(b)) \rightarrow P(a)}{\forall a \in A.P(a)}$$

A.1.2. Weak Mathematical Induction 3.1.3

Let P be a unary predicate over ω.

$$\frac{P(0) \quad \forall n \in \omega.(P(n) \rightarrow P(n+1))}{\forall n \in \omega.P(n)}$$

A.1.3. Strong Mathematical Induction 3.1.4

Let P be a unary predicate over ω.

$$\frac{P(0) \quad \forall n \in \omega.(\forall i \leq n.P(i)) \rightarrow P(n+1)}{\forall n \in \omega.P(n)}$$

A.1.4. Structural Induction 3.1.5

Let Σ be a signature, T_Σ be the set of terms over Σ and P be a property defined on T_Σ.

$$\frac{\forall t \in T_\Sigma.\ (\forall t' < t.\ P(t')) \Rightarrow P(t)}{\forall t \in T_\Sigma.\ P(t)}$$

A.1.5. Derivation Induction 3.1.6

Let R be a set of inference rules and D the set of derivations defined on R. We define:

$$\frac{\forall \{x_1, \ldots, x_n\}/y \in R.\ (P(d_1) \wedge \ldots \wedge P(d_n)) \Rightarrow P(\{d_1, \ldots, d_n\}/y)}{\forall d \in D.\ P(d)}$$

where $d_1, \ldots, d_n$ are derivation for $x_1, \ldots, x_n$.

A.1.6. Rule Induction 3.1.7

Let R be a set of rules and I_R the set of theorems of R.

$$\frac{\forall (X/y) \in R \quad (X \subseteq I_R \quad \forall x \in X.\ P(x)) \Longrightarrow P(y)}{\forall x \in I_R.P(x)}$$

A.1.7. Computational Induction 5.4

Let P be a property, $(D, \sqsubseteq)$ a $CPO_\perp$ and F a monotone, continuous function on it. We define:

$$\frac{P \text{ inclusive} \quad \perp \in P \quad \forall d \in D.d \in P \Longrightarrow F(d) \in P}{fix(F) \in P}$$

A.2. IMP 2

A.2.1. IMP Syntax 2.1

$$\begin{array}{rcl} a & ::= & n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \\ b & ::= & v \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \vee b_1 \mid b_0 \wedge b_1 \\ c & ::= & \textbf{skip} \mid x := a \mid c_0; c_1 \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b \textbf{ do } c \end{array}$$

A.2.2. IMP Operational Semantics 2.2

A.2.2.1. IMP Arithmetic Expressions

$$\frac{}{\langle x, \sigma\rangle \to \sigma(x)}\ (\text{ide}) \qquad \frac{}{\langle n, \sigma\rangle \to n}\ (\text{num}) \qquad \frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 + a_1, \sigma\rangle \to n_0 + n_1}\ (\text{sum})$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 - a_1, \sigma\rangle \to n_0 - n_1}\ (\text{dif}) \qquad \frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \times a_1, \sigma\rangle \to n_0 \times n_1}\ (\text{prod})$$

A.2.2.2. IMP Boolean Expressions

$$\frac{}{\langle v, \sigma\rangle \to v}\ (\text{bool}) \qquad \frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 = a_1, \sigma\rangle \to (n_0 = n_1)}\ (\text{equ}) \qquad \frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \leq a_1, \sigma\rangle \to (n_0 \leq n_1)}\ (\text{leq})$$

$$\frac{\langle b, \sigma\rangle \to v}{\langle \neg b, \sigma\rangle \to \neg v}\ (\text{not}) \qquad \frac{\langle b_0, \sigma\rangle \to v_0 \quad \langle b_1, \sigma\rangle \to v_1}{\langle b_0 \vee b_1, \sigma\rangle \to (v_0 \vee v_1)}\ (\text{or}) \qquad \frac{\langle b_0, \sigma\rangle \to v_0 \quad \langle b_1, \sigma\rangle \to v_1}{\langle b_0 \wedge b_1, \sigma\rangle \to (v_0 \wedge v_1)}\ (\text{and})$$

A.2.2.3. IMP Commands

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma}\ (\text{skip}) \qquad \frac{\langle a, \sigma\rangle \to m}{\langle x := a, \sigma\rangle \to \sigma[{}^{m}/_{x}]}\ (\text{assign})$$

$$\frac{\langle c_0, \sigma\rangle \to \sigma'' \quad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c_0; c_1, \sigma\rangle \to \sigma'}\ (\text{seq}) \qquad \frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c_0, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}\ (\text{iftt}) \qquad \frac{\langle b, \sigma\rangle \to \textbf{false} \quad \langle c_1, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}\ (\text{ifff})$$

