

Contents

- Introduction** **ix**
 - 1. Objectives ix
 - 2. Structure x
 - 3. References xi

- 1. Preliminaries** **1**
 - 1.1. Inference Rules 1
 - 1.2. Logic Programming 7

- I. IMP language** **9**

- 2. Operational Semantics of IMP** **11**
 - 2.1. Syntax of IMP 11
 - 2.1.1. Arithmetic Expressions 11
 - 2.1.2. Boolean Expressions 12
 - 2.1.3. Commands 12
 - 2.1.4. Abstract Syntax 12
 - 2.2. Operational Semantics of IMP 12
 - 2.2.1. Memory State 12
 - 2.2.2. Inference Rules 13
 - 2.2.3. Examples 16
 - 2.3. Abstract Semantics: Equivalence of IMP Expressions and Commands 20
 - 2.3.1. Examples: Simple Equivalence Proofs 21
 - 2.3.2. Examples: Parametric Equivalence Proofs 21
 - 2.3.3. Inequality Proofs 23
 - 2.3.4. Diverging Computations 24

- 3. Induction and Recursion** **27**
 - 3.1. Noether Principle of Well-founded Induction 27
 - 3.1.1. Well-founded Relations 27
 - 3.1.2. Noether Induction 32
 - 3.1.3. Weak Mathematical Induction 32
 - 3.1.4. Strong Mathematical Induction 33
 - 3.1.5. Structural Induction 33
 - 3.1.6. Induction on Derivations 35
 - 3.1.7. Rule Induction 35
 - 3.2. Well-founded Recursion 38

- 4. Partial Orders and Fixpoints** **41**
 - 4.1. Orderings and Continuous Functions 41
 - 4.1.1. Orderings 41
 - 4.1.2. Hasse Diagrams 43
 - 4.1.3. Chains 45
 - 4.1.4. Complete Partial Orders 47

4.2.	Continuity and Fixpoints	49
4.2.1.	Monotone and Continuous Functions	49
4.2.2.	Fixpoints	50
4.2.3.	Fixpoint Theorem	50
4.3.	Immediate Consequence Operator	52
4.3.1.	The \hat{R} Operator	52
4.3.2.	Fixpoint of \hat{R}	53
5.	Denotational Semantics of IMP	57
5.1.	λ -notation	57
5.2.	Denotational Semantics of IMP	59
5.2.1.	Function \mathcal{A}	59
5.2.2.	Function \mathcal{B}	60
5.2.3.	Function \mathcal{C}	60
5.3.	Equivalence Between Operational and Denotational Semantics	63
5.3.1.	Equivalence Proofs for \mathcal{A} and \mathcal{B}	63
5.3.2.	Equivalence of \mathcal{C}	64
5.3.2.1.	Completeness of the Denotational Semantics	64
5.3.2.2.	Correctness of the Denotational Semantics	66
5.4.	Computational Induction	68
II.	HOFL language	71
6.	Operational Semantics of HOFL	73
6.1.	HOFL	73
6.1.1.	Typed Terms	73
6.1.2.	Typability and Typechecking	76
6.1.2.1.	Church Type Theory	76
6.1.2.2.	Curry Type Theory	76
6.2.	Operational Semantics of HOFL	78
7.	Domain Theory	83
7.1.	The Domain \mathbb{N}_\perp	83
7.2.	Cartesian Product of Two Domains	83
7.3.	Functional Domains	85
7.4.	Lifting	87
7.5.	Function's Continuity Theorems	88
7.6.	Useful Functions	89
8.	HOFL Denotational Semantics	93
8.1.	HOFL Evaluation Function	93
8.1.1.	Constants	93
8.1.2.	Variables	93
8.1.3.	Binary Operators	94
8.1.4.	Conditional	94
8.1.5.	Pairing	94
8.1.6.	Projections	94
8.1.7.	Lambda Abstraction	94
8.1.8.	Function Application	94
8.1.9.	Recursion	95
8.2.	Typing the Clauses	95
8.3.	Continuity of Meta-language's Functions	96

8.4. Substitution Lemma	98
9. Equivalence between HOFL denotational and operational semantics	99
9.1. Completeness	100
9.2. Equivalence (on Convergence)	101
9.3. Operational and Denotational Equivalence	102
9.4. A Simpler Denotational Semantics	103
III. CCS	105
10. CCS, the Calculus for Communicating Systems	107
10.1. Syntax of CCS	110
10.2. Operational Semantics of CCS	111
10.3. Abstract Semantics of CCS	114
10.3.1. Graph Isomorphism	114
10.3.2. Trace Equivalence	115
10.3.3. Bisimilarity	116
10.4. Compositionality	120
10.4.1. Bisimilarity is Preserved by Parallel Composition	121
10.5. Hennessy - Milner Logic	122
10.6. Axioms for Strong Bisimilarity	124
10.7. Weak Semantics of CCS	125
10.7.1. Weak Bisimilarity	126
10.7.2. Weak Observational Congruence	127
10.7.3. Dynamic Bisimilarity	128
IV. Temporal and Modal Logic	129
11. Temporal Logic and μ-Calculus	131
11.1. Temporal Logic	131
11.1.1. Linear Temporal Logic	131
11.1.2. Computation Tree Logic	132
11.2. μ -Calculus	134
11.3. Model Checking	135
V. π-calculus	137
12. π-Calculus	139
12.1. Syntax of π -calculus	140
12.2. Operational Semantics of π -calculus	141
12.3. Structural Equivalence of π -calculus	143
12.4. Abstract Semantics of π -calculus	143
12.4.1. Strong Early Ground Bisimulations	143
12.4.2. Strong Late Ground Bisimulations	144
12.4.3. Strong Full Bisimilarity	144
12.4.4. Weak Early and Late Ground Bisimulations	145

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, \dots, x_n, y be (well-formed) formulas. An inference rule is written as

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

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

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

The idea is that if we can prove all the formulas x_1, x_2, \dots, 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:

$$r = \emptyset / y.$$

Equivalent notations are:

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

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

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:

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

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

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

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.

$$d_1 \Vdash_R x_1, \quad d_2 \Vdash_R x_2, \quad \dots, \quad d_n \Vdash_R x_n$$

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

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

then we can build a derivation for y :

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

Summarizing all the above:

- $(\emptyset/y) \Vdash_R y$ if $(\emptyset/y) \in R$ (axiom)
- $(\{d_1, \dots, d_n\}/y) \Vdash_R y$ if $(\{x_1, \dots, x_n\}/y) \in R$ and $d_1 \Vdash_R x_1, \dots, d_n \Vdash_R x_n$ (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:

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

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:

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

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:

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

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:

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

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

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

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).

$$\begin{array}{c} \frac{}{\lambda \in L_S} \quad (3) \\ \frac{(\lambda) = () \in L_S}{(()) \in L_S} \quad (2) \quad \frac{}{\lambda \in L_S} \quad (3) \\ \frac{(()) \in L_S}{(())() \in L_S} \quad (1) \end{array}$$

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 i th 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 $a = a_1 a_2 \dots a_n$ is balanced iff both the following properties hold:

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

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 $a = (())()$:

$$\begin{array}{rcccccc} m = & 1 & 2 & 3 & 4 & 5 & 6 \\ a_m = & (& (&) &) & (&) \\ f(a_m) = & 1 & 1 & -1 & -1 & 1 & -1 \\ \sum_{i=1}^m f(a_i) = & 1 & 2 & 1 & 0 & 1 & 0 \end{array}$$

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

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

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

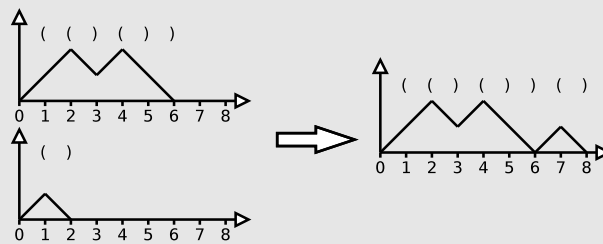
\Rightarrow all the strings produced by the grammar satisfy the two properties;

\Leftarrow any string that satisfies the two properties can be generated by the grammar.

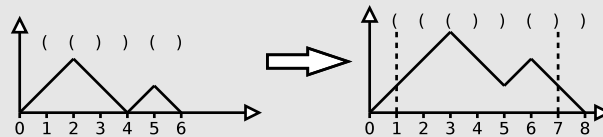
Proof of \Rightarrow) To show the first implication, we proceed by induction over the rules: we assume that the implication is valid for the premises and we show that it holds for the conclusion:

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).

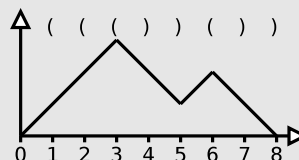


The third rule is just concerned with the empty string that trivially satisfies the two properties.

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

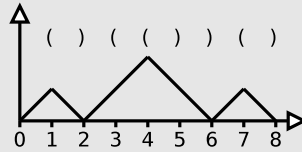
Proof of \Leftarrow) We need to find a derivation for any string that satisfies the two properties. Let a be such a generic string. (We only sketch this direction of the proof, that goes by induction over the length of the string a .) We proceed by case analysis, considering three cases:

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

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 $(())() \in L_S \searrow$ 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 $(())() \in L_S \searrow \lambda \in L_S, (())() \in L_S$ 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. $(())() \in L_S \searrow \lambda \in L_S, (())() \in L_S$
2. $(())() \in L_S \searrow (\in L_S, ())() \in L_S$
3. $(())() \in L_S \searrow ((\in L_S,))() \in L_S$
4. $(())() \in L_S \searrow (() \in L_S,)() \in L_S$
5. $(())() \in L_S \searrow (()) \in L_S, () \in L_S$
6. $(())() \in L_S \searrow (())(\in L_S,) \in L_S$
7. $(())() \in L_S \searrow (())() \in L_S, \lambda \in L_S$

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:

$$(())() \in L_S \searrow (())(\in L_S$$

- Rule (3) cannot be applied.

We show below a successful goal-oriented derivation.

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

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 $X = \{x, y, \dots\}$ be a set of variables, $\Sigma = \{f(.,.), g(.,.), \dots\}$ a signature of function symbols (with given arities), $\Pi = \{p(.,.), q(.,.), \dots\}$ 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, $p(f(g(x), x), g(y))$ 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:

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

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

Let us consider the goal $\text{sum}(s(s(0)), s(s(0)), v)$ 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 $\text{sum}(s(s(0)), s(s(0)), v)$ and $\text{sum}(s(x'), y', s(z'))$. 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

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

We write the derivation described above using the notation

$$\text{sum}(s(s(0)), s(s(0)), v) \quad \leftarrow_{v=s(z')} \quad \text{sum}(s(0), s(s(0)), z')$$

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:

$$\begin{array}{r} \text{sum}(s(s(0)), s(s(0)), v) \quad \leftarrow_{v=s(z')} \quad \text{sum}(s(0), s(s(0)), z') \\ \quad \quad \quad \leftarrow_{z'=s(z'')} \quad \text{sum}(0, s(s(0)), z'') \\ \quad \quad \quad \leftarrow_{z''=s(s(0))} \quad \square \end{array}$$

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

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

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 $m, n, m', n', m_0, n_0, m_1, n_1, \dots$

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

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

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

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 $x, y, x', y', x_0, y_0, x_1, y_1, \dots$

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

Definition 2.1 (IMP: syntax)

The grammar for IMP comprises:

Aexp: Arithmetic expressions, ranged over by a, a', a_0, a_1, \dots

Bexp: Boolean expressions, ranged over by b, b', b_0, b_1, \dots

Com: Commands, ranged over by c, c', c_0, c_1, \dots

The following productions define the syntax of IMP:

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

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.

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 $\text{if}(x = 0) \text{ then}(\text{skip}) \text{ else}(x := x - 1) \in Com$. We can prove its validity by the following (deterministic) derivation

$(\text{if}(x = 0) \text{ then}(\text{skip}) \text{ else}(x := x - 1)) \in Com \quad \swarrow \quad x = 0 \in Bexp, \text{skip} \in Com, x := (x - 1) \in Com$
 $\swarrow \quad x \in Aexp, 0 \in Aexp, \text{skip} \in Com, x := (x - 1) \in Com$
 $\swarrow \quad x - 1 \in Aexp$
 $\swarrow \quad x \in Aexp, 1 \in Aexp$
 $\swarrow \quad \square$

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*:

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

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 σ_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.

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

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

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

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

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

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

$$\frac{}{\langle n, \sigma \rangle \rightarrow n} \text{ (num)} \quad (2.1)$$

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).

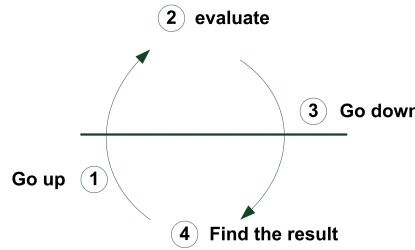
$$\frac{}{\langle x, \sigma \rangle \rightarrow \sigma(x)} \text{ (ide)} \quad (2.2)$$

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 .

$$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma \rangle \rightarrow n_0 + n_1} \text{ (sum)} \quad (2.3)$$

The rule 2.2 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 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.

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

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

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

$$\frac{}{\langle v, \sigma \rangle \rightarrow v} \text{ (bool)} \quad (2.6)$$

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

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

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

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

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

Next, we move to the inference rules for commands.

$$\frac{}{\langle \mathbf{skip}, \sigma \rangle \rightarrow \sigma} \text{ (skip)} \quad (2.12)$$

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

$$\frac{\langle a, \sigma \rangle \rightarrow m}{\langle x := a, \sigma \rangle \rightarrow \sigma[m/x]} \text{ (assign)} \quad (2.13)$$

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 $\sigma(x)$).

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

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).

$$\frac{\langle b, \sigma \rangle \rightarrow \mathbf{true} \quad \langle c_0, \sigma \rangle \rightarrow \sigma'}{\langle \mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma \rangle \rightarrow \sigma'} \text{ (ifft)} \quad (2.15)$$

$$\frac{\langle b, \sigma \rangle \rightarrow \mathbf{false} \quad \langle c_1, \sigma \rangle \rightarrow \sigma'}{\langle \mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma \rangle \rightarrow \sigma'} \text{ (iff)} \quad (2.16)$$

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).

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

$$\frac{\langle b, \sigma \rangle \rightarrow \mathbf{false}}{\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \sigma} \text{ (whff)} \quad (2.18)$$

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:

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

2.2.3. Examples

Example 2.7 (Semantic evaluation of a command)

Let us consider the (extra-bracketed) command

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

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

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

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 $y \geq 0$) 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^2 is computed as the sum of the first y odd numbers $\sum_{i=0}^y (2i + 1)$.

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.)

$$\begin{array}{c}
 \frac{}{x} \quad \frac{}{2} \quad \frac{}{y} \\
 \frac{}{x} \quad \frac{}{(2 \times y)} \\
 \frac{}{a_1 = ((x + 2 \times y))} \quad \frac{}{1} \\
 \frac{}{x} \quad \frac{}{a = ((x + (2 \times y)) + 1)} \quad \frac{}{y} \quad \frac{}{(y - 1)} \\
 \frac{}{0} \quad \frac{}{y} \quad \frac{}{c_3 = (x = ((x + (2 \times y)) + 1))} \quad \frac{}{c_4 = (y := (y - 1))} \\
 \frac{}{x} \quad \frac{}{0} \quad \frac{}{0 \leq y} \quad \frac{}{c_2 = ((x! = ((x + (2 \times y)) + 1)); (y := y - 1))} \\
 \frac{}{x := 0} \quad \frac{}{c_1 = (\mathbf{while}(0 \leq y) \mathbf{do}((x := ((x + 2 \times y)) + 1)); (y := (y - 1))))} \\
 \frac{}{c = ((x := 0); (\mathbf{while}(0 \leq y) \mathbf{do}((x := ((x + (2 \times y)) + 1)); (y := (y + 1))))}
 \end{array}$$

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

$$\begin{array}{c}
 \frac{}{a} \\
 \frac{}{a_1} \\
 c = x := 0; \mathbf{while} \ 0 \leq y \ \mathbf{do} \ (\underbrace{x := x + (2 \times y) + 1}_{c_3}; \underbrace{y := y - 1}_{c_4}) \\
 \frac{}{c_2} \\
 \frac{}{c_1} \\
 c
 \end{array}$$

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 $\langle c, ({}^{27}/x, {}^2/y) \rangle \rightarrow \sigma$, 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.

$$\frac{\frac{}{\langle 0, ({}^{27}/x, {}^2/y) \rangle \rightarrow 0} \text{ num}}{\langle x := 0, ({}^{27}/x, {}^2/y) \rangle \rightarrow ({}^{27}/x, {}^2/y) [{}^0/x] = \sigma'} \text{ assign} \quad \frac{}{\langle c_1, \sigma' \rangle \rightarrow \sigma} \text{ seq}}{\langle c, ({}^{27}/x, {}^2/y) \rangle \rightarrow \sigma}$$

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 $\sigma' = ({}^0/x, {}^2/y)$.

$$\frac{\frac{}{\langle 0, \sigma' \rangle \rightarrow 0} \text{ num} \quad \frac{}{\langle y, \sigma' \rangle \rightarrow \sigma'(y) = 2} \text{ ide}}{\langle 0 \leq y, \sigma' \rangle \rightarrow (0 \leq 2) = \mathbf{true}} \text{ leq} \quad \frac{}{\langle c_2, \sigma' \rangle \rightarrow \sigma''} \quad \frac{}{\langle c_1, \sigma'' \rangle \rightarrow \sigma} \text{ whtt}}{\langle c_1, \sigma' \rangle \rightarrow \sigma}$$

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

$$\frac{\frac{\langle a_1, ({}^0/x, {}^2/y) \rangle \rightarrow m' \quad \overline{\langle 1, ({}^0/x, {}^2/y) \rangle \rightarrow 1} \text{ num}}{\langle a, ({}^0/x, {}^2/y) \rangle \rightarrow m = m' + 1} \text{ sum} \quad \frac{\langle y - 1, \sigma''' \rangle \rightarrow m''}{\langle c_4, \sigma''' \rangle \rightarrow \sigma''' [m''/y] = \sigma''} \text{ assign}}{\frac{\langle c_3, ({}^0/x, {}^2/y) \rangle \rightarrow ({}^0/x, {}^2/y) [m/x] = \sigma''' \quad \langle c_4, \sigma''' \rangle \rightarrow \sigma''' [m''/y] = \sigma''}{\langle c_2, ({}^0/x, {}^2/y) \rangle \rightarrow \sigma''} \text{ seq}} \text{ assign}$$

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

$$\frac{\overline{\langle x, ({}^0/x, {}^2/y) \rangle \rightarrow 0} \text{ ide} \quad \frac{\overline{\langle 2, ({}^0/x, {}^2/y) \rangle \rightarrow 2} \text{ num} \quad \overline{\langle y, ({}^0/x, {}^2/y) \rangle \rightarrow 2} \text{ ide}}{\langle 2 \times y, ({}^0/x, {}^2/y) \rangle \rightarrow m''' = 2 \times 2 = 4} \text{ prod}}{\langle a_1, ({}^0/x, {}^2/y) \rangle \rightarrow m' = 0 + 4 = 4} \text{ sum}$$

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

$$\frac{\overline{\langle y, ({}^5/x, {}^2/y) \rangle \rightarrow 2} \text{ ide} \quad \overline{\langle 1, ({}^5/x, {}^2/y) \rangle \rightarrow 1} \text{ num}}{\langle y - 1, ({}^5/x, {}^2/y) \rangle \rightarrow m'' = 2 - 1 = 1} \text{ dif}$$

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

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

$$\frac{\overline{\langle 0 \leq y, ({}^5/x, {}^1/y) \rangle \rightarrow \mathbf{true}} \text{ leq} \quad \frac{\overline{\langle c_2, ({}^5/x, {}^1/y) \rangle \rightarrow ({}^5/x, {}^1/y) [{}^8/x] [{}^0/y] = \sigma''''} \text{ seq} \quad \langle c_1, \sigma'''' \rangle \rightarrow \sigma}{\langle c_1, ({}^5/x, {}^1/y) \rangle \rightarrow \sigma} \text{ whtt}$$

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

$$\frac{\overline{\langle 0 \leq y, ({}^8/x, {}^0/y) \rangle \rightarrow \mathbf{true}} \text{ leq} \quad \frac{\overline{\langle c_2, ({}^8/x, {}^0/y) \rangle \rightarrow ({}^8/x, {}^0/y) [{}^9/x] [{}^{-1}/y] = \sigma'''''} \text{ seq} \quad \langle c_1, \sigma''''' \rangle \rightarrow \sigma}{\langle c_1, ({}^8/x, {}^0/y) \rangle \rightarrow \sigma} \text{ whtt}$$

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

$$\frac{\overline{\langle 0 \leq y, ({}^9/x, {}^{-1}/y) \rangle \rightarrow \mathbf{false}} \text{ leq}}{\langle c_1, ({}^9/x, {}^{-1}/y) \rangle \rightarrow ({}^9/x, {}^{-1}/y) = \sigma} \text{ whff}$$

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

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}{l}
\langle c, ({}^{27}/x, {}^2/y) \rangle \rightarrow \sigma \quad \swarrow \\
\swarrow_{\sigma' = ({}^{27}/x, {}^2/y)[{}^n/x]} \\
\swarrow_{n=0 \ \sigma' = ({}^0/x, {}^2/y)} \\
\swarrow \\
\swarrow_{n_1=0 \ n_2=2} \\
\swarrow_{\sigma''' = ({}^0/x, {}^2/y)[{}^m/x]} \\
\swarrow_{m=0+(2 \times 2)+1=5 \ \sigma''' = ({}^5/x, {}^2/y)} \\
\swarrow_{\sigma'' = ({}^5/x, {}^2/y)[{}^{2^{-1}}/y] = ({}^5/x, {}^1/y)} \\
\swarrow_{\sigma'''' = ({}^5/x, {}^1/y)[{}^{5+2 \times 1+1}/x][{}^0/y] = ({}^8/x, {}^0/y)} \\
\swarrow_{\sigma'''' = ({}^8/x, {}^0/y)[{}^{8+2 \times 0+1}/x][{}^{0^{-1}}/y] = ({}^9/x, {}^{-1}/y)} \\
\swarrow_{\sigma = ({}^9/x, {}^{-1}/y)} \\
\swarrow^* \\
\langle x := 0, ({}^{27}/x, {}^2/y) \rangle \rightarrow \sigma' \quad \langle c_1, \sigma' \rangle \rightarrow \sigma \\
\langle 0, ({}^{27}/x, {}^2/y) \rangle \rightarrow n \quad \langle c_1, ({}^{27}/x, {}^2/y)[{}^n/x] \rangle \rightarrow \sigma \\
\langle c_1, ({}^0/x, {}^2/y) \rangle \rightarrow \sigma \\
\langle 0 \leq y, ({}^0/x, {}^2/y) \rangle \rightarrow \mathbf{true} \quad \langle c_2, ({}^0/x, {}^2/y) \rangle \rightarrow \sigma'' \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma \\
\langle 0, ({}^0/x, {}^2/y) \rangle \rightarrow n_1 \quad \langle y, ({}^0/x, {}^2/y) \rangle \rightarrow n_2 \quad n_1 \leq n_2 \\
\langle c_2, ({}^0/x, {}^2/y) \rangle \rightarrow \sigma'' \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma \\
\langle c_3, ({}^0/x, {}^2/y) \rangle \rightarrow \sigma''' \quad \langle c_4, \sigma''' \rangle \rightarrow \sigma'' \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma \\
\langle x + (2 \times y) + 1, ({}^0/x, {}^2/y) \rangle \rightarrow m \quad \langle c_4, ({}^0/x, {}^2/y)[{}^m/x] \rangle \rightarrow \sigma'' \\
\langle c_1, \sigma'' \rangle \rightarrow \sigma \\
\langle c_4, ({}^5/x, {}^2/y) \rangle \rightarrow \sigma'' \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma \\
\langle c_1, ({}^5/x, {}^1/y) \rangle \rightarrow \sigma \\
\langle c_1, ({}^8/x, {}^0/y) \rangle \rightarrow \sigma \\
\langle c_1, ({}^9/x, {}^{-1}/y) \rangle \rightarrow \sigma \\
\langle 0 \leq y, ({}^9/x, {}^{-1}/y) \rangle \rightarrow \mathbf{false} \\
\quad \square
\end{array}$$

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

$$\langle c, \sigma \rangle \not\rightarrow \quad \text{iff} \quad \neg \exists \sigma'. \langle c, \sigma \rangle \rightarrow \sigma'$$

The condition $\neg \exists \sigma'. \langle c, \sigma \rangle \rightarrow \sigma'$ can be written equivalently as $\forall \sigma'. \langle c, \sigma \rangle \not\rightarrow \sigma'$.

