Dipartimento di Informatica, Università di Pisa

Notes on

Models of Computation

Introduction, Preliminaries, Operational Semantics of IMP, Induction, Recursion, Partial Orders, Denotational Semantics of IMP, Operational Semantics of HOFL, Domain Theory, Denotational Semantics of HOFL, CCS, Temporal Logic, μ-calculus, π-calculus Markov Chains with Actions and Nondeterminism, PEPA

Roberto Bruni Lorenzo Galeotti Ugo Montanari*

February 20, 2015

---

*Also based on material by Andrea Cimino, Lorenzo Muti, Gianmarco Saba, Marco Stronati

Contents

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

1. Preliminaries **1**
1.1. Inference Rules 1
1.2. Logic Programming 6

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 20
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 31
3.1.3. Weak Mathematical Induction 32
3.1.4. Strong Mathematical Induction 32
3.1.5. Structural Induction 32
3.1.6. Induction on Derivations 34
3.1.7. Rule Induction 34
3.2. Well-founded Recursion 37

4. Partial Orders and Fixpoints **39**
4.1. Orderings and Continuous Functions 39
4.1.1. Orderings 39
4.1.2. Hasse Diagrams 40
4.1.3. Chains 43
4.1.4. Complete Partial Orders 44

4.2. Continuity and Fixpoints . . . 46
  4.2.1. Monotone and Continuous Functions . . . 46
  4.2.2. Fixpoints . . . 46
  4.2.3. Fixpoint Theorem . . . 47
4.3. Immediate Consequence Operator . . . 48
  4.3.1. The $\hat{R}$ Operator . . . 48
  4.3.2. Fixpoint of $\hat{R}$ . . . 49

5. Denotational Semantics of IMP 53
5.1. λ-notation . . . 53
5.2. Denotational Semantics of IMP . . . 55
  5.2.1. Function $\mathscr{A}$ . . . 55
  5.2.2. Function $\mathscr{B}$ . . . 56
  5.2.3. Function $\mathscr{C}$ . . . 56
5.3. Equivalence Between Operational and Denotational Semantics . . . 59
  5.3.1. Equivalence Proofs for $\mathscr{A}$ and $\mathscr{B}$ . . . 59
  5.3.2. Equivalence of $\mathscr{C}$ . . . 59
    5.3.2.1. Completeness of the Denotational Semantics . . . 60
    5.3.2.2. Correctness of the Denotational Semantics . . . 61
5.4. Computational Induction . . . 63

II. HOFL language 67

6. Operational Semantics of HOFL 69
6.1. HOFL . . . 69
  6.1.1. Typed Terms . . . 69
  6.1.2. Typability and Typechecking . . . 72
    6.1.2.1. Church Type Theory . . . 72
    6.1.2.2. Curry Type Theory . . . 72
6.2. Operational Semantics of HOFL . . . 73

7. Domain Theory 77
7.1. The Domain $\mathbb{N}_\perp$ . . . 77
7.2. Cartesian Product of Two Domains . . . 77
7.3. Functional Domains . . . 78
7.4. Lifting . . . 80
7.5. Function's Continuity Theorems . . . 81
7.6. Useful Functions . . . 83

8. HOFL Denotational Semantics 87
8.1. HOFL Evaluation Function . . . 87
  8.1.1. Constants . . . 87
  8.1.2. Variables . . . 87
  8.1.3. Binary Operators . . . 88
  8.1.4. Conditional . . . 88
  8.1.5. Pairing . . . 88
  8.1.6. Projections . . . 88
  8.1.7. Lambda Abstraction . . . 88
  8.1.8. Function Application . . . 88
  8.1.9. Recursion . . . 89
8.2. Typing the Clauses . . . 89
8.3. Continuity of Meta-language's Functions . . . 90

8.4. Substitution Lemma 92

9. Equivalence between HOFL denotational and operational semantics 93
9.1. Completeness 94
9.2. Equivalence (on Convergence) 95
9.3. Operational and Denotational Equivalence 96
9.4. A Simpler Denotational Semantics 97