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'}\ (\text{whtt}) \qquad \frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma}\ (\text{whff})$$

A.2.3. IMP Denotational Semantics 5

A.2.3.1. IMP Arithmetic Expressions $\mathscr{A} : Aexpr \to (\Sigma \to \mathbb{N})$

$$\begin{array}{rcl} \mathscr{A}\,[\![n]\!]\,\sigma & = & n \\ \mathscr{A}\,[\![x]\!]\,\sigma & = & \sigma x \\ \mathscr{A}\,[\![a_0 + a_1]\!]\,\sigma & = & (\mathscr{A}\,[\![a_0]\!]\,\sigma) + (\mathscr{A}\,[\![a_1]\!]\,\sigma) \\ \mathscr{A}\,[\![a_0 - a_1]\!]\,\sigma & = & (\mathscr{A}\,[\![a_0]\!]\,\sigma) - (\mathscr{A}\,[\![a_1]\!]\,\sigma) \\ \mathscr{A}\,[\![a_0 \times a_1]\!]\,\sigma & = & (\mathscr{A}\,[\![a_0]\!]\,\sigma) \times (\mathscr{A}\,[\![a_1]\!]\,\sigma) \end{array}$$

A.2.3.2. IMP Boolean Expressions $\mathscr{B} : Bexpr \to (\Sigma \to \mathbb{B})$

$$\begin{aligned}
\mathscr{B}[\![v]\!]\sigma &= v \\
\mathscr{B}[\![a_0 = a_1]\!]\sigma &= (\mathscr{A}[\![a_0]\!]\sigma) = (\mathscr{A}[\![a_1]\!]\sigma) \\
\mathscr{B}[\![a_0 \leq a_1]\!]\sigma &= (\mathscr{A}[\![a_0]\!]\sigma) \leq (\mathscr{A}[\![a_1]\!]\sigma) \\
\mathscr{B}[\![\neg b_0]\!]\sigma &= \neg(\mathscr{B}[\![b]\!]\sigma) \\
\mathscr{B}[\![b_0 \vee b_1]\!]\sigma &= (\mathscr{B}[\![b_0]\!]\sigma) \vee (\mathscr{B}[\![b_1]\!]\sigma) \\
\mathscr{B}[\![b_0 \wedge b_1]\!]\sigma &= (\mathscr{B}[\![b_0]\!]\sigma) \wedge (\mathscr{B}[\![b_1]\!]\sigma)
\end{aligned}$$

A.2.3.3. IMP Commands $\mathscr{C} : Com \to (\Sigma \rightharpoonup \Sigma)$

$$\begin{aligned}
\mathscr{C}[\![\mathbf{skip}]\!]\sigma &= \sigma \\
\mathscr{C}[\![x := a]\!]\sigma &= \sigma\left[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}\right] \\
\mathscr{C}[\![c_0; c_1]\!]\sigma &= \mathscr{C}[\![c_1]\!]^*(\mathscr{C}[\![c_0]\!]\sigma) \\
\mathscr{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]\sigma &= \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma \\
\mathscr{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] &= fix\,\Gamma = \bigsqcup_{n\in\omega} \Gamma^n(\perp_{\Sigma\to\Sigma_\perp})
\end{aligned}$$

where $\Gamma : (\Sigma \to \Sigma_\perp) \to \Sigma \to \Sigma_\perp$ is defined as follows:

$$\Gamma\varphi\sigma \quad = \quad \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![c]\!]\sigma), \sigma$$

A.3. HOFL 6.1

A.3.1. HOFL Syntax 6.1

$t ::=$	$x \mid$		variables
	$n \mid$		constants
	$t_0 + t_1 \mid t_0 - t_1 \mid t_0 \times t_1 \mid$		arithmetic operators
	$\mathbf{if}\ t\ \mathbf{then}\ t_0\ \mathbf{else}\ t_1 \mid$		conditional
	$(t_0, t_1) \mid \mathbf{fst}(t) \mid \mathbf{snd}(t) \mid$		pairing and projection operators
	$\lambda x.t \mid (t_0\ t_1) \mid$		function abstraction and application
	$\mathbf{rec}\ x.t$		recursive definition