Example 2.8 (Non termination)

Let us consider the command

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

Given σ , the only possible derivation goes as follows:

$$\begin{array}{l}
\langle c, \sigma \rangle \rightarrow \sigma' \quad \swarrow \\
\swarrow \\
\swarrow_{\sigma'' = \sigma} \\
\langle \mathbf{true}, \sigma \rangle \rightarrow \mathbf{true}, \langle \mathbf{skip}, \sigma \rangle \rightarrow \sigma'', \langle c, \sigma'' \rangle \rightarrow \sigma' \\
\swarrow \\
\langle \mathbf{skip}, \sigma \rangle \rightarrow \sigma'', \langle c, \sigma'' \rangle \rightarrow \sigma' \\
\swarrow_{\sigma'' = \sigma} \\
\langle c, \sigma \rangle \rightarrow \sigma'
\end{array}$$

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 $\langle c, \sigma \rangle \not\rightarrow$. We proceed by contradiction, assuming there exists σ' for which we can find a (finite) derivation d for $\langle c, \sigma \rangle \rightarrow \sigma'$. Let d be the derivation sketched below:

$$\begin{array}{c}
\langle c, \sigma \rangle \rightarrow \sigma' \quad \swarrow \quad \langle \mathbf{true}, \sigma \rangle \rightarrow \mathbf{true}, \langle \mathbf{skip}, \sigma \rangle \rightarrow \sigma'', \langle c, \sigma'' \rangle \rightarrow \sigma' \\
\vdots \\
(*) \quad \swarrow \quad \langle c, \sigma \rangle \rightarrow \sigma' \\
\vdots \\
\swarrow \quad \square
\end{array}$$

We have marked by (*) the last occurrence of the goal $\langle c, \sigma \rangle \rightarrow \sigma'$. 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:

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

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:

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

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:

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

Notice that if the evaluation of $\langle c_1, \sigma \rangle$ diverges we have no σ' such that $\langle c_1, \sigma \rangle \rightarrow \sigma'$. 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:

$$\begin{aligned} c_1 &= \text{while } x \neq 0 \text{ do } x := 0 \\ c_2 &= x := 0 \end{aligned}$$

It is immediate to prove that

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

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.

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

$$\begin{array}{r} \langle c_1, \sigma \rangle \rightarrow \sigma' \quad \swarrow \quad \langle x \neq 0, \sigma \rangle \rightarrow \mathbf{true} \quad \langle x := 0, \sigma \rangle \rightarrow \sigma'' \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma' \\ \swarrow_{\sigma'' = \sigma[0/x]} \quad \langle c_1, \sigma[0/x] \rangle \rightarrow \sigma' \\ \swarrow_{\sigma' = \sigma[0/x]} \quad \langle x \neq 0, \sigma[0/x] \rangle \rightarrow \mathbf{false} \\ \swarrow \quad \sigma[0/x](x) = 0 \\ \swarrow \quad \square \end{array}$$

$(\sigma(x) = 0)$

$$\begin{array}{r} \langle c_1, \sigma \rangle \rightarrow \sigma' \quad \swarrow_{\sigma' = \sigma} \quad \langle x \neq 0, \sigma \rangle \rightarrow \mathbf{false} \\ \swarrow \quad \sigma(x) = 0 \\ \swarrow \quad \square \end{array}$$

Finally, we observe the following:

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

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:

$$c_1 = \mathbf{while } b \mathbf{ do } c$$

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

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.

$\langle b, \sigma \rangle \rightarrow \mathbf{false}$

$$\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \sigma' \quad \swarrow \quad \langle b, \sigma \rangle \rightarrow \mathbf{false}$$

$$\swarrow_{\sigma=\sigma'} \quad \square$$

$$\langle \mathbf{if } b \mathbf{ then}(c; c_1) \mathbf{ else skip}, \sigma \rangle \rightarrow \sigma' \quad \swarrow \quad \langle b, \sigma \rangle \rightarrow \mathbf{false} \quad \langle \mathbf{skip}, \sigma \rangle \rightarrow \sigma'$$

$$\swarrow_{\sigma'=\sigma} \quad \square$$

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

$\langle b, \sigma \rangle \rightarrow \mathbf{true}$

$$\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \sigma' \quad \swarrow \quad \langle b, \sigma \rangle \rightarrow \mathbf{true} \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma'$$

$$\swarrow \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma'$$

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 :

$$\langle \mathbf{if } b \mathbf{ then}(c; \mathbf{while } b \mathbf{ do } c) \mathbf{ else skip}, \sigma \rangle \rightarrow \sigma' \quad \swarrow \quad \langle b, \sigma \rangle \rightarrow \mathbf{true} \quad \langle c; \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \sigma'$$

$$\swarrow \quad \langle c; \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \sigma'$$

$$\swarrow \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma'$$

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 $\langle b, \sigma \rangle \rightarrow \mathbf{true}$ then the two derivations for c_1 and c_2 will necessarily lead to the same result whenever they terminate, and if one diverges the other diverges too.

Summing up the two cases, and since there is no more alternative, 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

$$\frac{\dots \quad \dots}{\frac{\langle x > 0, \sigma[1/x] \rangle \rightarrow \mathbf{true} \quad \langle x := 1, \sigma[1/x] \rangle \rightarrow \sigma[1/x] \quad \langle \mathbf{while} \ x > 0 \ \mathbf{do} \ x := 1, \sigma[1/x] \rangle \rightarrow \sigma'}{\langle \mathbf{while} \ x > 0 \ \mathbf{do} \ x := 1, \sigma[1/x] \rangle \rightarrow \sigma'}}$$

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

Summing up all the above we conclude that:

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

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).

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

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 $\sigma(x) \leq 0$)

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

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

$$\exists \sigma, \sigma'. \begin{cases} \langle (\mathbf{while} \ x > 0 \ \mathbf{do} \ x := 1); x := 0, \sigma \rangle \not\rightarrow \\ \langle x := 0, \sigma \rangle \rightarrow \sigma' \end{cases}$$

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 $c_1 \not\sim c_2$.

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:

$$\forall \sigma, \sigma' \quad \begin{cases} \langle c_1, \sigma \rangle \rightarrow \sigma' \Leftrightarrow \langle c_2, \sigma \rangle \rightarrow \sigma' \\ \langle c_1, \sigma \rangle \not\rightarrow \Leftrightarrow \langle c_2, \sigma \rangle \not\rightarrow \end{cases}$$

Next we see an example where this situation emerges.

Example 2.14 (Proofs of non-termination)

Let us consider the commands:

$$\begin{aligned} c_1 &= \mathbf{while} \ x > 0 \ \mathbf{do} \ x := 1 \\ c_2 &= \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x + 1 \end{aligned}$$

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 $x > 0$:

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

$$\begin{array}{c} \langle c_1, \sigma \rangle \rightarrow \sigma' \quad \swarrow_{\sigma'=\sigma} \quad \langle x > 0, \sigma \rangle \rightarrow \mathbf{false} \\ \swarrow \quad \square \end{array}$$

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.

$(\sigma(x) > 0)$

$$\begin{array}{c} \langle c_1, \sigma \rangle \rightarrow \sigma' \quad \swarrow \quad \langle x > 0, \sigma \rangle \rightarrow \mathbf{true} \quad \langle x := 1, \sigma \rangle \rightarrow \sigma'' \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma' \\ \swarrow \quad \langle x := 1, \sigma \rangle \rightarrow \sigma'' \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma' \\ \swarrow_{\sigma''=\sigma[1/x]} \quad \langle c_1, \sigma[1/x] \rangle \rightarrow \sigma' \\ \swarrow \quad \langle x > 0, \sigma[1/x] \rangle \rightarrow \mathbf{true} \quad \langle x := 1, \sigma[1/x] \rangle \rightarrow \sigma''' \quad \langle c_1, \sigma''' \rangle \rightarrow \sigma' \\ \swarrow \quad \langle x := 1, \sigma[1/x] \rangle \rightarrow \sigma''' \quad \langle c_1, \sigma''' \rangle \rightarrow \sigma' \\ \swarrow_{\sigma'''=\sigma[1/x][1/x]=\sigma[1/x]} \quad \langle c_1, \sigma[1/x] \rangle \rightarrow \sigma' \\ \swarrow \quad \dots \end{array}$$

Note that we reach the same subgoal $\langle c_1, \sigma[1/x] \rangle \rightarrow \sigma'$ already inspected, i.e., the derivation will loop.

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

$$\begin{array}{c} \langle c_2, \sigma \rangle \rightarrow \sigma' \quad \swarrow \quad \langle x > 0, \sigma \rangle \rightarrow \mathbf{true} \quad \langle x := x + 1, \sigma \rangle \rightarrow \sigma'' \quad \langle c_2, \sigma'' \rangle \rightarrow \sigma' \\ \swarrow \quad \langle x := x + 1, \sigma \rangle \rightarrow \sigma'' \quad \langle c_2, \sigma'' \rangle \rightarrow \sigma' \\ \swarrow_{\sigma''=\sigma[\sigma(x)+1/x]} \quad \langle c_2, \sigma[\sigma(x)+1/x] \rangle \rightarrow \sigma' \\ \swarrow \quad \langle x > 0, \sigma[\sigma(x)+1/x] \rangle \rightarrow \mathbf{true} \quad \langle x := x + 1, \sigma[\sigma(x)+1/x] \rangle \rightarrow \sigma''' \\ \langle c_2, \sigma''' \rangle \rightarrow \sigma' \\ \swarrow \quad \langle x := x + 1, \sigma[\sigma(x)+1/x] \rangle \rightarrow \sigma''' \quad \langle c_2, \sigma''' \rangle \rightarrow \sigma' \\ \swarrow_{\sigma'''=\sigma''[\sigma''(x)+1/x]=\sigma[\sigma(x)+2/x]} \quad \dots \end{array}$$

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 i th 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 k th 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 $w = \mathbf{while} \ b \ \mathbf{do} \ c$. 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:

$$\frac{\sigma \in S \quad \forall \sigma' \in S. (\langle c, \sigma' \rangle \rightarrow \sigma'' \Rightarrow \sigma'' \in S) \quad \forall \sigma' \in S. (\langle b, \sigma' \rangle \rightarrow \mathbf{true})}{\langle w, \sigma \rangle \rightarrow} \quad (2.19)$$

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 in S . Note that the condition

$$(\langle c, \sigma' \rangle \rightarrow \sigma'' \Rightarrow \sigma'' \in S)$$

is satisfied even when $\langle c, \sigma' \rangle \rightarrow$, 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 $<$ over a set A is a subset of the cartesian product $A \times A$.

$$< \subseteq A \times A$$

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

Definition 3.2 (Infinite descending chain)

Given a relation $<$ over the set A , an infinite descending chain is an infinite sequence $\{a_i\}_{i \in \omega}$ of elements in A such that

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

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

$$a(0) > a(1) > a(2) > \dots$$

Definition 3.3 (Well-founded relation)

A relation is well-founded if it admits no infinite descending chains.

Definition 3.4 (Transitive closure)

Let $<$ be a relation over A . The transitive closure of $<$, written $<^+$, is defined by the following inference rules

$$\frac{a < b}{a <^+ b} \quad \frac{a <^+ b \quad b <^+ c}{a <^+ c}$$

Definition 3.5 (Transitive and reflexive closure)

Let $<$ be a relation over A . The transitive and reflexive closure of $<$, written $<^*$, is defined by the following inference rules

$$a <^* a \quad \frac{a <^* b \quad b < c}{a <^* c}$$

Theorem 3.6

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

$$x = z_0 < z_1 < \dots < z_k = y.$$

Theorem 3.7 (Well-foundedness of $<^+$)

A relation $<$ is well-founded if and only if its transitive closure $<^+$ is well-founded.

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

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

$$a_0 >^+ a_1 >^+ a_2 \dots$$

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

$$a_0 > \dots > a_1 > \dots > a_2 > \dots$$

leading to a contradiction. □

Definition 3.8 (Acyclic relation)

We say that $<$ has a cycle if $\exists a \in A. a <^+ a$. We say that $<$ is acyclic if it has no cycle.

Theorem 3.9 (Well-founded relations are acyclic)

If the relation $<$ is well-founded, then it is acyclic.

Proof. We need to prove that:

$$\forall x \in A. x \not<^+ x$$

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

$$x < x_1 < \dots < x_n < x$$

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

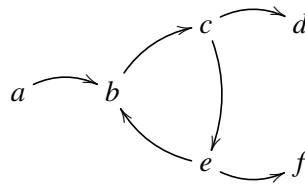


Figure 3.1.: Graph of a relation

$$x > x_n > \dots > x_1 > x > x_n > \dots > x_1 > x > \dots$$

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

Theorem 3.10 (Well-founded relations over finite sets)

Let A be a finite set and let $<$ be acyclic, then $<$ 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 $<$ is acyclic by hypothesis. \square

Lemma 3.11 (Well-founded relation)

Let $<$ be a relation over the set A . The relation $<$ 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 $<$ 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.

(\Leftarrow) We assume that every nonempty subset of A has a minimal element and we need to show that $<$ has no infinite descending chain. By contradiction, we assume that $<$ has an infinite descending chain $a_1 > a_2 > a_3 > \dots$ and we let $Q = \{a_1, a_2, a_3, \dots\}$ 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 $a_i > a_{i+1}$. This contradicts the hypothesis, concluding the proof.

(\Rightarrow) We assume the relation $<$ 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 exist an element $a_1 \in Q$ such that $a_0 > a_1$, and we can iterate the reasoning (i.e. a_1 is not minimal and there is $a_2 \in Q$ with $a_1 > a_2$, etc.). But since $<$ is well-founded, we cannot build such an infinite descending chain. \square

Example 3.12 (Natural numbers)

Both $n < n + 1$ and $n < n + 1 + k$, 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 3.13 (Terms over sorted signatures)

Let

- S be a set of sorts (i.e. the set of the different data types we want to consider);
- $\Sigma = \{\Sigma_{s_1 \dots s_n, s}\}_{s_1 \dots s_n, s \in S}$ a signature over S , i.e. a set of typed operators ($f \in \Sigma_{s_1 \dots s_n, s}$ is an operator that takes n arguments, the i th 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 3.14 (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:

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

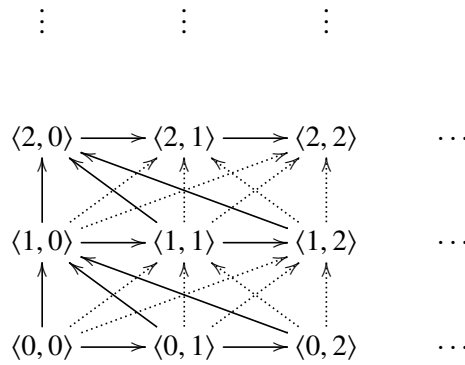


Figure 3.2.: Graph of the lexicographic order relation over pairs of natural numbers.

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

Example 3.15 (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:

$$t_i < f(t_1, \dots, t_n) \quad i = 1, \dots, n$$

The idea is that a term t_i precedes (according to $<$, 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.:

$$c < f(c, c) < f(f(c, c), c) < f(f(f(c, c), c), f(c, c))$$

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 $s < t$ 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.

Example 3.16 (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):

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

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:

$$\langle 5, 1 \rangle > \langle 3, 100 \rangle > \langle 3, 14 \rangle > \langle 0, 1000 \rangle > \langle 0, 0 \rangle$$

It is worth to note that any element $\langle n, m \rangle$ with $n \geq 1$ is preceded by infinitely many elements (e.g., $\forall k. \langle 0, k \rangle < \langle 1, 0 \rangle$) 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 $\forall b < m. b \notin Q$. 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 exist 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 3.11 the relation is well-founded.

Example 3.17 (A counterexample: 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:

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

3.1.2. Noether Induction

Theorem 3.18

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

$$\frac{\forall a \in A. (\forall b < a. P(b)) \rightarrow P(a)}{\forall a \in A. P(a)}$$

Proof. We show the two implications separately.

(\Leftarrow): if $\forall a. P(a)$ then $(\forall b < a. P(b)) \rightarrow P(a)$ is true for any a because the premise $(\forall b < a. P(b))$ is not relevant (the conclusion of the implication is true).

(\Rightarrow): 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 $<$ is well-founded, we know by Lemma 3.11 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 $\forall b < m. b \notin Q$, i.e., $\forall b < m. P(b)$. But this leads to a contradiction, because by hypothesis we have $\forall a \in A. (\forall b < a. P(b)) \rightarrow P(a)$ and instead the predicate $(\forall b < m. P(b)) \rightarrow P(m)$ is false. Therefore Q must be empty and $\forall a \in A. P(a)$ must hold.

□

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

$$A = \omega \quad n < m \leftrightarrow m = n + 1$$

Theorem 3.19 (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:

$$n < m \leftrightarrow m = n + k + 1$$

Theorem 3.20 (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 $i = 0, \dots, n$.

The principle is helpful, because it allows us to exploit the hypothesis $P(0) \wedge P(1) \wedge \dots \wedge P(n)$ 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:

$$t_i < f(t_1, \dots, t_n) \quad i = 1, \dots, n$$

Definition 3.21 (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

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

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 3.22

Let us consider the grammar of IMP arithmetic expressions:

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

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 $\forall a \in Aexp. P(a)$.) 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 3.23 (Structural induction over arithmetic expressions)

Let us consider the case of arithmetic expressions seen above and prove that the evaluation of expressions is deterministic:

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

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 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 obtained from the evaluation of a_0 and a_1 . By hypothesis for structural induction we can assume that $m_0 = m'_0$ and $m_1 = m'_1$. Thus, $m = m'$.

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 3.24 (Immediate (sub-)derivation)

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

Example 3.25 (Immediate (sub-)derivation)

Let us consider the derivation

$$\frac{\frac{}{\langle 1, \sigma \rangle \rightarrow 1} \text{ num} \quad \frac{}{\langle 2, \sigma \rangle \rightarrow 2} \text{ num}}{\langle 1 + 2, \sigma \rangle \rightarrow 1 + 2 = 3} \text{ sum}$$

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 3.26 (Proper sub-derivation)

We say that d' is a closed sub-derivation of d if and only if $d' <^+ d$.

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

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 (\emptyset/y) if it is an axiom, or $(\{x_1, \dots, x_n\}/y)$ 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 (\emptyset/y) , we have that $P(y)$ holds;
- for any other rule $(\{x_1, \dots, x_n\}/y)$ we have that $(\forall 1 \leq i \leq n. x_i \in I_R \wedge P(x_i)) \Rightarrow P(y)$.

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

Note that most of the time we will use a simpler but less powerful rule

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

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 3.27 (Proof by rule induction)

We have seen in Example 3.23 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

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

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”:

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

(rule skip): we want to show that

$$P(\langle \text{skip}, \sigma \rangle \rightarrow \sigma) \stackrel{\text{def}}{=} \forall \sigma_1. \langle \text{skip}, \sigma \rangle \rightarrow \sigma_1 \implies \sigma_1 = \sigma$$

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

$$\langle \text{skip}, \sigma \rangle \rightarrow \sigma_1 \quad \nwarrow_{\sigma_1 = \sigma} \quad \square$$

(rule assign): assuming

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

we want to show that

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

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

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

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

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

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

We have:

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

But now we can apply the first inductive hypotheses:

$$P(\langle c_0, \sigma \rangle \rightarrow \sigma'') \stackrel{\text{def}}{=} \forall \sigma''_1. \langle c_0, \sigma \rangle \rightarrow \sigma''_1 \implies \sigma''_1 = \sigma''$$

to conclude that $\sigma''_1 = \sigma''$, which together with the second inductive hypothesis

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

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

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

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

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

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

But then, exploiting the inductive hypothesis

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

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

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

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

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

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

$$\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \sigma_1 \frown_{\sigma_1 = \sigma} \square$$

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

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

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

$$\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \sigma_1 \frown \langle c, \sigma \rangle \rightarrow \sigma''_1 \quad \langle \mathbf{while } b \mathbf{ do } c, \sigma''_1 \rangle \rightarrow \sigma_1$$

But now we can apply the first inductive hypotheses:

$$P(\langle c, \sigma \rangle \rightarrow \sigma'') \stackrel{\text{def}}{=} \forall \sigma''_1. \langle c, \sigma \rangle \rightarrow \sigma''_1 \implies \sigma''_1 = \sigma''$$

to conclude that $\sigma''_1 = \sigma''$, which together with the second inductive hypothesis

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

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 3.28 (Well-founded recursion)

Let us consider the Peano formula that defines the product of natural numbers

$$\begin{aligned} p(0, y) &= 0 \\ p(x + 1, y) &= y + p(x, y) \end{aligned}$$

Let us write the definition in a slightly different way

$$\begin{aligned} p_y(0) &= 0 \\ p_y(x + 1) &= y + p_y(x) \end{aligned}$$

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

Let us make the intuition more precise.

Definition 3.29 (Set of predecessors)

Given a well founded relation $< \subseteq B \times B$, the set of predecessors of a set $I \subseteq B$ is the set

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

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

Theorem 3.30 (Well-founded recursion)

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

- $b \in B$
- $h : <^{-1} \{b\} \rightarrow C$

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

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

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 3.31 (Primitive recursion)

Let us recast the Peano formula seen above (Example 3.28) to the formal scheme of primitive recursion.

- $p_y(0) = F_y(0, p_y \upharpoonright \emptyset) = 0$
- $p_y(x + 1) = F_y(x + 1, p_y \upharpoonright^{<-1} \{x + 1\}) = y + p_y(x)$

Example 3.32 (Structural recursion)

Let us consider the signature Σ for binary trees $B = T_\Sigma$, where $\Sigma_0 = \{0, 1, \dots\}$ and $\Sigma_2 = \text{cons}$. Take the well-founded relation $x_i < \text{cons}(x_1, x_2)$, $i = 1, 2$. Let $C = \omega$.

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

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

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

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

or more formally

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

Example 3.33 (Ackermann function)

The Ackermann function $\text{ack}(z, x, y) = \text{ack}_y(z, x)$ is defined by well-founded recursion (exploiting the lexicographic order over pair of natural numbers) by letting

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

We have

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

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

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

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 (ω, \leq) be the usual ordering on the set of natural numbers, (ω, \leq) is a total order.