III. Concurrency and Logic 99

10. CCS, the Calculus for Communicating Systems 101
10.1. Syntax of CCS 104
10.2. Operational Semantics of CCS 105
10.3. Abstract Semantics of CCS 108
  10.3.1. Graph Isomorphism 108
  10.3.2. Trace Equivalence 109
  10.3.3. Bisimilarity 109
10.4. Compositionality 113
  10.4.1. Bisimilarity is Preserved by Parallel Composition 114
10.5. Hennessy - Milner Logic 115
10.6. Axioms for Strong Bisimilarity 117
10.7. Weak Semantics of CCS 118
  10.7.1. Weak Bisimilarity 118
  10.7.2. Weak Observational Congruence 119
  10.7.3. Dynamic Bisimilarity 120

11. Temporal Logic and μ-Calculus 123
11.1. Temporal Logic 123
  11.1.1. Linear Temporal Logic 123
  11.1.2. Computation Tree Logic 125
11.2. μ-Calculus 127
11.3. Model Checking 128

12. π-Calculus 129
12.1. Syntax of π-calculus 130
12.2. Operational Semantics of π-calculus 131
12.3. Structural Equivalence of π-calculus 133
  12.3.1. Reduction semantics 133
12.4. Abstract Semantics of π-calculus 134
  12.4.1. Strong Early Ground Bisimulations 134
  12.4.2. Strong Late Ground Bisimulations 135
  12.4.3. Strong Full Bisimilarity 135
  12.4.4. Weak Early and Late Ground Bisimulations 136

IV. Probabilistic Models and PEPA 137

13. Measure Theory and Markov Chains 139
13.1. Measure Theory 139
  13.1.1. σ-field 139
  13.1.2. Constructing a σ-field 140
  13.1.3. Continuous Random Variables 141
13.2. Stochastic Processes 144

13.3. Markov Chains 145
  13.3.1. Discrete and Continuous Time Markov Chain 145
  13.3.2. DTMC as LTS 146
  13.3.3. DTMC Steady State Distribution 148
  13.3.4. CTMC as LTS 149
  13.3.5. Embedded DTMC of a CTMC 149
  13.3.6. CTMC Bisimilarity 150
  13.3.7. DTMC Bisimilarity 151

14. Markov Chains with Actions and Non-determinism 153
14.1. Discrete Markov Chains With Actions 153
  14.1.1. Reactive DTMC 153
    14.1.1.1. Larsen-Skou Logic 154
  14.1.2. DTMC With Non-determinism 155
    14.1.2.1. Segala Automata 155
    14.1.2.2. Simple Segala Automata 156
    14.1.2.3. Non-determinism, Probability and Actions 157

15. PEPA - Performance Evaluation Process Algebra 159
15.1. CSP 160
  15.1.1. Syntax of CSP 160
  15.1.2. Operational Semantics of CSP 160
15.2. PEPA 160
  15.2.1. Syntax of PEPA 161
  15.2.2. Operational Semantics of PEPA 162

V. Appendices 167

A. Summary 169
A.1. Induction rules 3.1.2 169
  A.1.1. Noether 169
  A.1.2. Weak Mathematical Induction 3.1.3 169
  A.1.3. Strong Mathematical Induction 3.1.4 169
  A.1.4. Structural Induction 3.1.5 169
  A.1.5. Derivation Induction 3.1.6 169
  A.1.6. Rule Induction 3.1.7 169
  A.1.7. Computational Induction 5.4 169
A.2. IMP 2 170
  A.2.1. IMP Syntax 2.1 170
  A.2.2. IMP Operational Semantics 2.2 170
    A.2.2.1. IMP Arithmetic Expressions 170
    A.2.2.2. IMP Boolean Expressions 170
    A.2.2.3. IMP Commands 170
  A.2.3. IMP Denotational Semantics 5 170
    A.2.3.1. IMP Arithmetic Expressions $\mathscr{A} : Aexpr \to (\Sigma \to \mathbb{N})$ 170
    A.2.3.2. IMP Boolean Expressions $\mathscr{B} : Bexpr \to (\Sigma \to \mathbb{B})$ 171
    A.2.3.3. IMP Commands $\mathscr{C} : Com \to (\Sigma \rightharpoonup \Sigma)$ 171
A.3. HOFL 6.1 171
  A.3.1. HOFL Syntax 6.1 171
  A.3.2. HOFL Types 6.1.1 171
  A.3.3. HOFL Typing Rules 6.1.1 171