A.3.2. HOFL Types 6.1.1

$$\tau ::= int \mid \tau * \tau \mid \tau \to \tau$$

A.3.3. HOFL Typing Rules 6.1.1

$$n : int \qquad \frac{t_0 : int \quad t_1 : int}{t_0\ \mathrm{op}\ t_1 : int}\ \text{with op} = +, -, \mathrm{x} \qquad \frac{t_0 : int \quad t_1 : \tau \quad t_2 : \tau}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : \tau}$$

$$\frac{t_0 : \tau_0 \quad t_1 : \tau_1}{(t_0, t_1) : \tau_0 * \tau_1} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{fst}(t) : \tau_0} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{snd}(t) : \tau_1}$$

$$\frac{x : \tau_0 \quad t : \tau_1}{\lambda x.t : \tau_0 \to \tau_1} \qquad \frac{t_1 : \tau_0 \to \tau_1 \qquad t_0 : \tau_0}{(t_1\ t_0) : \tau_1}$$

$$\frac{x : \tau \quad t : \tau}{\mathbf{rec}\ x.t : \tau}$$

A.3.4. HOFL Operational Semantics 6.2

A.3.4.1. HOFL Canonical Forms

$$\frac{\varnothing}{n \in C_{int}} \qquad \frac{t_0 : \tau_0 \quad t_1 : \tau_1 \quad t_0, t_1 \text{ closed}}{(t_0, t_1) \in C_{\tau_0 * \tau_1}} \qquad \frac{\lambda x.t : \tau_0 \to \tau_1 \quad \lambda x.t \text{ closed}}{\lambda x.t \in C_{\tau_0 \to \tau_1}}$$

A.3.4.2. HOFL Axiom

$$\frac{}{c \to c}$$

A.3.4.3. HOFL Arithmetic and Conditional Expressions

$$\frac{t_0 \to n_0 \quad t_1 \to n_1}{t_0 \text{ op } t_1 \to n_0 \underline{op} n_1} \qquad \frac{t \to 0 \quad t_0 \to c_0}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_0} \qquad \frac{t \to n \quad n \neq 0 \quad t_1 \to c_1}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_1}$$

A.3.4.4. HOFL Pairing Rules

$$\frac{t \to (t_0, t_1) \quad t_0 \to c_0}{\textbf{fst}(t) \to c_0} \qquad \frac{t \to (t_0, t_1) \quad t_1 \to c_1}{\textbf{snd}(t) \to c_1}$$

A.3.4.5. HOFL Function Application

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[{}^{t_0}/_x] \to c}{(t_1 \quad t_0) \to c} \quad \text{(lazy)}$$

$$\frac{t_1 \to \lambda x.t_1' \quad t_0 \to c_0 \quad t_1'[{}^{c_0}/_x] \to c}{(t_1 \quad t_0) \to c} \quad \text{(eager)}$$

A.3.4.6. HOFL Recursion

$$\frac{t[{}^{\textbf{rec } x.t}/_x] \to c}{\textbf{rec } x.t \to c}$$

A.3.5. HOFL Denotational Semantics 8 $[\![t : \tau]\!] : Env \longrightarrow (V_\tau)_\perp$

$$\begin{aligned}
[\![n]\!]\rho &= \lfloor n \rfloor \\
[\![x]\!]\rho &= \rho x \\
[\![t_0 \textbf{ op } t_1]\!]\rho &= [\![t_0]\!]\rho \; \underline{op}_\perp \; [\![t_1]\!]\rho \\
[\![\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2]\!]\rho &= Cond\,([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho) \\
[\![(t_0, t_1)]\!]\rho &= \lfloor ([\![t_0]\!]\rho, [\![t_1]\!]\rho) \rfloor \\
[\![\textbf{fst}(t)]\!]\rho &= \textbf{let } v \Leftarrow [\![t]\!]\rho. \quad \pi_1 v \\
[\![\textbf{snd}(t)]\!]\rho &= \textbf{let } v \Leftarrow [\![t]\!]\rho. \quad \pi_2 v \\
[\![\lambda x.t]\!]\rho &= \left\lfloor \lambda d. \, [\![t]\!]\rho[{}^d/_x] \right\rfloor \\
[\![(t_1 \; t_0)]\!] &= \textbf{let } \varphi \Leftarrow [\![t_1]\!]\rho.\varphi([\![t_0]\!]\rho) \\
[\![\textbf{rec } \; x.t]\!]\rho &= \textbf{fix } \lambda d. \, [\![t]\!]\rho[{}^d/_x]
\end{aligned}$$