- reflexivity: $\forall n \in \omega. n \leq n$
- antisymmetric: $\forall n, m \in \omega. n \leq m \wedge m \leq n \implies m = n$
- transitivity: $\forall n, m, z \in \omega. n \leq m \wedge m \leq z \implies n \leq z$
- 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:

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

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:

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

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.

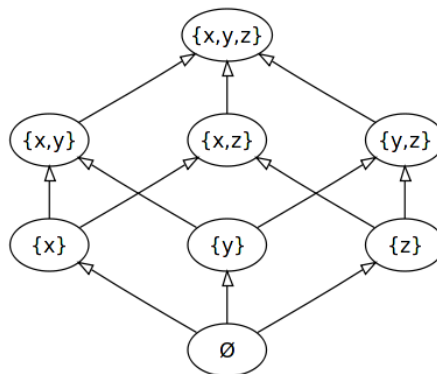


Figure 4.1.: Hasse diagram for the powerset over $\{x, y, z\}$ ordered by inclusion

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 :

$$\frac{\emptyset}{xH^*x} \quad \frac{xH^*y \wedge yH^*z}{xH^*z}$$

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.

$$\frac{\frac{\emptyset}{yH^*y \wedge yHz}}{yH^*z}$$

The above theorem only allows to represent finite orders.

Example 4.10 (Infinite order)

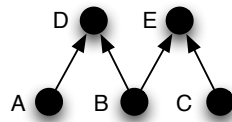
Let us see that the Hasse diagrams does not work with infinite orders. Let $(\omega \cup \{\infty\}, \leq)$ be the usual order on natural numbers extended by placing $n \leq \infty$ and $\infty \leq \infty$. The Hasse diagram eliminates all the arcs between each natural number and ∞ . Now using the transitive 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 $a = b$. \square

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. $x \sqsubseteq^{-1} y \Leftrightarrow y \sqsubseteq x$).

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. $\forall a \in A. a \not\sqsubseteq y$.

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 X . Formally:

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

and we write $\text{lub}(X) = p$.

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*.

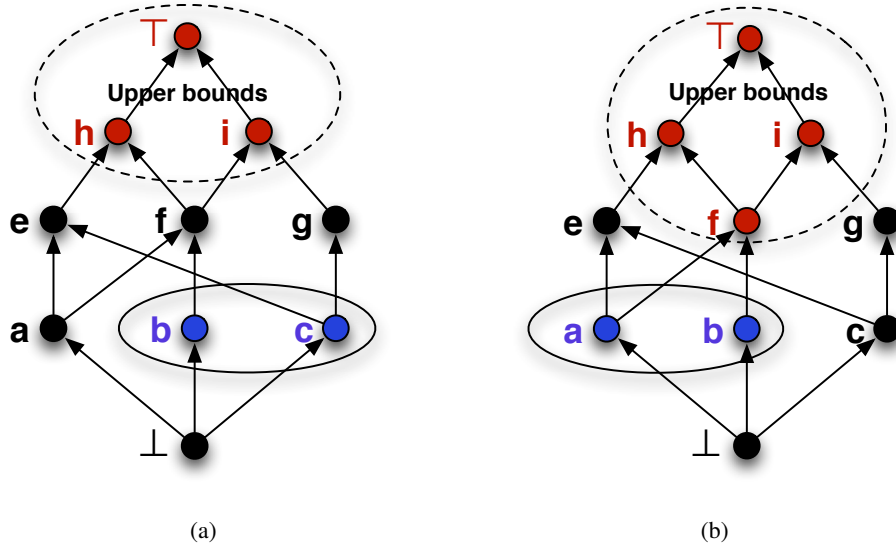


Figure 4.2.: Two subsets of a poset, the right one with LUB and the left one without LUB

Definition 4.17 (Chain)

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

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \dots$$

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 $C : \omega \rightarrow P$ 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. \square

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.

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

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:

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

Where $\sqsubseteq \upharpoonright \omega$ 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:

$$P = \omega \rightarrow \omega$$

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

$$nfm \wedge nfm' \implies m = m'$$

So the set P can be viewed as:

$$P = \{f \subseteq \omega \times \omega \mid nfm \wedge nfm' \implies m = m'\}$$

Now it is easy to define a partial order on P :

$$f \sqsubseteq g \iff \forall x. f(x) \downarrow \implies f(x) = g(x)$$

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

Proof. Let $R_0 \subseteq R_1 \subseteq R_2 \subseteq \dots$ with $\forall i. nR_i m \wedge nR_i m' \implies m = m'$ be a chain of P , we will show that the union relation is still a function, that is $\cup R_i$ has the functional property:

$$n(\cup R_i)m \wedge n(\cup R_i)m' \implies m = m'$$

Let us assume $n(\cup R_i)m \wedge n(\cup R_i)m'$ so we have $\exists k, k'. nR_k m \wedge nR_{k'} m'$, we take $k'' = \max\{k, k'\}$ then it holds $nR_{k''} m \wedge nR_{k''} m'$ then by hypothesis $m = m'$. \square

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:

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

where we see ω_\perp as the flat order obtained by adding \perp to the discrete order of the natural numbers. We define the following order on P'

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

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:

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

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

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

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 $f : D \rightarrow D$ is monotone if and only if

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

We will say that the 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 and c 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:

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

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 f preserves the 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:

$$0 \leq 2 \leq 4 \leq 6 \leq \dots$$

so we have

$$f\left(\bigsqcup d_i\right) = f(\infty) = 1 \quad \neq \quad \bigsqcup f(d_i) = 0$$

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:

$$f(d) = d$$

The set of fixpoints of a function f is denoted as $Fix(f)$.

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

Next theorem ensures that the least fixpoint exists and that it can be found.

Theorem 4.35 (Fixpoint theorem)

Let $f : D \rightarrow D$ be a continuous function on the complete partial order with bottom CPO_{\perp} . Then

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

has the following properties:

1. $fix(f)$ is a fixpoint

$$f(fix(f)) = fix(f)$$

2. $fix(f)$ is the least pre-fixpoint of f

$$f(d) \sqsubseteq d \Rightarrow fix(f) \sqsubseteq d$$

so $fix(f)$ is the least fixpoint of f .

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

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

Now we have a new chain :

$$f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq \dots$$

Now we have:

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

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

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

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

So $fix(f) \sqsubseteq d$ 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 $fix(f)$.

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 $lfp(Id)$ 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 $\sqsubseteq \upharpoonright \omega = \leq$, $\forall d \in \omega \cup \{\infty_1\}. d \sqsubseteq \infty_1$ 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 \mathcal{F} of the well-formed formulas of the language handled by the rules. We define an operator on the CPO $(2^{\mathcal{F}}, \subseteq)$.

Definition 4.38 (Immediate consequence operator \hat{R})

Let R be a set of inference rules and let $B \subseteq \mathcal{F}$ be a set of well-formed formulas, we define:

$$\hat{R}(B) = \{y \mid \exists (X/y) \in R. X \subseteq B\}$$

$\hat{R} : 2^{\mathcal{F}} \rightarrow 2^{\mathcal{F}}$ 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 $y \in \hat{R}(B_2)$. \square

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 $X = \{x_i\}_{i=1, \dots, N}$, then $k_i = \min\{j \mid x_i \in B_j\}$ and $k = \max\{k_i\}_{i=1, \dots, N}$. 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

$$\begin{aligned} I_R^0 &\stackrel{\text{def}}{=} \emptyset \\ I_R^{n+1} &\stackrel{\text{def}}{=} \hat{R}(I_R^n) \cup I_R^n \end{aligned}$$

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:

$$I_R^n = \hat{R}^n(\emptyset)$$

Proof. By induction on n

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

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

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

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

Theorem 4.42 (Fixpoint of \hat{R})

Let R a rule system, it holds:

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

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

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

as required. \square

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

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

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 $\forall \{B_n\}_{n \in \omega}. \bigcup_{n \in \omega} \hat{R}(B_n) = \hat{R}(\bigcup_{n \in \omega} B_n)$, does not hold in this case. Indeed if we take the chain

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

We have :

$$\begin{array}{llll} B_i & \{P(1)\} \subseteq & \{P(1), P(3)\} \subseteq & \{P(1), P(3), P(5)\} \\ \hat{R}(B_i) & \{P(1), P(2)\} \subseteq & \{P(1), P(2), P(4)\} \subseteq & \{P(1), P(2), P(4), P(6)\} \end{array}$$

Then we have:

$$\begin{aligned} \bigcup_{i \in \omega} B_i &= \{P(1), P(3), P(5), \dots\} \\ \bigcup_{i \in \omega} \hat{R}(B_i) &= \{P(1), P(2), P(4), \dots\} \\ \hat{R}\left(\bigcup_{i \in \omega} B_i\right) &= \{P(1), P(2), P(4) \dots \underbrace{P(0)}_{\text{3rd rule}}\} \end{aligned}$$

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

$$\begin{aligned} \text{fix}(\hat{R}) &= \bigcup_{n \in \omega} \hat{R}^n(\emptyset) = \{P(1), P(2), P(3), P(4), \dots\} \\ \hat{R}(\text{fix}(\hat{R})) &= \{P(0), P(1), P(2), P(3), P(4), \dots\} \end{aligned}$$

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

$$S ::= \lambda | (S) | S S$$

Obviously using the inference rule formalism we can write:

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

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

$$S_0 = \hat{R}^0(\emptyset) = \emptyset$$

$$S_1 = \hat{R}(S_0) = \lambda + (\emptyset) + \emptyset\emptyset = \{\lambda\}$$

$$S_2 = \hat{R}(S_1) = \lambda + (\lambda) + \lambda\lambda = \{\lambda, ()\}$$

$$S_3 = \hat{R}(S_2) = \lambda + (\lambda) + ((\)) + ()() = \{\lambda, (), ((\)), ()()\}$$

...

So $\mathcal{L} = \text{fix}(\hat{R})$.

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

We define lambda terms as:

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

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' .
- $t \rightarrow t', t''$ 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.

Definition 5.2

Let t, t' and t'' be three lambda terms, we define:

$$t \rightarrow t', t'' = \begin{cases} t' & \text{if } t = \text{True} \\ t'' & \text{if } t = \text{False} \end{cases}$$

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

Let t and t' be two lambda terms, we define:

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

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

Definition 5.4

Let t, t', t'' and t''' be four lambda terms, we define:

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

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

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

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.

- $A \rightarrow B \times C = A \rightarrow (B \times C)$
- $A \times B \rightarrow C = (A \times B) \rightarrow C$
- $A \times B \times C = (A \times B) \times C$
- $A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)$
- $tt't'' = (tt')t''$
- $\lambda x.tt' = \lambda x.(tt')$

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

5.2.1. Function \mathcal{A}

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

$$\mathcal{A} : Aexp \rightarrow \Sigma \rightarrow \mathbb{N}$$

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

Notation

We will rely on definitions of the form

$$\mathcal{A} \llbracket n \rrbracket = \lambda \sigma . n$$

with the following meaning:

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

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

$$\mathcal{A} \llbracket n \rrbracket \sigma = n$$

instead of

$$\mathcal{A} \llbracket n \rrbracket = \lambda \sigma . n$$

In this way, we simplify the notation in the right-hand side. Notice that now both the sides of the equation have the type \mathbb{N} .

Definition 5.7

The denotational semantics of arithmetic expressions is defined by structural recursion as follows:

$$\begin{aligned}
 \mathcal{A} \llbracket n \rrbracket \sigma &= n \\
 \mathcal{A} \llbracket x \rrbracket \sigma &= \sigma x \\
 \mathcal{A} \llbracket a_0 + a_1 \rrbracket \sigma &= (\mathcal{A} \llbracket a_0 \rrbracket \sigma) + (\mathcal{A} \llbracket a_1 \rrbracket \sigma) \\
 \mathcal{A} \llbracket a_0 - a_1 \rrbracket \sigma &= (\mathcal{A} \llbracket a_0 \rrbracket \sigma) - (\mathcal{A} \llbracket a_1 \rrbracket \sigma) \\
 \mathcal{A} \llbracket a_0 \times a_1 \rrbracket \sigma &= (\mathcal{A} \llbracket a_0 \rrbracket \sigma) \times (\mathcal{A} \llbracket a_1 \rrbracket \sigma)
 \end{aligned}$$

Let us briefly comment the above definitions.

Constants The denotational semantics of any constant n is just the constant function that 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 “ \times ” 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 \mathcal{B}

Function \mathcal{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

The denotational semantics of boolean expressions is defined by structural recursion as follows:

$$\begin{aligned}
 \mathcal{B} \llbracket v \rrbracket \sigma &= v \\
 \mathcal{B} \llbracket a_0 = a_1 \rrbracket \sigma &= (\mathcal{A} \llbracket a_0 \rrbracket \sigma) = (\mathcal{A} \llbracket a_1 \rrbracket \sigma) \\
 \mathcal{B} \llbracket a_0 \leq a_1 \rrbracket \sigma &= (\mathcal{A} \llbracket a_0 \rrbracket \sigma) \leq (\mathcal{A} \llbracket a_1 \rrbracket \sigma) \\
 \mathcal{B} \llbracket \neg b_0 \rrbracket \sigma &= \neg (\mathcal{B} \llbracket b_0 \rrbracket \sigma) \\
 \mathcal{B} \llbracket b_0 \vee b_1 \rrbracket \sigma &= (\mathcal{B} \llbracket b_0 \rrbracket \sigma) \vee (\mathcal{B} \llbracket b_1 \rrbracket \sigma) \\
 \mathcal{B} \llbracket b_0 \wedge b_1 \rrbracket \sigma &= (\mathcal{B} \llbracket b_0 \rrbracket \sigma) \wedge (\mathcal{B} \llbracket b_1 \rrbracket \sigma)
 \end{aligned}$$

5.2.3. Function \mathcal{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:

$$\mathcal{C} : Com \rightarrow (\Sigma \rightarrow \Sigma)$$

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

$$\mathcal{C} : Com \rightarrow (\Sigma \rightarrow \Sigma_{\perp})$$

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

$$\begin{aligned}\mathcal{C} \llbracket \text{skip} \rrbracket \sigma &= \sigma \\ \mathcal{C} \llbracket x := a \rrbracket \sigma &= \sigma \left[\mathcal{A} \llbracket a \rrbracket \sigma / x \right]\end{aligned}$$

We see that “skip” does not modify the memory, while “ $x := a$ ” evaluates the arithmetic expression “ a ” with its function \mathcal{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 $\mathcal{C} \llbracket c_0 \rrbracket$ we obtain a value in Σ_{\perp} so we can not apply $\mathcal{C} \llbracket c_1 \rrbracket$. To work this problem out we introduce a *lifting* operator $_*$.

Definition 5.9

Let $f : \Sigma \rightarrow \Sigma_{\perp}$, we define a function $f^* : \Sigma_{\perp} \rightarrow \Sigma_{\perp}$ as follows:

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

Note that $_*$ is an operator of the type $(\Sigma \rightarrow \Sigma_{\perp}) \rightarrow (\Sigma_{\perp} \rightarrow \Sigma_{\perp})$.

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

$$\mathcal{C} \llbracket c_0; c_1 \rrbracket \sigma = \mathcal{C} \llbracket c_1 \rrbracket^* (\mathcal{C} \llbracket c_0 \rrbracket \sigma)$$

Let us now consider the conditional command “if”, recall that the λ -calculus provides a conditional operator, then we have simply:

$$\mathcal{C} \llbracket \text{if } b \text{ then } c_0 \text{ else } c_1 \rrbracket \sigma = \mathcal{B} \llbracket b \rrbracket \sigma \rightarrow \mathcal{C} \llbracket c_0 \rrbracket \sigma, \mathcal{C} \llbracket c_1 \rrbracket \sigma$$

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

$$\mathcal{C} \llbracket \text{while } b \text{ do } c \rrbracket \sigma = \mathcal{B} \llbracket b \rrbracket \sigma \rightarrow \mathcal{C} \llbracket \text{while } b \text{ do } c \rrbracket^* (\mathcal{C} \llbracket c \rrbracket \sigma), \sigma$$

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 $\Gamma_{b,c}$.

$$\Gamma_{b,c} = \lambda\varphi. \lambda\sigma. \underbrace{\underbrace{\mathcal{B} \llbracket b \rrbracket \sigma \rightarrow \varphi^*(\mathcal{C} \llbracket c \rrbracket \sigma), \sigma}_{\Sigma_{\perp}}}_{\Sigma \rightarrow \Sigma_{\perp}}}_{(\Sigma \rightarrow \Sigma_{\perp}) \rightarrow \Sigma \rightarrow \Sigma_{\perp}}$$

As we can see the function $\Gamma_{b,c}$ is of type $(\Sigma \rightarrow \Sigma_{\perp}) \rightarrow \Sigma \rightarrow \Sigma_{\perp}$ and contains only subterms of “while b 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:

$$\mathcal{C} \llbracket \text{while } b \text{ do } c \rrbracket = \text{fix } \Gamma_{b,c} = \bigsqcap_{n \in \omega} \Gamma_{b,c}^n (\perp_{\Sigma \rightarrow \Sigma_{\perp}})$$

To prove continuity we will consider $\Gamma_{b,c}$ as applied to partial functions: $\Gamma_{b,c} : (\Sigma \rightarrow \Sigma) \rightarrow (\Sigma \rightarrow \Sigma)$. Partial functions in $\Sigma \rightarrow \Sigma$ can be represented as sets of (well formed) formulas $\sigma \rightarrow \sigma'$. Then the effect of $\Gamma_{b,c}$ can be represented by the immediate consequence operators for the following (infinite) set of rules.

$$R_{\Gamma_{b,c}} = \left\{ \frac{\mathcal{B} \llbracket b \rrbracket \sigma \quad \mathcal{C} \llbracket c \rrbracket \sigma = \sigma'' \quad \sigma'' \rightarrow \sigma'}{\sigma \rightarrow \sigma'}, \quad \frac{\neg \mathcal{B} \llbracket b \rrbracket \sigma}{\sigma \rightarrow \sigma} \right\} \text{ with } \hat{R}_{\Gamma_{b,c}} = \Gamma_{b,c}$$

Notice that here the only well formed formulas are $\sigma'' \rightarrow \sigma'$, $\sigma \rightarrow \sigma'$ and $\sigma \rightarrow \sigma$. An instance of the first rule schema is obtained by picking up a value of σ such that $\mathcal{B}[[b]]\sigma$ is true, and a (the) value of σ'' such that $\mathcal{C}[[c]]\sigma = \sigma''$. Then for every σ'' such that $\sigma'' \rightarrow \sigma'$ we can imply $\sigma \rightarrow \sigma'$. 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 th.4.40, which ensures the continuity of $\hat{R}_{\Gamma_{b,c}}$. Now by using th.4.3.2 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 results we achieved.

Example 5.10

Let us consider the command:

$$w = \text{while true do skip}$$

now we will see how to calculate its semantics. We have $\mathcal{C}[[w]] = \text{fix } \Gamma_{\text{true,skip}}$ where

$$\Gamma_{\text{true,skip}}\varphi\sigma = \mathcal{B}[[\text{true}]]\sigma \rightarrow \varphi^*(\mathcal{C}[[\text{skip}]]\sigma), \sigma = \varphi^*(\mathcal{C}[[\text{skip}]]\sigma) = \varphi^*\sigma = \varphi\sigma$$

So we have $\Gamma_{\text{true,skip}}\varphi = \varphi$, that is $\Gamma_{\text{true,skip}}$ is the identity function. Then each function φ is a fixpoint of $\Gamma_{\text{true,skip}}$, 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 $\text{fix } \Gamma_{\text{true,skip}} = \lambda\sigma.\perp_{\Sigma_{\perp}}$.

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 $w = \text{while } b \text{ do } c$ and $\text{if } b \text{ then } c; w \text{ else skip}$. Since $\mathcal{C}[[w]]$ is a fixpoint we have:

$$\mathcal{C}[[w]] = \Gamma(\mathcal{C}[[w]]) = \lambda\sigma.\mathcal{B}[[b]]\sigma \rightarrow \mathcal{C}[[w]]^*(\mathcal{C}[[c]]\sigma), \sigma$$

Moreover we have:

$$\begin{aligned} \mathcal{C}[[\text{if } b \text{ then } c; w \text{ else skip}]] &= \lambda\sigma.\mathcal{B}[[b]]\sigma \rightarrow \mathcal{C}[[c;w]]\sigma, \mathcal{C}[[\text{skip}]]\sigma \\ &= \lambda\sigma.\mathcal{B}[[b]]\sigma \rightarrow \mathcal{C}[[w]]^*(\mathcal{C}[[c]]\sigma), \sigma \end{aligned}$$

Example 5.12

Let us consider the command:

$$c = \text{while } x \neq 0 \text{ do } x := x - 1$$

we have:

$$\mathcal{C}[[c]] = \text{fix } \Gamma$$

where $\Gamma\varphi = \lambda\sigma.\sigma x \neq 0 \rightarrow \varphi\sigma[\sigma x - 1/x], \sigma$. Let us see some iterations:

$$\begin{aligned} \varphi_0 &= \Gamma^0 \perp_{\Sigma \rightarrow \Sigma_{\perp}} = \perp_{\Sigma \rightarrow \Sigma_{\perp}} = \lambda\sigma.\perp_{\Sigma_{\perp}} \\ \varphi_1 &= \Gamma \varphi_0 = \lambda\sigma.\sigma x \neq 0 \rightarrow \perp_{\Sigma_{\perp}}, \sigma \\ \varphi_2 &= \Gamma \varphi_1 = \lambda\sigma.\sigma x \neq 0 \rightarrow (\lambda\sigma'.\sigma' x \neq 0 \rightarrow \perp_{\Sigma_{\perp}}, \sigma') \sigma[\sigma x - 1/x], \sigma \end{aligned}$$

Now we have the followings possibilities:

$$\frac{\sigma x < 0}{\varphi_2\sigma = \perp} \quad \frac{\sigma x = 0}{\varphi_2\sigma = \sigma} \quad \frac{\sigma x = 1}{\varphi_2\sigma = \sigma[\sigma x - 1/x]} \quad \frac{\sigma x > 1}{\varphi_2\sigma = \perp}$$

So we have:

$$\varphi_2 = \lambda \sigma. \sigma x < 0 \rightarrow \perp, 0 \leq \sigma x < 2 \rightarrow \sigma[0/x], \perp$$

We can now attempt to formulate a conjecture:

$$\forall n \in \omega. \varphi_n = \lambda \sigma. \sigma x < 0 \rightarrow \perp, 0 \leq \sigma x < n \rightarrow \sigma[0/x], \perp$$

We are now ready to prove our conjecture by mathematical induction.

The base case is trivial, indeed $\varphi_0 = \Gamma^0 \perp = \lambda \sigma. \perp_{\Sigma \perp}$.