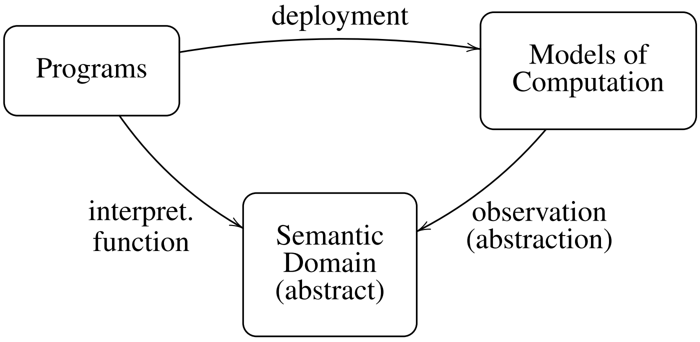
A.3.4. HOFL Operational Semantics 6.2 172
A.3.4.1. HOFL Canonical Forms 172
A.3.4.2. HOFL Axiom 172
A.3.4.3. HOFL Arithmetic and Conditional Expressions 172
A.3.4.4. HOFL Pairing Rules 172
A.3.4.5. HOFL Function Application 172
A.3.4.6. HOFL Recursion 172
A.3.5. HOFL Denotational Semantics 8 $[\![t : \tau]\!] : Env \longrightarrow (V_\tau)_\perp$ 172
A.4. CCS 10 173
A.4.1. CCS Syntax 10.1 173
A.4.2. CCS Operational Semantics 10.2 173
A.4.3. CCS Abstract Semantics 10.3 173
A.4.3.1. CCS Strong Bisimulation 10.3.3 173
A.4.3.2. CCS Weak Bisimulation 10.7 173
A.4.3.3. CCS Observational Congruence 10.7.2 173
A.4.3.4. CCS Axioms for Observational Congruence (Milner τ Laws) 10.7.2 . . . 174
A.4.3.5. CCS Dynamic Bisimulation 10.7.3 174
A.4.3.6. CCS Axioms for Dynamic Bisimilarity 10.7.3 174
A.5. Temporal and Modal Logic 174
A.5.1. Hennessy - Milner Logic 10.5 174
A.5.2. Linear Temporal Logic 11.1.1 174
A.5.3. Computation Tree Logic 11.1.2 175
A.6. μ-Calculus 11.2 175
A.7. π-calculus 12 176
A.7.1. π-calculys Syntax 12.1 176
A.7.2. π-calculus Operational Semantics 12.2 176
A.7.3. π-calculus Abstract Semantics 12.4 176
A.7.3.1. Strong Early Ground Bisimulation 12.4.1 176
A.7.3.2. Strong Early Full Bisimilarity 12.4.3 176
A.7.3.3. Strong Late Ground Bisimulation 12.4.2 177
A.7.3.4. Strong Late Full Bisimilarity 12.4.3 177
A.7.3.5. Weak Early Ground Bisimulation 12.4.4 177
A.7.3.6. Weak Late Ground Bisimulation 12.4.4 177
A.8. LTL for Action, Non-determinism and Probability 177
A.9. Larsen-Skou Logic 14.1.1.1 178
A.10. PEPA 15 178
A.10.1. PEPA Syntax 15.2.1 178
A.10.2. PEPA Operational Semantics 15.2.2 178

Introduction

1. Objectives

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

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



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

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

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

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

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

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

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

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

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

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

We will use these two paradigms for:

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

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

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

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

$$\bullet_p \xrightarrow{a} \bullet_q$$

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

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

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

2. Structure

The course comprises four main parts

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

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

3. References

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

1. Preliminaries

1.1. Inference Rules

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

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

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

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

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

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

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

$$r = \varnothing/y.$$

Equivalent notations are:

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

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

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

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

Example 1.4 (Some inference rules)
The inference rule

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

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

The rule

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

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

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

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

$$d \Vdash_R y$$

where

- *either* $d = \varnothing/y$ *is an axiom of R, i.e.,* $(\varnothing/y) \in R$*;*
- *or* $d = (\{d_1, \dots, d_n\}/y)$ *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:

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

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

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

is a derivation of y in R.