For the inductive case. let us now assume the predicate for n and prove it for $n + 1$. So as usual we assume:

$$P(n) = (\varphi_n = \lambda \sigma. \sigma x < 0 \rightarrow \perp, 0 \leq \sigma x < n \rightarrow \sigma[0/x], \perp)$$

and prove:

$$P(n + 1) = (\varphi_{n+1} = \lambda \sigma. \sigma x < 0 \rightarrow \perp, 0 \leq \sigma x < n + 1 \rightarrow \sigma[0/x], \perp)$$

By definition:

$$\varphi_{n+1} = \Gamma \varphi_n = \lambda \sigma. \sigma x \neq 0 \rightarrow (\lambda \sigma'. \sigma' x < 0 \rightarrow \perp, 0 \leq \sigma' x < n \rightarrow \sigma'[0/x], \perp) \sigma[\sigma x - 1/x], \sigma$$

we have:

$$\frac{\sigma x < 0}{\varphi_{n+1} \sigma = \perp} \quad \frac{\sigma x = 0}{\varphi_{n+1} \sigma = \sigma = \sigma[0/x]}$$

$$\frac{1 \leq \sigma x < n + 1}{\varphi_{n+1} \sigma = \sigma[\sigma x - 1/x][0/x] = \sigma[0/x]} \quad \frac{\sigma x \geq n + 1}{\varphi_{n+1} \sigma = \perp}$$

Then:

$$\varphi_{n+1} = \lambda \sigma. \sigma x < 0 \rightarrow \perp, 0 \leq \sigma x < n + 1 \rightarrow \sigma[0/x], \perp$$

Finally we have:

$$\mathcal{C} \llbracket c \rrbracket = \text{fix } \Gamma = \bigsqcup_{i \in \omega} \varphi^i \perp = \lambda \sigma. \sigma x < 0 \rightarrow \perp, \sigma[0/x]$$

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 \mathcal{A} and \mathcal{B}

We start from the arithmetic expressions, the property which we will prove is the following:

$$P(a) \stackrel{\text{def}}{=} \langle a, \sigma \rangle \rightarrow \mathcal{A} \llbracket a \rrbracket \sigma \quad \forall a \in A \text{ expr}$$

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 $P(n) \stackrel{\text{def}}{=} \langle n, \sigma \rangle \rightarrow \mathcal{A} \llbracket n \rrbracket \sigma$ Since $\mathcal{A} \llbracket n \rrbracket \sigma = n$ and $\langle n, \sigma \rangle \rightarrow n$.

Variables $P(x) \stackrel{\text{def}}{=} \langle x, \sigma \rangle \rightarrow \mathcal{A} \llbracket x \rrbracket \sigma$ Since $\mathcal{A} \llbracket x \rrbracket \sigma = \sigma(x)$ and $\langle x, \sigma \rangle \rightarrow \sigma(x)$.

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 $P(a_0) \stackrel{\text{def}}{=} \langle a_0, \sigma \rangle \rightarrow \mathcal{A} \llbracket a_0 \rrbracket \sigma$ and $P(a_1) \stackrel{\text{def}}{=} \langle a_1, \sigma \rangle \rightarrow \mathcal{A} \llbracket a_1 \rrbracket \sigma$ and we prove $P(a_0 \circ a_1) \stackrel{\text{def}}{=} \langle a_0 \circ a_1, \sigma \rangle \rightarrow \mathcal{A} \llbracket a_0 \circ a_1 \rrbracket \sigma$.

We have:

$$\begin{aligned} \langle a_0 \circ a_1, \sigma \rangle \rightarrow \mathcal{A} \llbracket a_0 \rrbracket \sigma \odot \mathcal{A} \llbracket a_1 \rrbracket \sigma & \text{ by operational semantics definition and using the preconditions} \\ \langle a_0 \circ a_1, \sigma \rangle \rightarrow \mathcal{A} \llbracket a_0 \circ a_1 \rrbracket \sigma & \text{ by denotational semantics definition} \end{aligned}$$

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 \mathcal{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 formalism, 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:

$$\forall c. \forall \sigma, \sigma' \in \Sigma. \langle c, \sigma \rangle \rightarrow \sigma' \iff \mathcal{C} \llbracket c \rrbracket \sigma = \sigma'$$

As usual we will divide the proof in two parts:

- $P(\langle c, \sigma \rangle \rightarrow \sigma') \stackrel{\text{def}}{=} \mathcal{C} \llbracket c \rrbracket \sigma = \sigma'$ (Completeness)
- $P(c) \stackrel{\text{def}}{=} \mathcal{C} \llbracket c \rrbracket \sigma = \sigma' \Rightarrow \langle c, \sigma \rangle \rightarrow \sigma'$ (Correctness)

Notice that in this way it also guaranteed equivalence for the non defined cases: for instance we have $\langle c, \sigma \rangle \not\rightarrow \sigma' \iff \mathcal{C} \llbracket c \rrbracket \sigma = \perp_{\Sigma_{\perp}}$ since otherwise $\mathcal{C} \llbracket c \rrbracket \sigma = \sigma'$ would imply $\langle c, \sigma \rangle \rightarrow \sigma'$. Similarly in the other direction.

5.3.2.1. Completeness of the Denotational Semantics

Let us prove the first part of the theorem: $P(\langle c, \sigma \rangle \rightarrow \sigma') \stackrel{\text{def}}{=} \mathcal{C} \llbracket c \rrbracket \sigma = \sigma'$.

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 We would like to prove:

- $P(\langle \text{skip}, \sigma \rangle \rightarrow \sigma) \stackrel{\text{def}}{=} \mathcal{C} \llbracket \text{skip} \rrbracket \sigma = \sigma$

Obviously the proposition is true by definition of denotational semantic.

Assignment Let us consider the assignment command:

$$\frac{\langle a, \sigma \rangle \rightarrow m}{\langle x := a, \sigma \rangle \rightarrow \sigma [^m/x]}$$

We assume:

- $P(\langle a, \sigma \rangle \rightarrow m) \stackrel{\text{def}}{=} \mathcal{A} \llbracket a \rrbracket \sigma = m$.

We will prove:

- $P(\langle x := a, \sigma \rangle \rightarrow \sigma [^m/x]) \stackrel{\text{def}}{=} \mathcal{C} \llbracket x := a \rrbracket \sigma = \sigma [^m/x]$

We have by the definition of denotational semantics:

$$\mathcal{C} \llbracket x := a \rrbracket \sigma = \sigma \left[\mathcal{A} \llbracket a \rrbracket \sigma / x \right] = \sigma [^m/x]$$

Concatenation Let us consider the concatenation rule:

$$\frac{\langle c_0, \sigma \rangle \rightarrow \sigma'' \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma'}{\langle c_0; c_1, \sigma \rangle \rightarrow \sigma'}$$

We assume:

- $P(\langle c_0, \sigma \rangle \rightarrow \sigma'') \stackrel{\text{def}}{=} \mathcal{C} \llbracket c_0 \rrbracket \sigma = \sigma''$
- $P(\langle c_1, \sigma'' \rangle \rightarrow \sigma') \stackrel{\text{def}}{=} \mathcal{C} \llbracket c_1 \rrbracket \sigma'' = \sigma'$.

We will prove:

- $P(\langle c_0; c_1, \sigma \rangle \rightarrow \sigma') \stackrel{\text{def}}{=} \mathcal{C} \llbracket c_0; c_1 \rrbracket \sigma = \sigma'$

We have by definition:

$$\mathcal{C} \llbracket c_0; c_1 \rrbracket \sigma = \mathcal{C} \llbracket c_1 \rrbracket^* (\mathcal{C} \llbracket c_0 \rrbracket \sigma) = \mathcal{C} \llbracket c_1 \rrbracket^* \sigma'' = \mathcal{C} \llbracket c_1 \rrbracket \sigma'' = \sigma'$$

Note that the lifting operator can be deleted because $\sigma'' \neq \perp$.

Conditional For the conditional command we have two rules:

$$\frac{\langle b, \sigma \rangle \rightarrow \mathbf{true} \quad \langle c_0, \sigma \rangle \rightarrow \sigma'}{\langle \mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma \rangle \rightarrow \sigma'} \quad \frac{\langle b, \sigma \rangle \rightarrow \mathbf{false} \quad \langle c_1, \sigma \rangle \rightarrow \sigma'}{\langle \mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma \rangle \rightarrow \sigma'}$$

We will prove only the first case, the second proof is completely analogous, so we leave that as an exercise.

We assume:

- $P(\langle b, \sigma \rangle \rightarrow \mathbf{true}) \stackrel{\text{def}}{=} \mathcal{B} \llbracket b \rrbracket \sigma = \mathbf{true}$
- $P(\langle c_0, \sigma \rangle \rightarrow \sigma') \stackrel{\text{def}}{=} \mathcal{C} \llbracket c_0 \rrbracket \sigma = \sigma'$.

And we will prove:

- $P(\langle \mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma \rangle \rightarrow \sigma') \stackrel{\text{def}}{=} \mathcal{C} \llbracket \mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1 \rrbracket \sigma = \sigma'$

We have:

$$\mathcal{C} \llbracket \mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1 \rrbracket \sigma = \mathcal{B} \llbracket b \rrbracket \sigma \rightarrow \mathcal{C} \llbracket c_0 \rrbracket \sigma, \mathcal{C} \llbracket c_1 \rrbracket \sigma = \mathcal{C} \llbracket c_0 \rrbracket \sigma = \sigma'$$

While As for the conditional we have two rules for the “while” command:

$$\frac{\langle b, \sigma \rangle \rightarrow \mathbf{false}}{\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \sigma} \quad \frac{\langle b, \sigma \rangle \rightarrow \mathbf{true} \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle \mathbf{while } b \mathbf{ do } c, \sigma'' \rangle \rightarrow \sigma'}{\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \sigma'}$$

Let us consider the first rule. We assume:

- $P(\langle b, \sigma \rangle \rightarrow \mathbf{false}) \stackrel{\text{def}}{=} \mathcal{B} \llbracket b \rrbracket \sigma = \mathbf{false}$

We will prove:

- $P(\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \sigma) \stackrel{\text{def}}{=} \mathcal{C} \llbracket \mathbf{while } b \mathbf{ do } c \rrbracket \sigma = \sigma$

We have by the fixpoint property of the denotational semantics:

$$\mathcal{C} \llbracket \mathbf{while } b \mathbf{ do } c \rrbracket \sigma = \mathcal{B} \llbracket b \rrbracket \sigma \rightarrow \mathcal{C} \llbracket \mathbf{while } b \mathbf{ do } c \rrbracket^* (\mathcal{C} \llbracket c \rrbracket \sigma), \sigma = \sigma$$

For the second rule we assume:

- $P(\langle b, \sigma \rangle \rightarrow \mathbf{true}) \stackrel{\text{def}}{=} \mathcal{B} \llbracket b \rrbracket \sigma = \mathbf{true}$
- $P(\langle c, \sigma \rangle \rightarrow \sigma'') \stackrel{\text{def}}{=} \mathcal{C} \llbracket c \rrbracket \sigma = \sigma''$
- $P(\langle \mathbf{while } b \mathbf{ do } c, \sigma'' \rangle \rightarrow \sigma') \stackrel{\text{def}}{=} \mathcal{C} \llbracket \mathbf{while } b \mathbf{ do } c \rrbracket \sigma'' = \sigma'$

We will prove:

- $P(\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \sigma') \stackrel{\text{def}}{=} \mathcal{C} \llbracket \mathbf{while } b \mathbf{ do } c \rrbracket \sigma = \sigma'$

By the definition of the denotational semantics:

$$\mathcal{B} \llbracket b \rrbracket \sigma \rightarrow \mathcal{C} \llbracket \mathbf{while } b \mathbf{ do } c \rrbracket^* (\mathcal{C} \llbracket c \rrbracket \sigma), \sigma = \mathcal{C} \llbracket \mathbf{while } b \mathbf{ do } c \rrbracket^* (\mathcal{C} \llbracket c \rrbracket \sigma) = \mathcal{C} \llbracket \mathbf{while } b \mathbf{ do } c \rrbracket \sigma' = \sigma'$$

Note that the lifting operator can be deleted since $\sigma'' \neq \perp$.

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:

$$P(c) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \mathcal{C} \llbracket c \rrbracket \sigma = \sigma' \Rightarrow \langle c, \sigma \rangle \rightarrow \sigma'$$

Skip We will prove:

- $P(\mathbf{skip}) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \mathcal{C} \llbracket \mathbf{skip} \rrbracket \sigma = \sigma' \Rightarrow \langle \mathbf{skip}, \sigma \rangle \rightarrow \sigma'$

By definition we have $\mathcal{C} \llbracket \mathbf{skip} \rrbracket \sigma = \sigma$ and $\langle \mathbf{skip}, \sigma \rangle \rightarrow \sigma$.

Assignment We will prove:

- $P(x := a) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \mathcal{C} \llbracket x := a \rrbracket \sigma = \sigma' \Rightarrow \langle x := a, \sigma \rangle \rightarrow \sigma'$

By definition have $\sigma' = \sigma[n/x]$ where $\mathcal{A} \llbracket a \rrbracket \sigma = n$ so we have $\langle a, \sigma \rangle \rightarrow n$ and thus we can apply the rule:

$$\frac{\langle a, \sigma \rangle \rightarrow n}{\langle x := a, \sigma \rangle \rightarrow \sigma[n/x]}$$

Concatenation We will prove:

$$\bullet P(c_0; c_1) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \mathcal{C} \llbracket c_0; c_1 \rrbracket \sigma = \sigma' \Rightarrow \langle c_0; c_1, \sigma \rangle \rightarrow \sigma'$$

We assume:

$$\bullet P(c_0) \stackrel{\text{def}}{=} \forall \sigma, \sigma'' \mathcal{C} \llbracket c_0 \rrbracket \sigma = \sigma'' \Rightarrow \langle c_0, \sigma \rangle \rightarrow \sigma''$$

$$\bullet P(c_1) \stackrel{\text{def}}{=} \forall \sigma'', \sigma' \mathcal{C} \llbracket c_1 \rrbracket \sigma'' = \sigma' \Rightarrow \langle c_1, \sigma \rangle \rightarrow \sigma'$$

Assuming the premise of the implication we want to prove, we have $\mathcal{C} \llbracket c_0; c_1 \rrbracket \sigma = \mathcal{C} \llbracket c_1 \rrbracket^* (\mathcal{C} \llbracket c_0 \rrbracket \sigma) = \sigma'$. Since we assume the termination of c_0 we can omit the lifting operator. We obtain $\mathcal{C} \llbracket c_0 \rrbracket \sigma = \sigma''$ and $\mathcal{C} \llbracket c_1 \rrbracket \sigma'' = \sigma'$, which allows us to apply modus ponens to the inductive assumptions, obtaining $\langle c_0, \sigma \rangle \rightarrow \sigma''$ and $\langle c_1, \sigma'' \rangle \rightarrow \sigma'$. Thus we can apply:

$$\frac{\langle c_0, \sigma \rangle \rightarrow \sigma'' \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma'}{\langle c_0; c_1, \sigma \rangle \rightarrow \sigma'}$$

Conditional We will prove:

$$\bullet P(\text{if } b \text{ then } c_0 \text{ else } c_1) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \mathcal{C} \llbracket \text{if } b \text{ then } c_0 \text{ else } c_1 \rrbracket \sigma = \sigma' \Rightarrow \langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow \sigma'$$

As usual we must distinguish two cases, let us consider only the case with $\mathcal{B} \llbracket b \rrbracket \sigma = \mathbf{false}$, namely $\langle b, \sigma \rangle \rightarrow \mathbf{false}$

We assume:

$$\bullet P(c_0) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \mathcal{C} \llbracket c_0 \rrbracket \sigma = \sigma' \Rightarrow \langle c_0, \sigma \rangle \rightarrow \sigma'$$

$$\bullet P(c_1) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \mathcal{C} \llbracket c_1 \rrbracket \sigma = \sigma' \Rightarrow \langle c_1, \sigma \rangle \rightarrow \sigma'$$

$$\bullet \mathcal{C} \llbracket \text{if } b \text{ then } c_0 \text{ else } c_1 \rrbracket \sigma = \sigma'$$

By definition and since $\mathcal{B} \llbracket b \rrbracket \sigma = \mathbf{false}$ we have $\mathcal{C} \llbracket \text{if } b \text{ then } c_0 \text{ else } c_1 \rrbracket \sigma = \mathcal{B} \llbracket b \rrbracket \sigma \rightarrow \mathcal{C} \llbracket c_0 \rrbracket \sigma, \mathcal{C} \llbracket c_1 \rrbracket \sigma = \mathcal{C} \llbracket c_1 \rrbracket \sigma = \sigma'$ this implies by hypothesis $\langle c_1, \sigma \rangle \rightarrow \sigma'$. Thus we can apply the rule:

$$\frac{\langle c_1, \sigma \rangle \rightarrow \sigma' \quad \langle b, \sigma \rangle \rightarrow \mathbf{false}}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow \sigma'}$$

While We will prove:

$$P(\text{while } b \text{ do } c) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \mathcal{C} \llbracket \text{while } b \text{ do } c \rrbracket \sigma = \sigma' \Rightarrow \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma'$$

We assume by structural induction:

$$\bullet P(c) \stackrel{\text{def}}{=} \forall \sigma, \sigma'' \mathcal{C} \llbracket c \rrbracket \sigma = \sigma'' \Rightarrow \langle c, \sigma \rangle \rightarrow \sigma''$$

By definition $\mathcal{C} \llbracket \text{while } b \text{ do } c \rrbracket \sigma = \text{fix } \Gamma_{b,c} \sigma$ so:

$$\begin{aligned} P(\text{while } b \text{ do } c) &= \mathcal{C} \llbracket \text{while } b \text{ do } c \rrbracket \sigma = \sigma' \Rightarrow \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma' \\ &= \text{fix } \Gamma_{b,c} \sigma = \sigma' \Rightarrow \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma' \\ &= \left(\bigsqcup_{i \in \omega} \Gamma_{b,c}^i \perp \right) \sigma = \sigma' \Rightarrow \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma' \\ &= \forall n. (\Gamma_{b,c}^n \perp \sigma = \sigma' \Rightarrow \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma') \end{aligned}$$

Notice that the last two properties are equivalent. In fact, if there is a pair $\sigma \rightarrow \sigma'$ in the limit, it must also occur for some n . Vice versa, if it belongs to $\Gamma_{b,c}^n \perp$ then it belongs also to the limit. We prove the last implication by mathematical induction

P(0) We have to prove:

$$\Gamma_{b,c}^0 \perp \sigma = \sigma' \Rightarrow \langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle \rightarrow \sigma'$$

Obviously:

$$\perp \sigma = \sigma' \Rightarrow \langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle \rightarrow \sigma'$$

Since $\perp \sigma = \sigma'$ is always false then the implication is true.

P(n) \Rightarrow P(n+1) Let us assume $A(n) \stackrel{\text{def}}{=} \Gamma_{b,c}^n \perp \sigma = \sigma' \Rightarrow \langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle \rightarrow \sigma'$. We will show that $A(n+1) \stackrel{\text{def}}{=} \Gamma_{b,c}^{n+1} \perp \sigma = \sigma' \Rightarrow \langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle \rightarrow \sigma'$. We assume the premise of the implication, i.e. $\Gamma_{b,c}(\Gamma_{b,c}^n \perp) \sigma = \sigma'$, that is

$$\mathcal{B} \llbracket b \rrbracket \sigma \rightarrow (\Gamma_{b,c}^n \perp)^* (\mathcal{C} \llbracket c \rrbracket \sigma), \sigma = \sigma'$$

Now we distinguish two cases $\mathcal{B} \llbracket b \rrbracket \sigma = \mathbf{false}$ and $\mathcal{B} \llbracket b \rrbracket \sigma = \mathbf{true}$.

- if $\mathcal{B} \llbracket b \rrbracket \sigma = \mathbf{false}$, we have $\sigma' = \sigma$. As proved in the previous sections $\mathcal{B} \llbracket b \rrbracket \sigma = \mathbf{false} \Leftrightarrow \langle b, \sigma \rangle \rightarrow \mathbf{false}$. Now by using the rule:

$$\frac{\langle b, \sigma \rangle \rightarrow \mathbf{false}}{\langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle \rightarrow \sigma}$$

we have $\langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle \rightarrow \sigma$ as required.

- if $\mathcal{B} \llbracket b \rrbracket \sigma = \mathbf{true}$ as for the false case $\langle b, \sigma \rangle \rightarrow \mathbf{true}$. The premise of the implication becomes $\sigma' = (\Gamma_{b,c}^n \perp)^* (\mathcal{C} \llbracket c \rrbracket \sigma)$. We can omit the lifting operator since we are working with terminating commands. So we have $\mathcal{C} \llbracket c \rrbracket \sigma = \sigma''$ and by structural induction $\langle c, \sigma \rangle \rightarrow \sigma''$. Now we have by mathematical induction hypothesis $A(n) = \Gamma_{b,c}^n \perp \sigma'' = \sigma' \Rightarrow \langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma'' \rangle \rightarrow \sigma'$. Finally by using the rule:

$$\frac{\langle b, \sigma \rangle \rightarrow \mathbf{true} \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma'' \rangle \rightarrow \sigma'}{\langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle \rightarrow \sigma'}$$

we have $\langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle \rightarrow \sigma'$ 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.

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 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 not inclusive, indeed, $\bigsqcup_{n \in \omega} a^n = a^\infty$. Fairness is the property of an arbiter which does not favor one of two competitors all the times from some point on.

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:

$$\frac{P \text{ inclusive} \quad \perp \in P \quad \forall d \in D. (d \in P \implies F(d) \in P)}{\text{fix}(F) \in P}$$

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 $\bigsqcup_{n \in \omega} F^n(\perp) = \text{fix}(F) \in P$ since P is inclusive. \square

Example 5.16 (Computational induction)

$$\mathcal{C} \llbracket \text{while } x \neq 0 \text{ do } x := x - 1 \rrbracket \sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[{}^0/x]$$

By definition:

$$\mathcal{C} \llbracket w \rrbracket = \text{fix}(\Gamma) \quad \Gamma \varphi \sigma = \sigma x \neq 0 \rightarrow \varphi \sigma[{}^{\sigma x - 1}/x], \sigma$$

Let us rewrite the property:

$$P(\varphi) \stackrel{\text{def}}{=} \forall \sigma. (\varphi \sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[{}^0/x])$$

we will show that the property is inclusive, that is:

$$\begin{aligned} & (\forall i \forall \sigma. (\varphi_i \sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[{}^0/x])) \Rightarrow \\ & \Rightarrow \forall \sigma. ((\bigsqcup_{i \in \omega} \varphi_i) \sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[{}^0/x]) \end{aligned}$$

Let us assume $(\bigsqcup_{i \in \omega} \varphi_i) \sigma = \sigma'$ so we have:

$$\exists k. \varphi_k \sigma = \sigma'$$

then we can conclude the thesis by modus ponens on the premise.

We can now use the computational induction:

$$\frac{P \text{ inclusive} \quad \perp \in P \quad \forall \varphi. P(\varphi) \Rightarrow P(\Gamma \varphi)}{P(\mathcal{C} \llbracket \text{while } x \neq 0 \text{ do } x := x - 1 \rrbracket)}$$

- $P(\perp)$ obviously since $\perp \sigma = \sigma'$ is always false.

- $P(\varphi) \stackrel{?}{\implies} P(\Gamma\varphi)$

Let us assume

$$P(\varphi) = \forall\sigma.(\varphi\sigma = \sigma' \implies \sigma x \geq 0 \wedge \sigma' = \sigma[0/x])$$

we will show

$$P(\Gamma\varphi) = \Gamma\varphi\sigma = \sigma' \stackrel{?}{\implies} \sigma x \geq 0 \wedge \sigma' = \sigma[0/x]$$

we assume the premise $\Gamma\varphi\sigma = \sigma'$

$$\Gamma\varphi\sigma = \sigma x \neq 0 \rightarrow \varphi\sigma[\sigma^{x-1}/x], \sigma = \sigma'$$

if $\sigma x = 0$:

$$\sigma = \sigma' \quad \sigma x \geq 0 \quad \sigma' = \sigma[0/x]$$

and the consequence holds.

if $\sigma x \neq 0$, we use $P(\varphi)$ for the argument σ''

$$\varphi \underbrace{\sigma[\sigma^{x-1}/x]}_{\sigma''} = \sigma' \implies \sigma'' x \geq 0 \wedge \sigma' = \sigma''[0/x]$$

we have:

$$\sigma'' x \geq 0 \Leftrightarrow \sigma[\sigma^{x-1}/x] x \geq 0 \Leftrightarrow \sigma x - 1 \geq 0 \Leftrightarrow \sigma x \geq 1 \implies \sigma x \geq 0$$

$$\sigma' = \sigma''[0/x] = \sigma[\sigma^{x-1}/x][0/x] = \sigma[0/x]$$

And the consequence holds also in this case. By computational induction the property holds for the “while” command.

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	Variable
	n	Constant
	$t_1 + t_2$ $t_1 - t_2$ $t_1 \times t_2$	Arithmetic Operators
	if t then t_1 else t_2	Conditional (it reads if $t = 0$ then t_1 else t_2)
	(t_1, t_2) fst (t) snd (t)	Pairing and Projection Operators
	$\lambda x.t$ $(t_1 t_2)$	Function Abstraction and Application
	rec $x.t$	Recursion

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 **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:

$$\tau ::= \text{int} \mid \tau * \tau \mid \tau \rightarrow \tau$$

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 $(\text{int} * \text{int}) \rightarrow \text{int}$ and $\text{int} \rightarrow (\text{int} * (\text{int} \rightarrow \text{int}))$. 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 $x : \text{type}(x) = \hat{x}$ where *type* is a function which assigns a type to each variable.

Arithmetic operators

$$n : \text{int} \quad \frac{t_1 : \text{int} \quad t_2 : \text{int}}{t_1 \text{ op } t_2 : \text{int}} \text{ with op} = +, -, \times \quad \frac{t_0 : \text{int} \quad t_1 : \tau \quad t_2 : \tau}{\text{if } t_0 \text{ then } t_1 \text{ else } t_2 : \tau}$$

Pairings

$$\frac{t_1 : \tau_1 \quad t_2 : \tau_2}{(t_1, t_2) : \tau_1 * \tau_2} \quad \frac{t : \tau_1 * \tau_2}{\mathbf{fst}(t) : \tau_1} \quad \frac{t : \tau_1 * \tau_2}{\mathbf{snd}(t) : \tau_2}$$

Functions

$$\frac{x : \tau_1 \quad t : \tau_2}{\lambda x.t : \tau_1 \rightarrow \tau_2} \quad \frac{t_1 : \tau_1 \rightarrow \tau_2 \quad t_2 : \tau_1}{(t_1 \ t_2) : \tau_2}$$

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:

$$\begin{aligned} FV(n) &= \emptyset \\ FV(x) &= \{x\} \\ FV(t_1 \mathbf{op} t_2) &= FV(t_1) \cup FV(t_2) \\ FV(\mathbf{if} \ t_0 \ \mathbf{then} \ t_1 \ \mathbf{else} \ t_2) &= FV(t_0) \cup FV(t_1) \cup FV(t_2) \\ FV((t_1, t_2)) &= FV(t_1) \cup FV(t_2) \\ FV(\mathbf{fst}(t)) &= FV(\mathbf{snd}(t)) = FV(t) \\ FV(\lambda x.t) &= FV(t) \setminus \{x\} \\ FV((t_1 \ t_2)) &= FV(t_1) \cup FV(t_2) \\ FV(\mathbf{rec} \ x.t) &= FV(t) \setminus \{x\} \end{aligned}$$

Finally as done for λ -calculus we define the substitution operator on HOFL:

$$\begin{aligned} x[t/x] &= t \\ y[t/x] &= y \quad \text{if } x \neq y \\ n[t/x] &= n \\ (t_1 + t_2)[t/x] &= t_1[t/x] + t_2[t/x] \quad (\text{Analogously for } -, \times, \mathbf{if \ then \ else}, \text{ pairing, } \mathbf{fst}, \mathbf{snd} \) \\ (\lambda y.t')[t/x] &= \lambda z.t'[z/y][t/x] \quad z \notin FV(\lambda y.t'), FV(t), \{x\} \quad (\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 for $t' = z : \tau'$. We will prove the statement by structural induction. So we will prove:

$$P(t) = \forall y, t', \tau, \tau'. t : \tau \wedge y, t' : \tau' \Rightarrow t[t'/y] : \tau$$

Let us consider only two cases, we leave the others as an exercise.

- Variable: let z be a variable then we prove

$$P(z) = \forall y, t', \tau, \tau'. z : \tau \wedge y, t' : \tau' \Rightarrow z[t'/y] : \tau$$

Let us assume the premises, now we have two possibilities:

- $z = y$: then $\tau = \tau'$ and $z[t'/y] = t' : \tau$
- $z \neq y$: obvious since $z[t'/y] = z$.

- Lambda abstraction:

$$P(\lambda x.t) = \forall y, t', \tau_1, \tau_2, \tau'. \lambda x.t : \tau_1 \rightarrow \tau_2 \wedge y, t' : \tau' \Rightarrow (\lambda x.t)[t'/y] : \tau_1 \rightarrow \tau_2$$

By inductive hypothesis we have $x : \tau_1$, $t : \tau_2$ and $t[x/z][t'/y] : \tau_2$, by the above exercise and by the inductive hypothesis. Then $\lambda z.t[x/z][t'/y] = (\lambda x.t)[t'/y] : \tau_1 \rightarrow \tau_2$

□

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:

$$t = \mathbf{rec} \ p.\lambda x. (x, (p(x+2)))$$

The term $t0$ should behave like the following infinite list:

$$(0, (2, (4, \dots)))$$

Let us show that this term is not typable:

$$\begin{array}{rcl} t = \mathbf{rec} \ p.\lambda x. (x, (p(x+2))) : \tau & \swarrow_{\hat{p}=\tau} & \\ \lambda x. (x, (p(x+2))) : \tau & \swarrow_{\tau=\tau_1 \rightarrow \tau_2} & \\ x : \tau_1, (x, (p(x+2))) : \tau_2 & \swarrow_{\hat{x}=\tau_1} & \\ (x, (p(x+2))) : \tau_2 & \swarrow_{\tau_2=\tau_3 * \tau_4} & \\ x : \tau_3, (p(x+2)) : \tau_4 & \swarrow_{\hat{x}=\tau_3} & \\ (p(x+2)) : \tau_4 & \swarrow_{\hat{p}=\mathit{int} \rightarrow \tau_4, \hat{x}=\mathit{int}} & \end{array}$$

So we have:

$$\tau_1 = \hat{x} = \mathit{int} = \tau_3 \quad \text{and} \quad \tau_2 = \tau_3 * \tau_4$$

That is:

$$\hat{p} = \tau = \tau_1 \rightarrow \tau_2 = \tau_1 \rightarrow \tau_3 * \tau_4 = \mathit{int} \rightarrow \mathit{int} * \tau_4 = \mathit{int} \rightarrow \tau_4$$

Thus:

$$\mathit{int} * \tau_4 = \tau_4 \text{ which is absurd}$$

The above argument is represented more concisely below:

$$\begin{array}{c} t = \mathbf{rec} \ p.\lambda x_{\mathit{int}}. (x_{\mathit{int}}, (p_{\mathit{int} \rightarrow \tau} (x_{\mathit{int}}+2))) \\ \underbrace{\hspace{10em}}_{\mathit{int}} \\ \underbrace{\hspace{8em}}_{\tau} \\ \underbrace{\hspace{6em}}_{\mathit{int} * \tau} \\ \underbrace{\hspace{6em}}_{\mathit{int} \rightarrow \mathit{int} * \tau = \mathit{int} \rightarrow \tau \Rightarrow \tau = \mathit{int} * \tau} \end{array}$$

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, the first introduced by Church and the second by Curry.

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 $f : int \rightarrow int$. So we can type all the subterms:

$$\begin{array}{c}
 \mathbf{fact} \stackrel{\text{def}}{=} \mathbf{rec} \quad \underbrace{f}_{int \rightarrow int} \quad \underbrace{\lambda x}_{int} \cdot \underbrace{\mathbf{if} \ x}_{int} \ \mathbf{then} \ \underbrace{1}_{int} \ \mathbf{else} \ \underbrace{x}_{int} \times (\underbrace{f}_{int \rightarrow int} \ (\underbrace{x}_{int} - \underbrace{1}_{int})) \quad : int \rightarrow int \\
 \underbrace{\hspace{10em}}_{int} \\
 \underbrace{\hspace{10em}}_{int} \\
 \underbrace{\hspace{10em}}_{int} \\
 \underbrace{\hspace{10em}}_{int \rightarrow int}
 \end{array}$$

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:

$$\begin{array}{l}
 \lambda x. x : \tau \quad \swarrow_{\tau = \tau_1 \rightarrow \tau_2} \\
 x : \tau_1 \quad x : \tau_2 \quad \swarrow_{\hat{x} = \tau_2 = \tau_1} \\
 \square
 \end{array}$$

So we have that the principal type is $\lambda x. x : \tau_1 \rightarrow \tau_1$. 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 $\tau = int \rightarrow int$.