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

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

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

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

then we can build a derivation for y:

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

Summarizing all the above:

- $(\varnothing/y) \Vdash_R y \quad$ if $(\varnothing/y) \in R$ (axiom)
- $(\{d_1, \dots, d_n\}/y) \Vdash_R y \quad$ 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 | (S) | λ
```

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

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

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

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

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

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

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

For example, the rule

$$\frac{\mathtt{)(} \in L_S \qquad \mathtt{((} \in L_S}{\mathtt{)(((} \in L_S}\ (1)$$

is an instance of rule (1): it is obtained by replacing s_1 with $\mathtt{)(}$ and s_2 with $\mathtt{((}$. Of course the string $\mathtt{)(((}$ appearing in the conclusion does not belong to L_S, but the instance is perfectly valid, because it says that "$\mathtt{)(((} \in L_S$ if $\mathtt{)(} \in L_S$ and $\mathtt{((} \in L_S$" and since the premises are false we cannot draw any conclusion.

Let us see an example of valid derivation that uses some valid instances of rules (1) and (2).

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

Reading the proof (from the leaves to the root): Since $\lambda \in L_S$ by axiom (3), then we know that $\mathtt{()} \in L_S$ by (2); if we apply again rule (2) we derive also $\mathtt{(())} \in L_S$ and hence $\mathtt{(())()} \in L_S$ by (1). In other words $\mathtt{(())()} \in L_S$ is a theorem.

Let us introduce a second formalization of the same language (balanced parentheses) without using inference rules. In the following we let a_i denote the ith symbol of the string a. Let

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

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

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

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

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

An example is shown below for $a = \texttt{(())()}$:

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

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

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

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

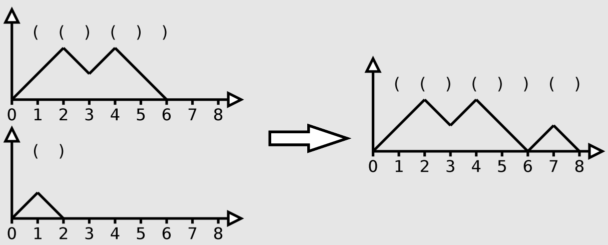
$\Longrightarrow$ *all the strings produced by the grammar satisfy the two properties;*

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

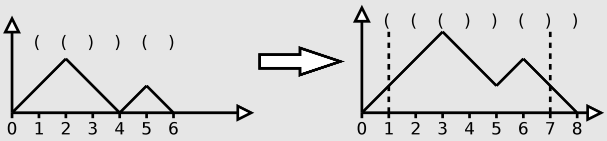
Proof of $\Longrightarrow$) *To show the first implication, we proceed by* induction over the rules*: we assume that the implication is valid for the premises and we show that it holds for the conclusion:*

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

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



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

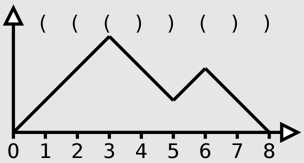


The third rule is just concerned with the empty string that trivially satisfies the two properties.

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

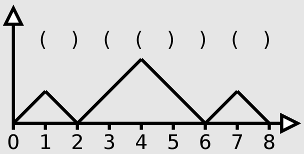
Proof of $\Longleftarrow$) *We need to find a derivation for any string that satisfies the two properties. Let a be such a generic string. (We only sketch this direction of the proof, that goes by induction over the length of the string a.) We proceed by case analysis, considering three cases:*

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



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

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



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

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

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

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

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

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

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

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

- *Rule (3) cannot be applied.*

We show below a successful goal-oriented derivation.

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

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

1.2. Logic Programming

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

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

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

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

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

Definition 1.12 (Horn clause)
A Horn clause *is written* $l\colon -r$ *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:

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

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

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

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

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

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

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

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

We write the derivation described above using the notation

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

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

The derivation can then be completed as follows:

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

By composing the computed substitutions we get

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

This gives us a proof of the theorem

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

Part I.

IMP language

2. Operational Semantics of IMP

2.1. Syntax of IMP

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

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