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
    fact(f, x)  $\stackrel{\text{def}}{=} \text{if } x = 0 \text{ then } 1 \text{ else } x \times f(f, x - 1)$ 
    fact(fact, n)
end

```

It can be written in HOFL as follows:

$$\mathbf{fact} \stackrel{\text{def}}{=} \lambda \underset{\tau * \text{int}}{y} . \text{if } \underset{\tau * \text{int}}{\text{snd}(y)} \text{ then } \underset{\text{int}}{1} \text{ else } \underset{\tau * \text{int}}{\text{snd}(y)} \times \left(\underset{\tau * \text{int}}{\text{fst}(y)} \left(\underset{\tau * \text{int}}{\text{fst}(y)} , \underset{\tau * \text{int}}{\text{snd}(y)} - \underset{\text{int}}{1} \right) \right)$$

$\underbrace{\hspace{10em}}_{\tau = \tau * \text{int} \rightarrow \text{int}}$

We conclude $\text{fst}(y) : \tau$ and $\text{snd}(y) : \tau * \text{int} \rightarrow \text{int}$. Thus we have $\tau = \tau * \text{int} \rightarrow \text{int}$ 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:

$$\tau = \tau * \text{int} \rightarrow \text{int}$$

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:

$$\frac{}{n \in C_{int}}$$

$$\frac{t_1 : \tau_1 \quad t_2 : \tau_2 \quad t_1, t_2 \text{ closed}}{(t_1, t_2) \in C_{\tau_1 * \tau_2}}$$

$$\frac{\lambda x.t : \tau_1 \rightarrow \tau_2 \quad \lambda x.t \text{ closed}}{\lambda x.t \in C_{\tau_1 \rightarrow \tau_2}}$$

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:

$$\frac{}{c \rightarrow c} \quad \frac{t_1 \rightarrow n_1 \quad t_2 \rightarrow n_2}{t_1 \text{ op } t_2 \rightarrow n_1 \text{ op } n_2} \quad \frac{t_0 \rightarrow 0 \quad t_1 \rightarrow c_1}{\mathbf{if } t_0 \text{ then } t_1 \text{ else } t_2 \rightarrow c_1} \quad \frac{t_0 \rightarrow n \quad n \neq 0 \quad t_2 \rightarrow c_2}{\mathbf{if } t_0 \text{ then } t_1 \text{ else } t_2 \rightarrow c_2}$$

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 $\mathbf{if } t_0 \text{ then } t_1 \text{ else } t_2$ stand for $\mathbf{if } t_0 = 0 \text{ then } t_1 \text{ else } t_2$, so $t_0 \neq 0$ means t_0 false and $t_0 = 0$ means t_0 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:

$$\frac{t \rightarrow (t_1, t_2) \quad t_1 \rightarrow c_1}{\mathbf{fst}(t) \rightarrow c_1} \quad \frac{t \rightarrow (t_1, t_2) \quad t_2 \rightarrow c_2}{\mathbf{snd}(t) \rightarrow c_2}$$

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.

$$\frac{t_1 \rightarrow \lambda x.t'_1 \quad t'_1[t_2/x] \rightarrow c}{(t_1 \ t_2) \rightarrow c} \quad (\text{lazy})$$

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.

$$\frac{t_1 \rightarrow \lambda x.t'_1 \quad t_2 \rightarrow c_2 \quad t'_1[c_2/x] \rightarrow c}{(t_1 \ t_2) \rightarrow c} \quad (\text{eager})$$

Finally we have a rule for “Rec”:

$$\frac{t[\text{rec } x.t/x] \rightarrow c}{\text{rec } x.t \rightarrow c}$$

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:

$$\mathbf{fact} = \mathbf{rec } f.\lambda x. \mathbf{if } x \mathbf{ then } 1 \mathbf{ else } x \times (f(x - 1))$$

As we said in the previous examples *fact* is closed and typable. So we can calculate its canonical form by using the last rule:

$$\mathbf{fact} \longrightarrow \lambda x. \mathbf{if } x \mathbf{ then } 1 \mathbf{ else } x \times (\mathbf{fact}(x - 1))$$

Now we can apply this function to a specific value and calculate the result as usual:

$$\begin{array}{l} (\mathbf{fact } 2) \rightarrow c \quad \swarrow \\ \swarrow \\ \swarrow_{t=\mathbf{if } x \mathbf{ then } 1 \mathbf{ else } x \times \mathbf{fact}(x-1)} \\ \swarrow_{c=c_1 \times c_2} \\ \swarrow_{c_1=2} \\ \swarrow \\ \swarrow_{n=n_1-n_2} \\ \swarrow_{n_1=2 \quad n_2=1} \\ \swarrow_{c_2=c_3 \times c_4} \\ \swarrow_{c_3=1} \\ \swarrow^* \\ \swarrow_{c_4=1 \quad c_3=1 \quad c_2=c_3 \times c_4=1 \quad c=c_1 \times c_2 \times 1=2} \end{array} \quad \begin{array}{l} \mathbf{fact} \rightarrow \lambda x.t \quad t[2/x] \rightarrow c \\ \lambda x. \mathbf{if } x \mathbf{ then } 1 \mathbf{ else } x \times (\mathbf{fact}(x - 1)) \rightarrow \lambda x.t \quad t[2/x] \rightarrow c \\ \mathbf{if } 2 \mathbf{ then } 1 \mathbf{ else } 2 \times (\mathbf{fact}(2 - 1)) \rightarrow c \\ 2 \times (\mathbf{fact}(2 - 1)) \rightarrow c \\ 2 \rightarrow c_1 \quad (\mathbf{fact}(2 - 1)) \rightarrow c_2 \\ \mathbf{fact} \rightarrow \lambda x.t \quad \underbrace{t[2-1/x] \rightarrow c_2}_{2-1 \text{ not evaluated}} \\ \mathbf{if}(2 - 1) \mathbf{ then } 1 \mathbf{ else}(2 - 1) \times (\mathbf{fact}((2 - 1) - 1)) \rightarrow c_2 \\ 2 - 1 \rightarrow n \quad n \neq 0 \quad (2 - 1) \times \mathbf{fact}((2 - 1) - 1) \rightarrow c_2 \\ 2 \rightarrow n_1 \quad 1 \rightarrow n_2 \quad 2 - 1 \rightarrow n_1 - n_2 \quad n_1 - n_2 \neq 0 \\ (2 - 1) \times \mathbf{fact}((2 - 1) - 1) \rightarrow c_2 \\ 2 - 1 \rightarrow 1 \quad 1 \neq 0 \quad (2 - 1) \times \mathbf{fact}((2 - 1) - 1) \rightarrow c_2 \\ 2 - 1 \rightarrow c_3 \quad \mathbf{fact}((2 - 1) - 1) \rightarrow c_4 \\ \mathbf{if}(2 - 1) - 1 \mathbf{ then } 1 \mathbf{ else}((2 - 1) - 1) \times (\mathbf{fact}(((2 - 1) - 1) - 1)) \rightarrow c_4 \\ (2 - 1) - 1 \rightarrow 0 \quad 1 \rightarrow c_4 \\ \square \end{array}$$

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:

$$\begin{array}{ccc}
 ((\lambda x : \text{int}.3) \text{rec } y : \text{int}.y) : \text{int} \rightarrow c & \searrow & \lambda x : \text{int}.3 \rightarrow \lambda x.t \quad t[\text{rec } y.y/x] \rightarrow c \\
 & \searrow_{t=3} & 3[\text{rec } y.y/x] \rightarrow c \\
 & \searrow_{c=3} & \square
 \end{array}$$

So although the term $\text{rec } y.y$ 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:

$$\begin{array}{ccc}
 ((\lambda x : \text{int}.3) \text{rec } y : \text{int}.y) : \text{int} \rightarrow c & \searrow & \lambda x : \text{int}.3 \rightarrow \lambda x.t \quad \text{rec } y.y \rightarrow c_1 \quad t[c_1/x] \rightarrow c \\
 & \searrow_{t=3} & \text{rec } y.y \rightarrow c_1 \quad 3[c_1/x] \rightarrow c \\
 & \searrow_{y=\text{rec } y.y} & \text{rec } y.y \rightarrow c_1 \quad 3[c_1/x] \rightarrow c \\
 & \searrow_{y=\text{rec } y.y} & \dots
 \end{array}$$

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

- If $t \rightarrow c$ and $t \rightarrow c'$ then $c = c'$ (if a canonical form exists it is unique)
- if $t \rightarrow c$ and $t : \tau$ then $c : \tau$ (the evaluation relation respects the types)

Proof. We prove the property i) by rule induction. Let us show only the function rule, the remainder is left as exercise. We have the rule:

$$\frac{t_1 \rightarrow \lambda x.t'_1 \quad t'_1[t^2/x] \rightarrow c}{(t_1 \ t_2) \rightarrow c}$$

We will show:

$$P((t_1 \ t_2) \rightarrow c) \stackrel{\text{def}}{=} \forall c' (t_1 \ t_2) \rightarrow c' \Rightarrow c = c'$$

As usually let us assume the premise:

$$(t_1 \ t_2) \rightarrow c'$$

And the inductive hypothesis:

- $P(t_1 \rightarrow \lambda x.t'_1) \stackrel{\text{def}}{=} \forall c' t_1 \rightarrow c' \Rightarrow \lambda x.t'_1 = c'$
- $P(t'_1[t^2/x] \rightarrow c) \stackrel{\text{def}}{=} \forall c' t'_1[t^2/x] \rightarrow c' \Rightarrow c = c'$

From $(t_1 \ t_2) \rightarrow c'$ by goal reduction:

- $t_1 \rightarrow \lambda \bar{x}.\bar{t}'_1$
- $\bar{t}'_1[\bar{t}^2/\bar{x}] \rightarrow c'$

then we have by inductive hypothesis:

- $\lambda x.t'_1 = \lambda \bar{x}.\bar{t}'_1$
- $t'_1[t^2/x] = \bar{t}'_1[\bar{t}^2/\bar{x}]$

So we have $c = c'$ □

7. Domain Theory

As done for IMP we would like to introduce the denotational semantics of HOFL, for which we need to develop a proper domain theory.

In order to define the denotational semantics of IMP we have shown that the semantic domain of commands, for which we need to apply fixpoint theorem, has the required properties. The situation is more complicated for HOFL, because HOFL provides constructors for infinitely many term types, so there are infinitely many domains to be considered. We will handle this problem by showing by structural induction that the type constructors of HOFL correspond to domains which are equipped with adequate CPO_{\perp} structures.

7.1. The Domain \mathbb{N}_{\perp}

We define the $CPO_{\perp} \mathbb{N}_{\perp} = (\mathbb{N} \cup \{\perp_{\mathbb{N}_{\perp}}\}, \sqsubseteq)$ as follows:

- \mathbb{N} is the set of integer numbers
- $\forall x \in \mathbb{N} \cup \{\perp_{\mathbb{N}_{\perp}}\}. \perp_{\mathbb{N}_{\perp}} \sqsubseteq x$ and $x \sqsubseteq x$

obviously \mathbb{N}_{\perp} is a CPO with bottom, indeed $\perp_{\mathbb{N}_{\perp}}$ is the bottom element and each chain has a LUB (note that chains are all of length 1 or 2).

7.2. Cartesian Product of Two Domains

We start with two CPO_{\perp} :

$$\begin{aligned} \mathcal{D} &= (D, \sqsubseteq_D) \\ \mathcal{E} &= (E, \sqsubseteq_E) \end{aligned}$$

Now we construct the Cartesian product $\mathcal{D} \times \mathcal{E} = (D \times E, \sqsubseteq_{D \times E})$ which has as elements the pairs of elements of D and E . Let us define the order as follows:

- $\forall d_1, d_2 \in D \forall e_1, e_2 \in E. (d_1, e_1) \sqsubseteq_{D \times E} (d_2, e_2) \Leftrightarrow d_1 \sqsubseteq_D d_2 \wedge e_1 \sqsubseteq_E e_2$

Let us show that $\sqsubseteq_{D \times E}$ is a partial order:

- reflexivity: since \sqsubseteq_D and \sqsubseteq_E are reflexive we have $\forall e \in E e \sqsubseteq_E e$ and $\forall d \in D d \sqsubseteq_D d$ so by definition of $\sqsubseteq_{D \times E}$ we have $\forall d \in D \forall e \in E. (d, e) \sqsubseteq_{D \times E} (d, e)$.
- antisymmetry: let us assume $(d, e) \sqsubseteq_{D \times E} (d_1, e_1)$ and $(d_1, e_1) \sqsubseteq_{D \times E} (d, e)$ so by definition of $\sqsubseteq_{D \times E}$ we have $d \sqsubseteq_D d_1$ (using the first relation) and $d_1 \sqsubseteq_D d$ (by using the second relation) so it must be $d = d_1$ and similarly $e = e_1$, hence $(d, e) = (d_1, e_1)$.
- transitivity: let us assume $(d, e) \sqsubseteq_{D \times E} (d_1, e_1)$ and $(d_1, e_1) \sqsubseteq_{D \times E} (d_2, e_2)$. By definition of $\sqsubseteq_{D \times E}$ we have $d \sqsubseteq_D d_1$, $d_1 \sqsubseteq_D d_2$, $e \sqsubseteq_E e_1$ and $e_1 \sqsubseteq_E e_2$. By transitivity of \sqsubseteq_D and \sqsubseteq_E we have $d \sqsubseteq_D d_2$ and $e \sqsubseteq_E e_2$. By definition of $\sqsubseteq_{D \times E}$ we obtain $(d, e) \sqsubseteq_{D \times E} (d_2, e_2)$.

Now we show that the PO has a bottom element $\perp_{D \times E} = (\perp_D, \perp_E)$. In fact $\forall d \in D, e \in E. \perp_D \sqsubseteq_D d \wedge \perp_E \sqsubseteq_E e$, thus $(\perp_D, \perp_E) \sqsubseteq_{D \times E} (d, e)$. It remains to show the completeness of $\mathcal{D} \times \mathcal{E}$.

Theorem 7.1 (Completeness of $\mathcal{D} \times \mathcal{E}$)

The PO $\mathcal{D} \times \mathcal{E}$ defined above is complete.

Proof. We will prove that for each chain $(d_i, e_i)_{i \in \omega}$ it holds:

$$\bigsqcup_{i \in \omega} (d_i, e_i) = \left(\bigsqcup_{i \in \omega} d_i, \bigsqcup_{i \in \omega} e_i \right)$$

Obviously $(\bigsqcup_{i \in \omega} d_i, \bigsqcup_{i \in \omega} e_i)$ is an upper bound, indeed for each $j \in \omega$ we have $d_j \sqsubseteq_D \bigsqcup_{i \in \omega} d_i$ and $e_j \sqsubseteq_E \bigsqcup_{i \in \omega} e_i$ so by definition of $\sqsubseteq_{D \times E}$ it holds $(d_j, e_j) \sqsubseteq_{D \times E} (\bigsqcup_{i \in \omega} d_i, \bigsqcup_{i \in \omega} e_i)$.

Moreover $(\bigsqcup_{i \in \omega} d_i, \bigsqcup_{i \in \omega} e_i)$ is also the least upper bound. Indeed, let (\bar{d}, \bar{e}) be an upper bound of $\{(d_i, e_i)\}_{i \in \omega}$, since $\bigsqcup_{i \in \omega} d_i$ is the LUB of $\{d_i\}_{i \in \omega}$ we have $\bigsqcup_{i \in \omega} d_i \sqsubseteq_D \bar{d}$, furthermore we have that $\bigsqcup_{i \in \omega} e_i$ is the LUB of $\{e_i\}_{i \in \omega}$ then $\bigsqcup_{i \in \omega} e_i \sqsubseteq_E \bar{e}$. So by definition of $\sqsubseteq_{D \times E}$ we have $(\bigsqcup_{i \in \omega} d_i, \bigsqcup_{i \in \omega} e_i) \sqsubseteq_{D \times E} (\bar{d}, \bar{e})$. Thus $(\bigsqcup_{i \in \omega} d_i, \bigsqcup_{i \in \omega} e_i)$ is the least upper bound. \square

Let us define the projection operators of $\mathcal{D} \times \mathcal{E}$.

Definition 7.2 (Projection operators π_1 and π_2)

Let $(d, e) \in D \times E$ be a pair, we define the left and right projection functions $\pi_1 : D \times E \rightarrow D$ and $\pi_2 : D \times E \rightarrow E$ as follows.

- $\pi_1((d, e)) = d$
- $\pi_2((d, e)) = e$

Recall that in order to use a function in domain theory we have to show that it is continuous, this ensures that the function respects the domain structure (i.e. the function does not change the order and preserves limits) and so we can calculate its fixpoints.

So we have to prove that each function which we use on $\mathcal{D} \times \mathcal{E}$ is continuous.

Theorem 7.3 (Continuity of π_1 and π_2)

Let π_1 and π_2 be the functions of the previous definition then:

$$\pi_1 \left(\bigsqcup_i (d_i, e_i) \right) = \bigsqcup_i \pi_1((d_i, e_i))$$

and

$$\pi_2 \left(\bigsqcup_i (d_i, e_i) \right) = \bigsqcup_i \pi_2((d_i, e_i))$$

Proof. Let us prove the first statement:

$$\pi_1 \left(\bigsqcup_i (d_i, e_i) \right) = \pi_1 \left(\left(\bigsqcup_i d_i, \bigsqcup_i e_i \right) \right) = \bigsqcup_i d_i = \bigsqcup_i \pi_1((d_i, e_i)).$$

For π_2 the proof is analogous. \square

7.3. Functional Domains

As for Cartesian product, we start from two domains and we define the order on the set $[D \rightarrow E]$ of the continuous functions in $\{f \mid f : D \rightarrow E\}$. Note that as usual we require the continuity of the functions to preserve the applicability of fixpoint theory.

Let us consider the CPOs:

$$\begin{aligned}\mathcal{D} &= (D, \sqsubseteq_D) \\ \mathcal{E} &= (E, \sqsubseteq_E)\end{aligned}$$

Now we define an order on $[D \rightarrow E]$.

$$[\mathcal{D} \rightarrow \mathcal{E}] = ([D \rightarrow E], \sqsubseteq_{[D \rightarrow E]})$$

where:

- $[D \rightarrow E] = \{f \mid f : D \rightarrow E, f \text{ is continuous} \}$
- $f \sqsubseteq_{[D \rightarrow E]} g \Leftrightarrow \forall d \in D. f(d) \sqsubseteq_E g(d)$

We leave as an exercise the proof that $[\mathcal{D} \rightarrow \mathcal{E}]$ is a PO with bottom, namely the order is reflexive, antisymmetric, transitive and that $\perp_{[D \rightarrow E]}(d) = \perp_E$ is a continuous function.

Let us show that the PO is complete. In order to simplify the completeness proof we introduce the following lemmas:

Lemma 7.4 (Switch Lemma)

Let (E, \sqsubseteq_E) be a CPO whose elements are of the form $e_{n,m}$ with $n, m \in \omega$. If \sqsubseteq_E is such that:

$$e_{n,m} \sqsubseteq_E e_{n',m'} \text{ if } n \leq n' \text{ and } m \leq m'$$

then it holds:

$$\bigsqcup_{n,m \in \omega} e_{n,m} = \bigsqcup_{n \in \omega} (\bigsqcup_{m \in \omega} e_{n,m}) = \bigsqcup_{m \in \omega} (\bigsqcup_{n \in \omega} e_{n,m}) = \bigsqcup_k e_{k,k}$$

Proof. Our order can be summarized as follows:

$$\begin{array}{ccccccc} e_{00} & \sqsubseteq & e_{01} & \sqsubseteq & e_{02} & \sqsubseteq & \cdots & \bigsqcup_{i \in \omega} e_{0i} = e_0 \\ e_{10} & \sqsubseteq & e_{11} & \sqsubseteq & e_{12} & \sqsubseteq & \cdots & \bigsqcup_{i \in \omega} e_{1i} = e_1 \\ e_{20} & \sqsubseteq & e_{21} & \sqsubseteq & e_{22} & \sqsubseteq & \cdots & \bigsqcup_{i \in \omega} e_{2i} = e_2 \\ \vdots & & \vdots & & \vdots & & \ddots & \vdots \end{array}$$

We show that all the following sets have the same upper bounds:

$$\{e_{n,m}\}_{n,m \in \omega} \quad \{\bigsqcup_{m \in \omega} e_{n,m}\}_{n \in \omega} \quad \{\bigsqcup_{n \in \omega} e_{n,m}\}_{m \in \omega} \quad \{e_{k,k}\}_{k \in \omega}$$

Let e be an upper bound of $\{\bigsqcup_{m \in \omega} e_{n,m}\}_{n \in \omega}$ and take any $e_{n',m'}$ for some n', m' . Then

$$e_{n',m'} \sqsubseteq \bigsqcup_{m \in \omega} e_{n',m} \sqsubseteq e$$

Thus e is an upper bound for $\{e_{n,m}\}_{n,m \in \omega}$.

Vice versa, let e be an upper bound of $\{e_{n,m}\}_{n,m \in \omega}$ and consider $\bigsqcup_{m \in \omega} e_{n',m}$ for some n' . Since $\{e_{n',m}\}_{m \in \omega} \subseteq \{e_{n,m}\}_{n,m \in \omega}$, obviously e is an upper bound for $\{e_{n',m}\}_{m \in \omega}$ and therefore $\bigsqcup_{m \in \omega} e_{n',m} \sqsubseteq e$.

Now each element $e_{n,m}$ is smaller than $e_{k,k}$ with $k = \max\{n, m\}$ thus an upper bound of $\{e_{k,k}\}_{k \in \omega}$ is also an upper bound of $\{e_{n,m}\}_{n,m \in \omega}$. Moreover $\{e_{k,k}\}_{k \in \omega}$ is a subset of $\{e_{n,m}\}_{n,m \in \omega}$ so an upper bounds of $\{e_{n,m}\}_{n,m \in \omega}$ is also an upper bound of $\{e_{k,k}\}_{k \in \omega}$. The set of upper bounds $\{\sqcup_{m \in \omega} e_{n,m}\}_{n \in \omega}$ has a least element. In fact, $n_1 \leq n_2$ implies $\sqcup_{m \in \omega} e_{n_1,m} \sqsubseteq \sqcup_{m \in \omega} e_{n_2,m}$, since every upper bound of $\sqcup_{m \in \omega} e_{n_2,m}$ is an upper bound of $\sqcup_{m \in \omega} e_{n_1,m}$. Therefore $\{\sqcup_{m \in \omega} e_{n,m}\}_{n \in \omega}$ is a chain, and thus it has a LUB since E is a CPO. \square

Lemma 7.5

Let $\{f_n\}_{n \in \omega}$ be a chain of functions (not necessarily continuous) in $\mathcal{D} \rightarrow \mathcal{E}$ the LUB $\sqcup_{n \in \omega} f_n$ exists and is defined as:

$$\left(\sqcup_{n \in \omega} f_n \right) (d) = \sqcup_{n \in \omega} (f_n(d))$$

Proof. Function $\lambda d. \sqcup_{n \in \omega} (f_n(d))$ is clearly an upper bound for $\{f_n\}_{n \in \omega}$ since for every k and d we have $f_k(d) \sqsubseteq_E \sqcup_{n \in \omega} f_n(d)$. Function $\lambda d. \sqcup_{n \in \omega} (f_n(d))$ is also the LUB of $\{f_n\}_{n \in \omega}$ since taken g such that $f_n \sqsubseteq_{\mathcal{D} \rightarrow \mathcal{E}} g$ for any n , we have for any d that $f_n(d) \sqsubseteq_E g(d)$ and therefore $\sqcup_{n \in \omega} (f_n(d)) \sqsubseteq_E g(d)$. \square

Lemma 7.6

Let $\{f_n\}_{n \in \omega}$ be a chain of functions in $[\mathcal{D} \rightarrow \mathcal{E}]$ and let $\{d_n\}_{n \in \omega}$ be a chain on \mathcal{D} then function $\lambda d. \sqcup_{n \in \omega} (f_n(d))$ is continuous, namely

$$\sqcup_{n \in \omega} \left(f_n \left(\sqcup_{m \in \omega} d_m \right) \right) = \sqcup_{m \in \omega} \left(\sqcup_{n \in \omega} f_n(d_m) \right)$$

Furthermore, $\lambda d. \sqcup_{n \in \omega} (f_n(d))$ is the LUB of $\{f_n\}_{n \in \omega}$ not only in $\mathcal{D} \rightarrow \mathcal{E}$ as stated by lemma 7.5, but also in $[\mathcal{D} \rightarrow \mathcal{E}]$.

Proof.

$$\begin{aligned} \sqcup_n (f_n (\sqcup_m d_m)) &= && \text{by continuity} \\ \sqcup_n (\sqcup_m (f_n(d_m))) &= && \text{by lemma 7.4 (switch lemma)} \\ \sqcup_m (\sqcup_n (f_n(d_m))) & & & \end{aligned}$$

The upper bounds of $\{f_n\}_{n \in \omega}$ in $\mathcal{D} \rightarrow \mathcal{E}$ are a larger set than those in $[\mathcal{D} \rightarrow \mathcal{E}]$, thus if $\lambda d. \sqcup_{n \in \omega} (f_n(d))$ is the LUB in $\mathcal{D} \rightarrow \mathcal{E}$, it is also the LUB in $[\mathcal{D} \rightarrow \mathcal{E}]$. \square

Theorem 7.7 (Completeness of the functional space)

The PO $[\mathcal{D} \rightarrow \mathcal{E}]$ is a CPO_{\perp}

Proof. The statement follows immediately from the previous Lemmas. \square

7.4. Lifting

In IMP we introduced a lifting operator (Chapter 5.2.3) on memories Σ to obtain a CPO Σ_{\perp} . In the semantics of HOFL we need the same operator in a more general fashion: we need to apply the operator to any domain.

Definition 7.8

Let $\mathcal{D} = (D, \sqsubseteq_D)$ be a CPO and let \perp be an element not in D , so we define the lifted CPO \mathcal{D}_{\perp} as follows:

- $D_{\perp} = \{(0, \perp)\} \cup \{1\} \times D$
- $\perp_{D_{\perp}} = (0, \perp)$
- $d_1 \sqsubseteq_D d_2 \Rightarrow (1, d_1) \sqsubseteq_{D_{\perp}} (1, d_2)$
- $\perp_{D_{\perp}} \sqsubseteq_{D_{\perp}} \perp_{D_{\perp}}$ and $\forall d \in D \perp_{D_{\perp}} \sqsubseteq_{D_{\perp}} (1, d)$

Now we define a lifting function $\lfloor - \rfloor : D \rightarrow D_{\perp}$ as follows:

- $\lfloor d \rfloor = (1, d) \forall d \in D$

As it was the case for Σ in the IMP semantics, when we add a bottom element to a domain \mathcal{D} we would like to extend the continuous functions in $[D \rightarrow E]$ to continuous functions in $[D_{\perp} \rightarrow E]$. The function defining the extension should itself be continuous.

Definition 7.9

Let \mathcal{D} be a CPO and let \mathcal{E} be a CPO $_{\perp}$. We define a lifting operator $_{\perp}^* : [D \rightarrow E] \rightarrow [D_{\perp} \rightarrow E]$ for functions in $[D \rightarrow E]$ as follows:

$$\forall f \in [D \rightarrow E] \quad f^*(x) = \begin{cases} \perp_E & \text{if } x = \perp_{D_{\perp}} \\ f(d) & \text{if } x = \lfloor d \rfloor \end{cases}$$

Theorem 7.10

- i) If f is continuous in $[D \rightarrow E]$, then f^* is continuous in $[D_{\perp} \rightarrow E]$.
- ii) The operator $_{\perp}^*$ is continuous.

Proof.

- i) Let $\{d_i\}_{i \in \omega}$ be a chain in \mathcal{D}_{\perp} . We have to prove $f^*(\bigsqcup_{n \in \omega} d_n) = \bigsqcup_{n \in \omega} f^*(d_n)$. If $\forall n. d_n = \perp_{D_{\perp}}$, then this is obvious. Otherwise from some k and for all $m \geq k$ we have $d_m = \lfloor d'_m \rfloor$ and also $\bigsqcup_{n \in \omega} d_n = \lfloor \bigsqcup_{n \in \omega} d'_{n+k} \rfloor$. Then $f^*(\bigsqcup_{n \in \omega} d_n) = f^*(\lfloor \bigsqcup_{n \in \omega} d'_{n+k} \rfloor) = f(\bigsqcup_{n \in \omega} d'_{n+k}) = \bigsqcup_{n \in \omega} f(d'_{n+k}) = \bigsqcup_{n \in \omega} f^*(d_{n+k}) = \bigsqcup_{n \in \omega} f^*(d_n)$.
- ii) Let $\{f_i\}_{i \in \omega}$ be a chain of functions in $[\mathcal{D} \rightarrow \mathcal{E}]$. We will prove:

$$\left(\bigsqcup_{i \in \omega} f_i \right)^*(x) = \left(\bigsqcup_{i \in \omega} f_i^* \right)(x)$$

if $x = \perp_{D_{\perp}}$ both sides of the equation simplify to \perp_E . So let us consider $x = \lfloor d \rfloor$ for some $d \in D$ we have:

$$\begin{aligned} \left(\bigsqcup_{i \in \omega} f_i \right)^*(\lfloor d \rfloor) &= && \text{by definition of lifting} \\ \left(\bigsqcup_{i \in \omega} f_i \right)(d) &= && \text{by definition of LUB} \\ \bigsqcup_{i \in \omega} (f_i(d)) &= && \text{by definition of lifting} \\ \bigsqcup_{i \in \omega} (f_i^*(\lfloor d \rfloor)) &= && \text{by definition of LUB} \\ \left(\bigsqcup_{i \in \omega} f_i^* \right)(\lfloor d \rfloor) &= && \end{aligned}$$

□

7.5. Function's Continuity Theorems

In this section we show some theorems which allow to prove the continuity of the functions which we will define over our CPOs. We start proving that the composition of two continuous functions is continuous.

Theorem 7.11

Let $f : [D \rightarrow E]$ and $g : [E \rightarrow F]$ be two continuous functions on CPO_{\perp} 's. The composition

$$f; g = g \circ f = \lambda d. g(f(d)) : D \rightarrow F$$

is a continuous function, namely

$$g(f(\bigsqcup_{n \in \omega} d_n)) = \bigsqcup_{n \in \omega} g(f(d_n))$$

Proof. Immediate:

$$\begin{aligned} g(f(\bigsqcup_{n \in \omega} d_n)) &= && \text{by the continuity of } f \\ g(\bigsqcup_{n \in \omega} f(d_n)) &= && \text{by the continuity of } g \\ \bigsqcup_{n \in \omega} g(f(d_n)) & & & \end{aligned}$$

□

Now we consider a function whose outcome is a pair of values. So the function has as domain a CPO but the result is on a product of CPOs.

$$f : S \rightarrow D \times E$$

For this type of functions we introduce a theorem which allows to prove the continuity of f in a convenient way. We will consider f as the pairing of two simpler functions $g_1 : S \rightarrow D$ and $g_2 : S \rightarrow E$, then we can prove the continuity of f from the continuity of g_1 and g_2 .

Theorem 7.12

Let $f : S \rightarrow D \times E$ be a function over CPOs and let $g_1 : S \rightarrow D$ and $g_2 : S \rightarrow E$ be two functions defined as follows:

- $g_1 = f; \pi_1$
- $g_2 = f; \pi_2$

where $f; \pi_1 = \lambda s. \pi_1(f(s))$ is the composition of f and π_1 . Notice that we have

$$f = \lambda s. (g_1(s), g_2(s)) \stackrel{\text{def}}{=} (g_1, g_2)$$

Then we have:

f is continuous $\iff g_1$ and g_2 are continuous.

Proof.

\Leftrightarrow) By definition $f = (g_1, g_2)$. We assume the continuity of g_1 and g_2 . We prove:

$$(g_1, g_2)(\bigsqcup_{i \in \omega} s_i) = \bigsqcup_{i \in \omega} ((g_1, g_2)(s_i))$$

So we have:

$$\begin{aligned} (g_1, g_2)(\bigsqcup_{i \in \omega} s_i) &= && \text{by definition} \\ (g_1(\bigsqcup_{i \in \omega} s_i), g_2(\bigsqcup_{i \in \omega} s_i)) &= && \text{by continuity of } g_1 \text{ and } g_2 \\ (\bigsqcup_{i \in \omega} g_1(s_i), \bigsqcup_{i \in \omega} g_2(s_i)) &= && \text{definition of LUB of pairs} \\ \bigsqcup_{i \in \omega} (g_1(s_i), g_2(s_i)) &= && \text{definition of pairs of functions} \\ \bigsqcup_{i \in \omega} ((g_1, g_2)(s_i)) &= && \end{aligned}$$

\Rightarrow) Immediate by Theorem 7.11 (continuity of composition) and Theorem 7.3 (continuity of projections), since g_1 and g_2 are compositions of continuous functions. □

Note that in our construction we defined only ordered pairs of elements, this means that if we want to consider sequences (i.e. with finitely many elements) we have to use the pairing repeatedly. So for example (a, b, c) is defined as $((a, b), c)$.

Now let us consider the case of a function $f : D_1 \times D_2 \rightarrow E$ over CPOs which takes two arguments and returns an element of E . The following theorem allows us to study the continuity of f by analysing each parameter separately.

Theorem 7.13

Let $f : D_1 \times D_2 \rightarrow E$ be a function over CPOs. Then f is continuous iff all the functions in the following two classes are continuous.

$$\forall d_1. f_{d_1} : D_2 \rightarrow E \text{ defined as } f_{d_1} = \lambda d. f(d_1, d)$$

$$\forall d_2. f_{d_2} : D_1 \rightarrow E \text{ defined as } f_{d_2} = \lambda d. f(d, d_2)$$

Proof.

\Rightarrow) If f is continuous then $\forall d_1, d_2. f_{d_1}$ and f_{d_2} are continuous, since we are considering only certain chains (i.e. we are fixing one element of the pair).

\Leftarrow) On the other hand we have:

$$\begin{aligned} f(\bigsqcup_{i \in \omega} (x_i, y_i)) &= && \text{by definition of LUB on pairs} \\ f(\bigsqcup_{i \in \omega} x_i, \bigsqcup_{j \in \omega} y_j) &= && \text{by continuity of } f_{\bigsqcup_{j \in \omega} y_j} \\ \bigsqcup_{i \in \omega} f(x_i, \bigsqcup_{j \in \omega} y_j) &= && \text{by continuity of } f_{x_i} \\ \bigsqcup_{i \in \omega} \bigsqcup_{j \in \omega} f(x_i, y_j) &= && \text{by Lemma 7.4 (switch lemma)} \\ \bigsqcup_{i \in \omega} f(x_i, y_i) &= && \end{aligned}$$

□

7.6. Useful Functions

As done for IMP we will use the λ -notation as meta-language for the denotational semantics of HOFL. In previous sections we already defined two new functions for our meta-language: π_1 and π_2 . We also showed that π_1 and π_2 are continuous. In this section we introduce some functions which, together with $Cond$, π_1 and π_2 will form the kernel of our meta-language.

Definition 7.14 (Apply)

We define a function $\text{apply} : [D \rightarrow E] \times D \rightarrow E$ over domains as follows:

$$\text{apply}(f, d) \stackrel{\text{def}}{=} f(d)$$

The apply function represents the application of a function in our meta-language. In order to use the apply we prove that it is continuous.

Theorem 7.15 (Continuity of apply)

Let $\text{apply} : [D \rightarrow E] \times D \rightarrow E$ be the function defined above and let $\{f_i, d_i\}_{i \in \omega}$ be a chain on the CPO_{\perp} $[D \rightarrow E] \times D$ then:

$$\text{apply}\left(\bigsqcup_{i \in \omega} (f_i, d_i)\right) = \bigsqcup_{i \in \omega} \text{apply}(f_i, d_i)$$

Proof. By using the Theorem 7.13 we can test the continuity on each parameter separately. Let us fix $d \in D$, we have:

$$\begin{aligned} \text{apply}(\bigsqcup_n f_n, d) &= && \text{by definition} \\ (\bigsqcup_n f_n)(d) &= && \text{by definition of LUB of functions} \\ \bigsqcup_n (f_n(d)) &= && \text{by definition} \\ \bigsqcup_n \text{apply}(f_n, d) & & & \end{aligned}$$

Now we fix $f \in [D \rightarrow E]$:

$$\begin{aligned} \text{apply}(f, \bigsqcup_m d_m) &= && \text{by definition} \\ f(\bigsqcup_m d_m) &= && \text{by continuity of } f \\ \bigsqcup_m (f(d_m)) &= && \text{by definition} \\ \bigsqcup_m \text{apply}(f, d_m) & & & \end{aligned}$$

So apply is a continuous function. □

Definition 7.16 (Curry and un-curry)

We define the function $\text{curry} : (D \times F \rightarrow E) \rightarrow (D \rightarrow (F \rightarrow E))$ as:

$$\text{curry } g \ d \ f \stackrel{\text{def}}{=} g(d, f) \text{ where } g : D \times F \rightarrow E, d : D \text{ and } f : F$$

And we define the $\text{un-curry} : (D \rightarrow F \rightarrow E) \rightarrow (D \times F \rightarrow E)$ as:

$$\text{un-curry } g \ (d, f) \stackrel{\text{def}}{=} g \ d \ f \text{ where } g : D \rightarrow F \rightarrow E, d : D \text{ and } f : F$$

Theorem 7.17 (Continuity of the curry of a function)

Let $\text{curry} : (D \times F \rightarrow E) \rightarrow (D \rightarrow (F \rightarrow E))$ be the function defined above let $\{d_i\}_{i \in \omega}$ be a chain on D and let $g : D \times F \rightarrow E$ a continuous function then $(\text{curry } g)$ is a continuous function, namely

$$(\text{curry } g)\left(\bigsqcup_{i \in \omega} d_i\right) = \bigsqcup_{i \in \omega} (\text{curry } g)(d_i)$$

Proof. Let us note that since g is continuous, by Theorem 7.13 g is continuous separately on each parameter. So $(\text{curry } g)(\bigsqcup_{i \in \omega} d_i)$ is also continuous. Then let us take $f \in F$ we have:

$$\begin{aligned} (\text{curry } g)(\bigsqcup_{i \in \omega} d_i)(f) &= && \text{by definition of curry } g \\ g(\bigsqcup_{i \in \omega} d_i, f) &= && \text{by continuity of } g \\ \bigsqcup_{i \in \omega} g(d_i, f) &= && \text{by definition of curry } g \\ \bigsqcup_{i \in \omega} ((\text{curry } g)(d_i)(f)) & & & \end{aligned}$$

□

As shown in Chapter 4 in order to define the denotational semantics of recursive definition we provide a fixpoint operator. So it seems quite natural to introduce the fixpoint in our meta-theory.

Definition 7.18 (Fix)

Let D be a CPO_{\perp} . We define the function $\text{fix} : ([D \rightarrow D] \rightarrow D)$ as:

$$\text{fix} \stackrel{\text{def}}{=} \bigsqcup_{i \in \omega} \lambda f. f^i(\perp_D)$$

Notice that the LUB does exist since $\{\lambda f. f^i(\perp_D)\}$ is a chain of functions and $([D \rightarrow D] \rightarrow D)$ is complete.

Theorem 7.19 (Continuity of fix)

Function $\text{fix} : ([D \rightarrow D] \rightarrow D)$ is continuous, namely $\text{fix} : [[D \rightarrow D] \rightarrow D]$.

Proof. We know that $([D \rightarrow D] \rightarrow D)$ is complete, thus if for all i $\lambda f. f^i(\perp_D)$ is continuous, then $\bigsqcup_{i \in \omega} \lambda f. f^i(\perp_D) = \text{fix}$ is also continuous. We prove that $\forall i. \lambda f. f^i(\perp_D)$ is continuous by mathematical induction.

$i=0$) $\lambda f. f^0(\perp_D)$ is a constant function and thus it is continuous.

$i=n+1$) Let us assume that $\lambda f. f^n(\perp_D)$ is continuous, namely that $(\bigsqcup_{i \in \omega} f_i)^n(\perp_D) = \bigsqcup_{i \in \omega} f_i^n(\perp_D)$, and let us prove that $(\bigsqcup_{i \in \omega} f_i)^{n+1}(\perp_D) = \bigsqcup_{i \in \omega} f_i^{n+1}(\perp_D)$. In fact we have:

$$\begin{aligned} (\bigsqcup_{i \in \omega} f_i)^{n+1}(\perp_D) &= (\bigsqcup_{i \in \omega} f_i)((\bigsqcup_{i \in \omega} f_i)^n(\perp_D)) = && \text{by the inductive hypothesis} \\ (\bigsqcup_{i \in \omega} f_i)(\bigsqcup_{i \in \omega} f_i^n(\perp_D)) &= && \text{by the definition of LUB of functions} \\ \bigsqcup_{i \in \omega} f_i(\bigsqcup_{j \in \omega} f_j^n(\perp_D)) &= && \text{by continuity of } f_i \\ \bigsqcup_{i \in \omega} \bigsqcup_{j \in \omega} f_i(f_j^n(\perp_D)) &= && \text{by Lemma 7.4 (switch lemma)} \\ \bigsqcup_{k \in \omega} f_k(f_k^n(\perp_D)) &= \bigsqcup_{k \in \omega} f_k^{n+1}(\perp_D) \end{aligned}$$

□

Finally we introduce the “let” operator, whose role is that of binding a name to a de-lifted expression. Note that the continuity of the “let” operator directly follows from the continuity of the lifting operator.

Definition 7.20 (Let operator)

Let \mathcal{E} be a CPO_{\perp} and $\lambda x.e$ a function in $[D \rightarrow E]$. We define the let operator as follows:

$$\text{let } x \leftarrow d_{\perp}. e \stackrel{\text{def}}{=} \underbrace{\underbrace{(\lambda x. e)^*}_{D \rightarrow E} \underbrace{(d_{\perp})}_{D_{\perp}}}_{D_{\perp} \rightarrow E}}_E = \begin{cases} \perp_E & \text{if } d_{\perp} = \perp_{D_{\perp}} \\ e[d/x] & \text{if } d_{\perp} = [d] \end{cases}$$

8. HOFL Denotational Semantics

In order to define the denotational semantics of a computer language we have to define by structural recursion an evaluation function from each syntactic domain to a semantic domain.

Since HOFL has only one syntactic domain (i.e. the set of typed terms t) we have only one evaluation function $\llbracket t \rrbracket$. However, since the terms are typed, the evaluation function is parametrised by the types. We have

$$\llbracket t : \tau \rrbracket : Env \longrightarrow (V_\tau)_\perp$$

Here $\rho \in Env$ are environments, which contain the values to be assigned to variables, in practice the free variables of t

$$\rho : Env = Var \longrightarrow \bigcup_{\tau} (V_\tau)_\perp$$

with the condition $\rho(x : \tau) \in (V_\tau)_\perp$.

In our denotational semantics of HOFL we distinguish between V_τ , where we find the meanings of the terms of type τ with canonical forms, and $(V_\tau)_\perp$, where the additional $\perp_{(V_\tau)_\perp}$ is the meaning of all the terms of type τ without a canonical form.

The actual semantic domains V_τ (and $(V_\tau)_\perp$) are defined by structural recursion on the syntax of types:

$$\begin{array}{ll} V_{int} = \mathbb{N} & (V_{int})_\perp = \mathbb{N}_\perp \\ V_{\tau_1 * \tau_2} = (V_{\tau_1})_\perp \times (V_{\tau_2})_\perp & (V_{\tau_1 * \tau_2})_\perp = ((V_{\tau_1})_\perp \times (V_{\tau_2})_\perp)_\perp \\ V_{\tau_1 \rightarrow \tau_2} = [(V_{\tau_1})_\perp \rightarrow (V_{\tau_2})_\perp] & (V_{\tau_1 \rightarrow \tau_2})_\perp = [(V_{\tau_1})_\perp \rightarrow (V_{\tau_2})_\perp]_\perp \end{array}$$

Notice that the recursive definition takes advantage of the domain constructors we defined in Chapter 7. While the lifting \mathbb{N}_\perp of the integer numbers \mathbb{N} is strictly necessary, liftings on cartesian pairs and on continuous functions are actually optional, since cartesian products and functional domains are already CPO_\perp . We will discuss the motivation of our choice at the end of the following chapter.

8.1. HOFL Evaluation Function

Now we are ready to define the evaluation function, by structural recursion. As usual we start from the constant terms, then we define compositionally the more complicated ones.

8.1.1. Constants

We define the meaning of a constant as the obvious value on the lifted domains:

$$\llbracket n \rrbracket \rho = \lfloor n \rfloor$$

8.1.2. Variables

The meaning of a variable is defined by its value in the given environment:

$$\llbracket x \rrbracket \rho = \rho x$$

8.1.3. Binary Operators

We give the generic semantics of a binary operator as:

$$\llbracket t_1 \text{ op } t_2 \rrbracket \rho = \llbracket t_1 \rrbracket \rho \underline{op}_{\perp} \llbracket t_2 \rrbracket \rho$$

where

$$\underline{op}_{\perp} : (\mathbb{N}_{\perp} \times \mathbb{N}_{\perp}) \longrightarrow \mathbb{N}_{\perp}$$

$$x_1 \underline{op}_{\perp} x_2 = \begin{cases} \text{Case}(x_1, x_2) \\ ([n_1], [n_2]) : [n_1 \text{ op } n_2] \\ \perp_{\mathbb{N}_{\perp}} \text{ otherwise} \end{cases}$$

notice that for every operator $op \in \{+, -, \times, \dots\}$ in the syntax we have the corresponding function \underline{op} on the integers \mathbb{N} and also the binary function \underline{op}_{\perp} on \mathbb{N}_{\perp} . Notice also that \underline{op}_{\perp} yields $\perp_{\mathbb{N}_{\perp}}$ when at least one of the two arguments is $\perp_{\mathbb{N}_{\perp}}$.

8.1.4. Conditional

In order to define the semantics of the conditional expression we use the conditional operator of the meta-language:

$$\llbracket \text{if } t_0 \text{ then } t_1 \text{ else } t_2 \rrbracket \rho = \text{Cond}(\llbracket t_0 \rrbracket \rho, \llbracket t_1 \rrbracket \rho, \llbracket t_2 \rrbracket \rho)$$

where

$$\text{Cond}_{\tau} : \mathbb{N}_{\perp} \times (V_{\tau})_{\perp} \times (V_{\tau})_{\perp} \longrightarrow (V_{\tau})_{\perp}$$

$$\text{Cond}_{\tau}(d_0, d_1, d_2) = \begin{cases} d_1 & \text{if } d_0 = [0] \\ d_2 & \text{if } d_0 = [n] \quad n \neq 0 \\ \perp_{(V_{\tau})_{\perp}} & \text{if } d_0 = \perp_{\mathbb{N}_{\perp}} \end{cases}$$

8.1.5. Pairing

For the pairing operator we have:

$$\llbracket (t_1, t_2) \rrbracket \rho = \llbracket ([\llbracket t_1 \rrbracket \rho], \llbracket t_2 \rrbracket \rho]) \rrbracket$$

8.1.6. Projections

We define the projections by using the lifted version of π_1 and π_2 functions of the meta-language:

$$\llbracket \text{fst}(t) \rrbracket \rho = \mathbf{let} \, d \Leftarrow \llbracket t \rrbracket \rho. \pi_1 d = (\lambda d. \pi_1 d)^* \llbracket t \rrbracket \rho$$

$$\llbracket \text{snd}(t) \rrbracket \rho = \mathbf{let} \, d \Leftarrow \llbracket t \rrbracket \rho. \pi_2 d = (\lambda d. \pi_2 d)^* \llbracket t \rrbracket \rho$$

as we said in the previous chapter the “let” operator allows to *de-lift* $\llbracket t \rrbracket \rho$ in order to apply projections π_1 and π_2 .

8.1.7. Lambda Abstraction

Obviously we use the lambda operator of the lambda calculus:

$$\llbracket \lambda x. t \rrbracket \rho = \llbracket \lambda d. \llbracket t \rrbracket \rho [d/x] \rrbracket$$

8.1.8. Function Application

We simply apply the de-lifted version of the function to its argument:

$$\llbracket (t_1 \ t_2) \rrbracket \rho = \mathbf{let} \, \varphi \Leftarrow \llbracket t_1 \rrbracket \rho. \varphi(\llbracket t_2 \rrbracket \rho) = (\lambda \varphi. \varphi(\llbracket t_2 \rrbracket \rho))^* \llbracket t_1 \rrbracket \rho$$

8.1.9. Recursion

We simply apply the fix operator of the meta-language:

$$\llbracket \text{rec } x.t \rrbracket \rho = \mathbf{fix} \lambda d. \llbracket t \rrbracket \rho [^d/x]$$

8.2. Typing the Clauses

Now we show that the clauses of the structural recursion are typed correctly.

Constants

$$\frac{\llbracket n : \text{int} \rrbracket \rho = \lfloor n \rfloor}{(V_{\text{int}})_{\perp} = \mathbb{N}_{\perp}} \quad \frac{\quad}{\mathbb{N}_{\perp}}$$

as required.

Variables

$$\frac{\llbracket x : \tau \rrbracket \rho = \rho x}{(V_{\tau})_{\perp}} \quad \frac{\quad}{(V_{\tau})_{\perp}}$$

Binary operations

$$\frac{\frac{\llbracket t_1 \text{ op } t_2 : \text{int} \rrbracket \rho}{(V_{\text{int}})_{\perp}} = \frac{\llbracket t_1 \rrbracket \rho}{(V_{\text{int}})_{\perp}} \quad \frac{\text{op}_{\perp}}{(V_{\text{int}})_{\perp} \times (V_{\text{int}})_{\perp} \rightarrow (V_{\text{int}})_{\perp}} \quad \frac{\llbracket t_2 \rrbracket \rho}{(V_{\text{int}})_{\perp}}}{(V_{\text{int}})_{\perp}}}{(V_{\text{int}})_{\perp}}$$

Conditional

$$\frac{\llbracket \text{if } t_0 : \text{int} \text{ then } t_1 : \tau \text{ else } t_2 : \tau \rrbracket \rho}{(V_{\tau})_{\perp}} = \frac{\text{Cond}_{\tau} \quad (\llbracket t_0 \rrbracket \rho, \llbracket t_1 \rrbracket \rho, \llbracket t_2 \rrbracket \rho)}{\mathbb{N}_{\perp} \times (V_{\tau})_{\perp} \times (V_{\tau})_{\perp} \rightarrow (V_{\tau})_{\perp} \quad \frac{(V_{\text{int}})_{\perp} \quad (V_{\tau})_{\perp} \quad (V_{\tau})_{\perp}}{(V_{\tau})_{\perp}}}}{(V_{\tau})_{\perp}}$$

Pairing

$$\frac{\llbracket (t_1 : \tau_1, t_2 : \tau_2) \rrbracket \rho}{(V_{\tau_1 * \tau_2})_{\perp}} = \lfloor \frac{\llbracket t_1 \rrbracket \rho}{(V_{\tau_1})_{\perp}} \quad \frac{\llbracket t_2 \rrbracket \rho}{(V_{\tau_2})_{\perp}} \rfloor$$

$$\frac{\quad}{(V_{\tau_1})_{\perp} \times (V_{\tau_2})_{\perp}}$$

$$\frac{\quad}{((V_{\tau_1})_{\perp} \times (V_{\tau_2})_{\perp})_{\perp}}$$

Projections

$$\frac{\llbracket \text{fst}(t : \tau_1 * \tau_2) \rrbracket \rho}{(V_{\tau_1})_{\perp}} = \mathbf{let} \quad \frac{d}{(V_{\tau_1})_{\perp} \times (V_{\tau_2})_{\perp}} \leftarrow \frac{\llbracket t \rrbracket \rho}{(V_{\tau_1 * \tau_2})_{\perp}} \quad \frac{\pi_1 \quad d}{(V_{\tau_1})_{\perp} \times (V_{\tau_2})_{\perp} \rightarrow (V_{\tau_1})_{\perp}}}{(V_{\tau_1})_{\perp}}$$

Lambda abstraction

$$\frac{\llbracket \lambda x : \tau_1. t : \tau_2 \rrbracket \rho}{(V_{\tau_1 \rightarrow \tau_2})_{\perp}} = \lfloor \lambda \quad \frac{d}{(V_{\tau_1})_{\perp}} \cdot \frac{\llbracket t \rrbracket \rho [^d/x]}{(V_{\tau_2})_{\perp}} \rfloor$$

$$\frac{\quad}{[(V_{\tau_1})_{\perp} \rightarrow (V_{\tau_2})_{\perp}]}$$

$$\frac{\quad}{[(V_{\tau_1})_{\perp} \rightarrow (V_{\tau_2})_{\perp}]_{\perp}}$$

Function application

$$\underbrace{\llbracket (t_1 : \tau_1 \longrightarrow \tau_2 \ t_2 : \tau_1) \rrbracket \rho}_{(V_{\tau_2})_{\perp}} = \mathbf{let} \quad \underbrace{\varphi}_{\llbracket (V_{\tau_1})_{\perp} \rightarrow (V_{\tau_2})_{\perp} \rrbracket}} \leftarrow \underbrace{\llbracket t_1 \rrbracket \rho}_{(V_{\tau_1 \rightarrow \tau_2})_{\perp}} \cdot \underbrace{\varphi(\llbracket t_2 \rrbracket \rho)}_{(V_{\tau_1})_{\perp}}$$

Recursion

$$\underbrace{\llbracket \mathbf{rec} \ x : \tau. t : \tau \rrbracket \rho}_{(V_{\tau})_{\perp}} = \underbrace{\mathbf{fix} \quad \underbrace{\lambda d. \llbracket t \rrbracket \rho[d/x]}_{(V_{\tau})_{\perp}}}_{\llbracket (V_{\tau})_{\perp} \rightarrow (V_{\tau})_{\perp} \rrbracket \rightarrow (V_{\tau})_{\perp}} \quad \underbrace{\llbracket t \rrbracket \rho[d/x]}_{(V_{\tau})_{\perp}}$$

Example 8.1

Let us see some examples of evaluation of the denotational semantics. We consider three similar terms f, g, h such that f and h have the same denotational semantics while g has a different semantics because it requires a parameter x to be evaluated even if it is not used.

1. $f \stackrel{\text{def}}{=} \lambda x : \text{int}. 3$

2. $g \stackrel{\text{def}}{=} \lambda x : \text{int}. \mathbf{if} \ x \ \mathbf{then} \ 3 \ \mathbf{else} \ 3$

3. $h \stackrel{\text{def}}{=} \mathbf{rec} \ y : \text{int} \rightarrow \text{int}. \lambda x : \text{int}. 3$

1. $\llbracket f \rrbracket \rho = \llbracket \lambda x : \text{int}. 3 \rrbracket \rho = \llbracket \lambda d. \llbracket 3 \rrbracket \rho[d/x] \rrbracket = \llbracket \lambda d. \llbracket 3 \rrbracket \rrbracket$

2. $\llbracket g \rrbracket \rho = \llbracket \lambda x : \text{int}. \mathbf{if} \ x \ \mathbf{then} \ 3 \ \mathbf{else} \ 3 \rrbracket \rho = \llbracket \lambda d. \llbracket \mathbf{if} \ x \ \mathbf{then} \ 3 \ \mathbf{else} \ 3 \rrbracket \rho[d/x] \rrbracket = \llbracket \lambda d. \mathbf{Cond}(d, \llbracket 3 \rrbracket, \llbracket 3 \rrbracket) \rrbracket = \llbracket \lambda d. \mathbf{let} \ x \leftarrow d. \llbracket 3 \rrbracket \rrbracket$

3. $\llbracket h \rrbracket \rho = \llbracket \mathbf{rec} \ y : \text{int} \rightarrow \text{int}. \lambda x : \text{int}. 3 \rrbracket \rho = \mathbf{fix} \ \lambda d. \llbracket \lambda x. 3 \rrbracket \rho[d/x] = \mathbf{fix} \ \lambda d. \llbracket \lambda d'. \llbracket 3 \rrbracket \rrbracket$

- $d_0 = \perp_{[\mathbb{N}_{\perp} \rightarrow \mathbb{N}_{\perp}]_{\perp}}$

- $d_1 = (\lambda d. \llbracket \lambda d'. \llbracket 3 \rrbracket \rrbracket)_{\perp} = \llbracket \lambda d'. \llbracket 3 \rrbracket \rrbracket$

- $d_2 = (\lambda d. \llbracket \lambda d'. \llbracket 3 \rrbracket \rrbracket) \llbracket \lambda d'. \llbracket 3 \rrbracket \rrbracket = \llbracket \lambda d'. \llbracket 3 \rrbracket \rrbracket = d_1 = \llbracket h \rrbracket \rho$

8.3. Continuity of Meta-language's Functions

In order to show that the semantics is well defined we have to show that all the functions we employ in the definition are continuous.

Theorem 8.2

The following functions are continuous:

- op_{\perp}
- \mathbf{Cond}
- $(_, _)$
- π_1, π_2
- \mathbf{let}

- *apply*
- *fix*

Proof.

- op_{\perp} : Since op_{\perp} is monotone over a domain with only finite chains then it is also continuous.
- *Cond*: By using the Theorem 7.13, we can prove the continuity on each parameter separately. Let us show the continuity on the first parameter. Since chains are finite, it is enough to prove monotonicity. We fix d_1 and d_2 and we prove the monotonicity of $Cond(x, d_1, d_2)$
 - for the case $\perp_{\mathbf{N}_{\perp}} \sqsubseteq n$ then obviously $\forall n \in \mathbf{N}_{\perp} Cond(\perp_{\mathbf{N}_{\perp}}, d_1, d_2) \sqsubseteq Cond(n, d_1, d_2)$, namely $\perp_{(v_{\tau})_{\perp}} \sqsubseteq d_1$ or $\perp_{(v_{\tau})_{\perp}} \sqsubseteq d_2$.
 - for the case $n \sqsubseteq n'$, since \mathbf{N}_{\perp} is a flat domain we have $n = n'$ so obviously $Cond(n, d_1, d_2) \sqsubseteq Cond(n', d_1, d_2)$

Now let us show the continuity on the second parameter, namely we fix v and d and prove that

$$Cond(v, \bigsqcup_{i \in \omega} d_i, d) = \bigsqcup_{i \in \omega} Cond(v, d_i, d)$$