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

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

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

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 $x, y, x', y', x_0, y_0, x_1, y_1, ...$

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

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

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

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

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

The following productions define the syntax of IMP:

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

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

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

2.1.1. Arithmetic Expressions

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

2.1.2. Boolean Expressions

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

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)

$$\begin{array}{rccccl}
(\mathbf{while} & b & \mathbf{do} & c_1) & ; & c_2 & \textit{is a valid command} \\
\mathbf{while} & b & \mathbf{do} & (c_1 & ; & c_2) & \textit{is a valid command} \\
\mathbf{while} & b & \mathbf{do} & c_1 & ; & c_2 & \textit{is not a valid command, because it is ambiguous}
\end{array}$$

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

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

2.2. Operational Semantics of IMP

2.2.1. Memory State

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

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

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.

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

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

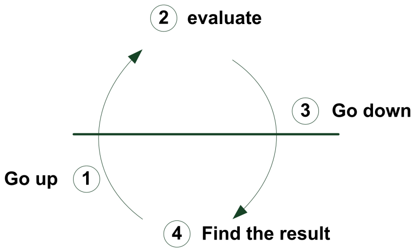
$$\frac{}{\langle x, \sigma\rangle \rightarrow \sigma(x)}\ (\text{ide}) \tag{2.2}$$

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

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

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.

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

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

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

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

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

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

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

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

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

Next, we move to the inference rules for commands.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.3. Examples

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

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

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

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

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* $y \ge 0$*) and stores it in x. In fact, by exploiting the well-known equalities* $0^2 = 0$ *and* $(n+1)^2 = n^2 + 2n + 1$, *the value of* y^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.)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Example 2.8 (Non termination)
Let us consider the command

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

Given σ, the only possible derivation goes as follows:

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

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

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

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

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

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

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:*

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

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:*

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

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:

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

It is immediate to prove that

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

Hence σ and σ' can differ only for the value stored in x. In particular, if $\sigma(x) = 0$ then $\sigma' = \sigma$.

The evaluation of c_1 in σ depends on $\sigma(x)$: if $\sigma(x) = 0$ we must apply the rule 2.18 (whff)*, otherwise the rule 2.17* (whtt) *must be applied. Since we do not know the value of $\sigma(x)$, we consider the two cases separately.*

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

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

($\sigma(x) = 0$)

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

Finally, we observe the following:

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

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

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

2.3.2. Examples: Parametric Equivalence Proofs

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

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

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

$$\begin{aligned}
c_1 &= \mathbf{while}\, b\, \mathbf{do}\, c \\
c_2 &= \mathbf{if}\, b\, \mathbf{then}(c; \mathbf{while}\, b\, \mathbf{do}\, c)\, \mathbf{else\ skip} = \mathbf{if}\, b\, \mathbf{then}(c; c_1)\, \mathbf{else\ skip}
\end{aligned}$$

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

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

($\langle b,\sigma\rangle \to$ false)

$$\begin{array}{ll} \langle \mathbf{while}\, b\, \mathbf{do}\, c,\sigma\rangle \to \sigma' & \nwarrow \quad \langle b,\sigma\rangle \to \mathbf{false} \\ & \nwarrow_{\sigma=\sigma'} \quad \square \end{array}$$

$$\begin{array}{ll} \langle \mathbf{if}\, b\, \mathbf{then}(c;c_1)\, \mathbf{else\ skip},\sigma\rangle \to \sigma' & \nwarrow \quad \langle b,\sigma\rangle \to \mathbf{false} \quad \langle \mathbf{skip},\sigma\rangle \to \sigma' \\ & \nwarrow_{\sigma'=\sigma} \quad \square \end{array}$$

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

($\langle b,\sigma\rangle \to$ true)