- $v = \perp_{\mathbf{N}_{\perp}}$ *Cond* is the constant $\perp_{\mathbf{N}_{\perp}}$
- $v = 0$ it is the identity.
- $v = n \neq \perp_{\mathbf{N}_{\perp}}$ then it is the constant d

In all cases it is continuous.

Continuity on the third parameter is analogous.

- π_1 and π_2 are continuous as shown by Theorem 7.3.
- $(_, _)$ we can use the Theorem 7.13 which allows to show separately the continuity on each parameter. If we fix the first element we have $(d, \bigsqcup_{i \in \omega} d_i) = \bigsqcup_{i \in \omega} (d, d_i)$ by Theorem 7.1. The same holds for the second parameter.
- the **let** function is continuous since $(_)^*$ is continuous by Theorem 7.10.
- *apply* is continuous as shown by Theorem 7.15
- *fix* is continuous as shown by Theorem 7.19.

□

In the previous theorem we have omitted the proofs for lambda abstraction and recursion, in the next theorem we fill these gaps.

Theorem 8.3

Let t be a well typed term of HOFL then the following holds:

- $(\lambda d : (V_{\tau_1})_{\perp}. \llbracket t : \tau_2 \rrbracket \rho^{d/x}) : (V_{\tau_1})_{\perp} \longrightarrow (V_{\tau_2})_{\perp}$ is a continuous function.
- $\text{fix } \lambda d. \llbracket t \rrbracket \rho^{d/x}$ is a continuous function.

Proof. Let us show the first proposition by structural induction on t and for any number of arguments $\rho^{d/x}[\rho^{d'/y}] \dots$

- $t = x$: $\lambda d. \llbracket x \rrbracket \rho^{d/x}$ is equal to $\lambda d. d$ which is obviously continuous.
- $t = t_1 \text{ op } t_2$: $\lambda d. \llbracket t_1 \text{ op } t_2 \rrbracket \rho^{d/x} = \lambda d. \llbracket t_1 \rrbracket \rho^{d/x} \text{ op } \llbracket t_2 \rrbracket \rho^{d/x}$ it is continuous since op_{\perp} , $\llbracket t_1 \rrbracket \rho^{d/x}$ and $\llbracket t_2 \rrbracket \rho^{d/x}$ are continuous by Theorem 8.2 and by the inductive hypothesis, and since the composition of continuous functions yields a continuous function by Theorem 7.11.

- $t = \lambda y.t'$: $\lambda d. \llbracket \lambda y.t' \rrbracket \rho^{[d/x]}$ is obviously continuous if $x = y$. Otherwise if $x \neq y$ we have by induction hypothesis that $\lambda(d, d'). \llbracket t' \rrbracket \rho^{[d/x, d'/y]}$ is continuous, then:

$$\begin{aligned} \text{curry}(\lambda(d, d'). \llbracket t' \rrbracket \rho^{[d/x, d'/y]}) &= \text{is continuous since curry is continuous} \\ \lambda d. \lambda d'. \llbracket t' \rrbracket \rho^{[d/x][d'/y]} &= \\ \lambda d. \llbracket \lambda y.t' \rrbracket \rho^{[d/x]} & \end{aligned}$$

We leave the remaining cases as an exercise.

To prove the second proposition we note that, fix $\lambda d. \llbracket t \rrbracket \rho^{[d/x]}$ is the application of a continuous function (i.e. fix by Th. 7.19) to its continuous argument $(\lambda d. \llbracket t \rrbracket \rho^{[d/x]})$, by the first part of this theorem) so it is continuous by Th. 7.15. \square

8.4. Substitution Lemma

We conclude this chapter by stating some useful theorems. The most important is the *Substitution Lemma* which states that the substitution operator commutes with the interpretation function.

Theorem 8.4 (Substitution Lemma)

Let t, t' be well typed terms: we have

$$\llbracket t[t'/x] \rrbracket \rho = \llbracket t \rrbracket \rho[\llbracket t' \rrbracket \rho / x]$$

Proof. By structural induction. \square

The substitution lemma is an important result, as it implies the compositionality of denotational semantics:

$$\llbracket t_1 \rrbracket \rho = \llbracket t_2 \rrbracket \rho \Rightarrow \llbracket t[t_1/x] \rrbracket \rho = \llbracket t[t_2/x] \rrbracket \rho$$

In words, replacing a variable x with a term t' in a term t returns a term $t[t'/x]$ whose denotational semantics $\llbracket t[t'/x] \rrbracket \rho = \llbracket t \rrbracket \rho[\llbracket t' \rrbracket \rho / x]$ depends only on the denotational semantics $\llbracket t' \rrbracket \rho$ of t' and *not* on t itself.

Theorem 8.5

Let t be a well-defined term of HOFL. Let $\rho, \rho' \in \text{Env}$ such that $\forall x \in \text{FV}(t). \rho(x) = \rho'(x)$ then:

$$\llbracket t \rrbracket \rho = \llbracket t \rrbracket \rho'$$

Proof. By structural induction. \square

Theorem 8.6

Let $c \in C_\tau$ be a closed term in canonical form of type τ . Then we have:

$$\forall \rho \in \text{Env}. \llbracket c \rrbracket \rho \neq \perp_{(V_\tau)_\perp}$$

Proof. Immediate, by inspection of the clauses for terms in canonical forms. \square

9. Equivalence between HOFL denotational and operational semantics

As we have done for IMP, now we address the relation between the denotational and operational semantics of HOFL. We would like to prove a complete equivalence, as in the case of IMP:

$$t \longrightarrow c \Leftrightarrow \llbracket t \rrbracket \rho = \llbracket c \rrbracket \rho$$

But, as we are going to show, the situation in the case of HOFL is more complex and the \Leftarrow case does not hold, i.e.:

$$t \longrightarrow c \Rightarrow \llbracket t \rrbracket \rho = \llbracket c \rrbracket \rho \quad \text{but} \quad \llbracket t \rrbracket \rho = \llbracket c \rrbracket \rho \not\Rightarrow t \longrightarrow c$$

Let us consider an example which shows the difference between the denotational and the operational semantics.

Example 9.1

Let $c = \lambda x. x$ and $t = \lambda x. x + 0$ be two HOFL terms, where $x : \text{int}$, then we have:

$$\llbracket t \rrbracket \rho = \llbracket c \rrbracket \rho$$

but

$$t \not\rightarrow c$$

In fact we have:

$$\llbracket t \rrbracket \rho = \llbracket \lambda x. x + 0 \rrbracket \rho = \llbracket \lambda d. d_{\perp} [0] \rrbracket \rho = \llbracket \lambda d. d \rrbracket \rho = \llbracket \lambda x. x \rrbracket \rho = \llbracket c \rrbracket \rho$$

but for the operational semantics we have that both $\lambda x.x$ and $\lambda x. x + 0$ are already in canonical form and $t \neq c$.

The counterexample shows that at least for the functional type $\text{int} \rightarrow \text{int}$, there are different canonical forms with the same denotational semantics, namely terms which compute the same function $[\mathbb{N}_{\perp} \rightarrow \mathbb{N}_{\perp}]_{\perp}$. One could think that a refined version of our operational semantics (e.g. one which could apply an axiom like $x + 0 = 0$) would be able to identify exactly all the canonical forms which compute the same function. However this is not possible on computability grounds: since HOFL is able to compute all computable functions, the set of canonical terms which compute the same function can be not recursively enumerable, while the set of theorems of every inference system is recursively enumerable.

Now we define the concept of termination for the denotational and the operational semantics.

Definition 9.2 (Operational convergence)

Let t be a closed term of HOFL with type τ , we define the following predicate:

$$t \downarrow \iff \exists c \in C_{\tau}. t \longrightarrow c$$

if the predicate holds for t , then we say that t converges operationally.

We say that t diverges and write $t \uparrow$ if t does not converge.

Obviously, a term t converges operationally if the term can be evaluated to a canonical form c . For the denotational semantics we have that a term t converges if the evaluation function applied to t takes a value different from \perp .

Definition 9.3 (Denotational convergence)

Let t be a closed term of HOFL with type τ , we define the following predicate:

$$t \Downarrow \iff \exists v \in V_\tau. \llbracket t \rrbracket \rho = \lfloor v \rfloor$$

if the predicate holds for t then we say that t converges denotationally.

We aim to prove that the two semantics agree at least on the notion of convergence.

As we will see, we can easily prove the implication:

$$t \Downarrow \implies t \Downarrow$$

For the opposite implication, the property holds but the proof is not straightforward: We cannot simply rely on structural induction; instead it is necessary to introduce a particular order relation.

9.1. Completeness

We are ready to show the completeness of the denotational semantics of HOFL w.r.t. the operational one.

Theorem 9.4 (Completeness)

Let t be a HOFL closed term with type τ and let c be a canonical form with type τ then we have:

$$t \rightarrow c \implies \forall \rho \in Env \llbracket t \rrbracket \rho = \llbracket c \rrbracket \rho$$

Proof. As usual we proceed by rule induction. So we will prove $P(t \rightarrow c) \equiv \forall \rho. \llbracket t \rrbracket \rho = \llbracket c \rrbracket \rho$ on each rule by assuming the premises.

$$c \rightarrow c$$

We have to prove $P(c \rightarrow c)$ that is by definition $\forall \rho. \llbracket c \rrbracket \rho = \llbracket c \rrbracket \rho$, which is obviously true.

$$\frac{t_1 \rightarrow n_1 \quad t_2 \rightarrow n_2}{t_1 \text{ op } t_2 \rightarrow n_1 \text{ op } n_2}$$

We have to prove $P(t_1 \text{ op } t_2 \rightarrow n_1 \text{ op } n_2) \equiv \forall \rho. \llbracket t_1 \text{ op } t_2 \rrbracket \rho = \llbracket n_1 \text{ op } n_2 \rrbracket \rho$.

Let us assume the property on the premises:

$$P(t_1 \rightarrow n_1) \equiv \forall \rho. \llbracket t_1 \rrbracket \rho = \llbracket n_1 \rrbracket \rho = \lfloor n_1 \rfloor \text{ and } P(t_2 \rightarrow n_2) \equiv \forall \rho. \llbracket t_2 \rrbracket \rho = \llbracket n_2 \rrbracket \rho = \lfloor n_2 \rfloor.$$

We have $\llbracket t_1 \text{ op } t_2 \rrbracket \rho = \llbracket t_1 \rrbracket \rho \text{ op } \llbracket t_2 \rrbracket \rho = \lfloor n_1 \rfloor \text{ op } \lfloor n_2 \rfloor = \lfloor n_1 \text{ op } n_2 \rfloor = \llbracket n_1 \text{ op } n_2 \rrbracket \rho$.

$$\frac{t_0 \rightarrow 0 \quad t_1 \rightarrow c_1}{\text{if } t_0 \text{ then } t_1 \text{ else } t_2 \rightarrow c_1}$$

We will prove $P(\text{if } t_0 \text{ then } t_1 \text{ else } t_2 \rightarrow c_1) \equiv \forall \rho. \llbracket \text{if } t_0 \text{ then } t_1 \text{ else } t_2 \rrbracket \rho = \llbracket c_1 \rrbracket \rho$.

We assume $P(t_0 \rightarrow 0) \equiv \forall \rho. \llbracket t_0 \rrbracket \rho = \llbracket 0 \rrbracket \rho = \lfloor 0 \rfloor$ and $P(t_1 \rightarrow c_1) \equiv \forall \rho. \llbracket t_1 \rrbracket \rho = \llbracket c_1 \rrbracket \rho$.

We have $\llbracket \text{if } t_0 \text{ then } t_1 \text{ else } t_2 \rrbracket \rho = \text{Cond}(\llbracket t_0 \rrbracket \rho, \llbracket t_1 \rrbracket \rho, \llbracket t_2 \rrbracket \rho) = \text{Cond}(\lfloor 0 \rfloor, \llbracket t_1 \rrbracket \rho, \llbracket t_2 \rrbracket \rho) = \llbracket t_1 \rrbracket \rho = \llbracket c_1 \rrbracket \rho$.

The same construction holds for the second rule of the conditional operator.

$$\frac{t \rightarrow (t_1, t_2) \quad t_1 \rightarrow c_1}{\mathbf{fst}(t) \rightarrow c_1}$$

We prove $P(\mathbf{fst}(t) \rightarrow c_1) \equiv \forall \rho. \llbracket \mathbf{fst}(t) \rrbracket \rho = \llbracket c_1 \rrbracket \rho$.

We assume $P(t \rightarrow (t_1, t_2)) \equiv \forall \rho. \llbracket t \rrbracket \rho = \llbracket (t_1, t_2) \rrbracket \rho$ and $P(t_1 \rightarrow c_1) \equiv \forall \rho. \llbracket t_1 \rrbracket \rho = \llbracket c_1 \rrbracket \rho$.

By definition $\llbracket \mathbf{fst}(t) \rrbracket \rho = \mathbf{let} \ v \leftarrow \llbracket t \rrbracket \rho. \pi_1 v$.

By inductive hypothesis $\mathbf{let} \ v \leftarrow \llbracket t \rrbracket \rho. \pi_1 v = \mathbf{let} \ v \leftarrow \llbracket (t_1, t_2) \rrbracket \rho. \pi_1 v$.

By definition $\mathbf{let} \ v \leftarrow \llbracket (t_1, t_2) \rrbracket \rho. \pi_1 v = \mathbf{let} \ v \leftarrow [\llbracket t_1 \rrbracket \rho, \llbracket t_2 \rrbracket \rho]. \pi_1 v = \llbracket t_1 \rrbracket \rho = \llbracket c_1 \rrbracket \rho$.

The same holds for the **snd** operator.

$$\frac{t_1 \rightarrow \lambda x. t'_1 \quad t'_1 [t_2/x] \rightarrow c}{(t_1 \ t_2) \rightarrow c}$$

We prove $P((t_1 \ t_2) \rightarrow c) \equiv \forall \rho. \llbracket (t_1 \ t_2) \rrbracket \rho = \llbracket c \rrbracket \rho$.

We assume $P(t_1 \rightarrow \lambda x. t'_1) \equiv \forall \rho. \llbracket t_1 \rrbracket \rho = \llbracket \lambda x. t'_1 \rrbracket \rho$ and $P(t'_1 [t_2/x] \rightarrow c) \equiv \forall \rho. \llbracket t'_1 [t_2/x] \rrbracket \rho = \llbracket c \rrbracket \rho$.

By definition $\llbracket (t_1 \ t_2) \rrbracket \rho = \mathbf{let} \ \varphi \leftarrow \llbracket t_1 \rrbracket \rho. \varphi(\llbracket t_2 \rrbracket \rho)$.

By inductive hypothesis $\mathbf{let} \ \varphi \leftarrow \llbracket t_1 \rrbracket \rho. \varphi(\llbracket t_2 \rrbracket \rho) = \mathbf{let} \ \varphi \leftarrow \llbracket \lambda x. t'_1 \rrbracket \rho. \varphi(\llbracket t_2 \rrbracket \rho)$.

By denotational semantics:

$$\begin{aligned} \mathbf{let} \ \varphi \leftarrow \llbracket \lambda x. t'_1 \rrbracket \rho. \varphi(\llbracket t_2 \rrbracket \rho) &= \mathbf{let} \ \varphi \leftarrow [\lambda d. \llbracket t'_1 \rrbracket \rho [d/x]]. \varphi(\llbracket t_2 \rrbracket \rho) \\ &= (\lambda d. \llbracket t'_1 \rrbracket \rho [d/x])(\llbracket t_2 \rrbracket \rho) \\ &= \llbracket t'_1 \rrbracket \rho [t_2/x]. \end{aligned}$$

Finally, by using the substitution lemma and by the second inductive hypothesis we have:

$$\llbracket t'_1 \rrbracket \rho [t_2/x] = \llbracket t'_1 [t_2/x] \rrbracket \rho = \llbracket c \rrbracket \rho.$$

$$\frac{t[\mathbf{rec} \ x.t/x] \rightarrow c}{\mathbf{rec} \ x.t \rightarrow c}$$

We prove $P(\mathbf{rec} \ x.t \rightarrow c) \equiv \forall \rho. \llbracket \mathbf{rec} \ x.t \rrbracket \rho = \llbracket c \rrbracket \rho$.

We assume $P(t[\mathbf{rec} \ x.t/x] \rightarrow c) \equiv \forall \rho. \llbracket t[\mathbf{rec} \ x.t/x] \rrbracket \rho = \llbracket c \rrbracket \rho$.

By definition we have $\llbracket \mathbf{rec} \ x.t \rrbracket \rho = \mathbf{fix} \ \lambda d. \llbracket t \rrbracket \rho [d/x]$.

By the fixpoint property $\mathbf{fix} \ \lambda d. \llbracket t \rrbracket \rho [d/x] = \llbracket t \rrbracket \rho [\mathbf{rec} \ x.t/x]$.

By the substitution lemma and by induction hypothesis $\llbracket t \rrbracket \rho [\mathbf{rec} \ x.t/x] = \llbracket t[\mathbf{rec} \ x.t/x] \rrbracket \rho = \llbracket c \rrbracket \rho$.

□

9.2. Equivalence (on Convergence)

As we have anticipated at the beginning of this chapter, the operational and the denotational semantics agree on convergence. We are not giving the full details of the proof, but we show that the standard structural induction does not help in proving the (left implication of) convergence agreement. Those who are interested in the full proof can refer to Winskel's book referenced in the introduction.

Theorem 9.5

Let t be a close typable term of HOFL. Then we have:

$$t \Downarrow \implies t \Downarrow$$

Proof. If $t \longrightarrow c$, then $\llbracket t \rrbracket \rho = \llbracket c \rrbracket \rho$ for the previous theorem. But $\llbracket c \rrbracket \rho$ is always a lifted value, and thus it is not \perp . \square

We give some insight on the reason why the usual structural induction does not work for proving $t \Downarrow \implies t \Downarrow$. Let us consider function application $(t_1 t_2)$. We assume by structural induction $t_1 \Downarrow \implies t_1 \Downarrow$ and $t_2 \Downarrow \implies t_2 \Downarrow$. Now we assume $(t_1 t_2) \Downarrow$, so by definition of denotational semantics we have $t_1 \Downarrow$. In fact

$$\llbracket (t_1 t_2) \rrbracket \rho = \mathbf{let} \varphi \Leftarrow \llbracket t_1 \rrbracket \rho. \varphi(\llbracket t_2 \rrbracket \rho)$$

and therefore $\llbracket (t_1 t_2) \rrbracket \rho \neq \perp$ requires $\llbracket t_1 \rrbracket \rho \neq \perp$. By induction we then have $t_1 \Downarrow$ and by definition of the operational semantics $t_1 \longrightarrow \lambda x.t'_1$ for some t'_1 and we also have $\llbracket t_1 \rrbracket \rho = \llbracket \lambda x.t'_1 \rrbracket \rho$. By denotational semantics definition we have:

$$\begin{aligned} \llbracket (t_1 t_2) \rrbracket \rho &= \mathbf{let} \varphi \Leftarrow \lambda d. \llbracket t'_1 \rrbracket \rho[d/x]. \varphi(\llbracket t_2 \rrbracket \rho) \\ &= (\lambda d. \llbracket t'_1 \rrbracket \rho[d/x])(\llbracket t_2 \rrbracket \rho) \\ &= \llbracket t'_1 \rrbracket \rho[\llbracket t_2 \rrbracket \rho/x] \\ &= \llbracket t'_1[t_2/x] \rrbracket \rho \end{aligned}$$

So if $(t_1 t_2) \Downarrow$ then also $t'_1[t_2/x] \Downarrow$ and we would like to conclude by structural induction that $t'_1[t_2/x] \Downarrow$ and then prove the theorem by using the rule:

$$\frac{t_1 \longrightarrow \lambda x.t'_1 \quad t'_1[t_2/x] \longrightarrow c}{(t_1 t_2) \longrightarrow c}$$

but this is incorrect since $t'_1[t_2/x]$ is not a sub-term of $(t_1 t_2)$.

9.3. Operational and Denotational Equivalence

In this section we take a closer look at the relationship between the operational and denotational semantics of HOFL. In the introduction of this chapter we said that the denotational semantics is more abstract than the operational. In order to study this relationship we now introduce two equivalence relations between terms. Operationally two terms are equivalent if they both diverge or have the same canonical form.

Definition 9.6 (Operational equivalence)

Let t_1 and t_2 be two well-typed terms of HOFL then we define a binary relation:

$$t_1 \equiv_{op} t_2 \iff (t_1 \Uparrow \wedge t_2 \Uparrow) \vee (t_1 \rightarrow c \wedge t_2 \rightarrow c)$$

And we say that t_1 is operationally equivalent to t_2 .

Obviously we have the denotational counterpart of the definition.

Definition 9.7 (Denotational equivalence)

Let t_1 and t_2 be two well-typed terms of HOFL then we define a binary relation:

$$t_1 \equiv_{den} t_2 \iff \forall \rho. \llbracket t_1 \rrbracket \rho = \llbracket t_2 \rrbracket \rho$$

And we say that t_1 is denotationally equivalent to t_2 .

From Theorem 9.4 it follows that:

$$\equiv_{op} \Rightarrow \equiv_{den}$$

As pointed out in Example 9.1, the opposite does not hold:

$$\equiv_{den} \not\Rightarrow \equiv_{op}$$

So in this sense we can say that the denotational semantics is more abstract than the operational one. Note that if we assume $t_1 \equiv_{den} t_2$ and $t_1, t_2 \neq \perp$ then we can only conclude that $t_1 \rightarrow c_1$ and $t_2 \rightarrow c_2$ for some suitable c_1 and c_2 . So we have $\llbracket c_1 \rrbracket \rho = \llbracket c_2 \rrbracket \rho$, but nothing ensures that $c_1 = c_2$ as shown in the Example 9.1 at the beginning of this chapter.

Now we prove that if we restrict our attention only to the integers terms of HOFL, then the corresponding operational and denotational semantics completely agree. This is due to the fact that if c_1 and c_2 are canonical forms in C_{int} then it holds that $\llbracket c_1 \rrbracket \rho = \llbracket c_2 \rrbracket \rho \Leftrightarrow c_1 = c_2$.

Theorem 9.8

Let t be a closed integer term of HOFL (i.e., $t : int$). Then:

$$\llbracket t \rrbracket \rho = \lfloor n \rfloor \iff t \longrightarrow n$$

Proof.

\Rightarrow) If $\llbracket t \rrbracket \rho = \lfloor n \rfloor$, then $t \Downarrow$ and thus $t \Downarrow$ by the soundness of denotational semantics (not proved here), namely $\exists n'$ such that $t \longrightarrow n'$, but $\llbracket t \rrbracket \rho = \lfloor n' \rfloor$ by Theorem 9.4, thus $n = n'$ and $t \longrightarrow n$.

\Leftarrow) Just Theorem 9.4.

□

9.4. A Simpler Denotational Semantics

In this section we introduce a simpler denotational semantics which we call *unlifted*, which does not use the lifted domains. This semantics is simpler but also less expressive than the lifted one.

We define the following new domains:

$$D_{int} = \mathbf{N}_\perp$$

$$D_{\tau_1 * \tau_2} = D_{\tau_1} \times D_{\tau_2}$$

$$D_{\tau_1 \rightarrow \tau_2} = [D_{\tau_1} \rightarrow D_{\tau_2}]$$

So we can simply define the interpretation function as follows:

$$\begin{aligned}
\llbracket n \rrbracket \rho &= \lfloor n \rfloor \\
\llbracket x \rrbracket \rho &= \rho x \\
\llbracket t_1 \text{ op } t_2 \rrbracket \rho &= \llbracket t_1 \rrbracket \rho \text{ op } \llbracket t_2 \rrbracket \rho \\
\llbracket \text{if } t_0 \text{ then } t_1 \text{ else } t_2 \rrbracket \rho &= \text{Cond}(\llbracket t_0 \rrbracket \rho, \llbracket t_1 \rrbracket \rho, \llbracket t_2 \rrbracket \rho) \\
\llbracket (t_1, t_2) \rrbracket \rho &= (\llbracket t_1 \rrbracket \rho, \llbracket t_2 \rrbracket \rho) \\
\llbracket \text{fst}(t) \rrbracket \rho &= \pi_1(\llbracket t \rrbracket \rho) \\
\llbracket \text{snd}(t) \rrbracket \rho &= \pi_2(\llbracket t \rrbracket \rho) \\
\llbracket \lambda x. t \rrbracket \rho &= \lambda d. \llbracket t \rrbracket \rho[d/x] \\
\llbracket (t_1 \ t_2) \rrbracket \rho &= (\llbracket t_1 \rrbracket \rho)(\llbracket t_2 \rrbracket \rho) \\
\llbracket \text{rec } x. t \rrbracket \rho &= \text{fix } \lambda d. \llbracket t \rrbracket \rho[d/x]
\end{aligned}$$

Note that the “unlifted” semantics differ from the “lifted” one only in the cases of pairing, projections, abstraction and application. Obviously, on the one hand this denotational semantics is much simpler. On the other hand this semantics is more abstract than the previous and does not express some interesting properties. For instance, consider the two HOFL terms:

$$t_1 = \text{rec } x. x : \text{int} \longrightarrow \text{int} \quad \text{and} \quad t_2 = \lambda x. \text{rec } y. y : \text{int} \longrightarrow \text{int}$$

In the lifted semantics we have $\llbracket t_1 \rrbracket \rho = \perp_{(\mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp)_\perp}$ and $\llbracket t_2 \rrbracket \rho = \lfloor \perp_{\mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp} \rfloor$ while in unlifted semantics $\llbracket t_1 \rrbracket \rho = \llbracket t_2 \rrbracket \rho = \perp_{\mathbb{N}_\perp \longrightarrow \mathbb{N}_\perp}$. Note however that $t_1 \not\downarrow$ while $t_2 \downarrow$, thus the completeness property $t \downarrow \Rightarrow \Downarrow t$ does not hold for the unlifted semantics, at least for $t : \text{int} \longrightarrow \text{int}$, since $t_2 \downarrow$ but $t_2 \not\Downarrow$. However, completeness holds for the unlifted semantics in the case of integers.

As a final comment, notice that the existence of two different, both reasonable, denotational semantics for HOFL shows that denotational semantics is, to some extent, an arbitrary construction, which depends on the properties one wants to express.