$$\begin{array}{ll} \langle \mathbf{while}\, b\, \mathbf{do}\, c,\sigma\rangle \to \sigma' & \nwarrow \quad \langle b,\sigma\rangle \to \mathbf{true} \quad \langle c,\sigma\rangle \to \sigma'' \quad \langle c_1,\sigma''\rangle \to \sigma' \\ & \nwarrow \quad \langle c,\sigma\rangle \to \sigma'' \quad \langle c_1,\sigma''\rangle \to \sigma' \end{array}$$

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*:*

$$\begin{array}{ll} \langle \mathbf{if}\, b\, \mathbf{then}(c;\mathbf{while}\, b\, \mathbf{do}\, c)\, \mathbf{else\ skip},\sigma\rangle \to \sigma' & \nwarrow \quad \langle b,\sigma\rangle \to \mathbf{true} \quad \langle c;\mathbf{while}\, b\, \mathbf{do}\, c,\sigma\rangle \to \sigma' \\ & \nwarrow \quad \langle c;\mathbf{while}\, b\, \mathbf{do}\, c,\sigma\rangle \to \sigma' \\ & \nwarrow \quad \langle c,\sigma\rangle \to \sigma'' \quad \langle c_1,\sigma''\rangle \to \sigma' \end{array}$$

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

Summing up the two cases, and since there 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 convenient methods for proving the equivalence of **while** commands, whose evaluation may diverge.

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

$$\begin{aligned} c_1 &= \mathbf{while}\, b\, \mathbf{do}\, c \\ c_2 &= \mathbf{if}\, b\, \mathbf{then}\, c_1\, \mathbf{else\ skip} \end{aligned}$$

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

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

$$\begin{array}{lll}\langle \mathbf{if}\ b\ \mathbf{then}\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{else}\ \mathbf{skip}, \sigma\rangle \to \sigma' & \nwarrow & \langle b,\sigma\rangle \to \mathbf{true} \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \to \sigma' \\ & \nwarrow & \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \to \sigma'\end{array}$$

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

2.3.3. Inequality Proofs

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

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

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

Let us prove that $c_1 \nsim c_2$.
We focus on the first part of c_1. Assuming $\sigma(x) \leq 0$ it is easy to check that

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

The derivation is sketched below:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Since we can find pairs (σ, σ') such that c_1 loops and c_2 terminates we have that $c_1 \nsim c_2$.

Note that in disproving the equivalence we have exploited a standard technique in logic: to show that a universally quantified formula is not valid we can exhibit one counterexample. Formally:

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

2.3.4. Diverging Computations

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

Let us rephrase the definition of equivalence between commands:

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

Next we see an example where this situation emerges.

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

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

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 $x > 0$:

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

$$\begin{array}{ccc} \langle c_1, \sigma\rangle \to \sigma' & \nwarrow_{\sigma'=\sigma} & \langle x > 0, \sigma\rangle \to \textbf{false} \\ & \nwarrow & \square \end{array}$$

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.*

($\sigma(x) > 0$**)**

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

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

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

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

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

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

Let us consider the command $w = \mathbf{while}\, b\, \mathbf{do}\, c$. 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:

$$\frac{\sigma \in S \quad \forall \sigma' \in S.(\langle c,\sigma'\rangle \to \sigma'' \implies \sigma'' \in S) \quad \forall \sigma' \in S.(\langle b,\sigma'\rangle \to \mathbf{true})}{\langle w,\sigma\rangle \not\to} \tag{2.19}$$

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

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

is satisfied even when $\langle c, \sigma' \rangle \not\to$, as the left-hand side of the implication is false and therefore the implication is true.

3. Induction and Recursion

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

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

3.1. Noether Principle of Well-founded Induction

3.1.1. Well-founded Relations

We recall some key mathematical notions and definitions.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

leading to a contradiction. □

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

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

Proof. We need to prove that:

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

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

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

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

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

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

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

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

Lemma 3.11 (Well-founded relation)
Let $\prec$ be a relation over the set A. The relation $\prec$ is well-founded if and only if every nonempty subset $Q \subseteq A$ contains a minimal element m.

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

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

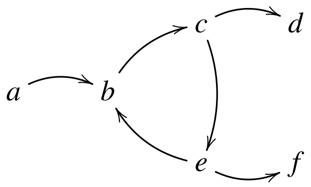


Figure 3.1.: Graph of a relation

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

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

□

Example 3.12 (Natural numbers)
Both $n \prec n+1$ and $n \prec n+1+k$, with n and k in the set ω of natural numbers, are simple examples of well-founded relations. In fact, from every element $n \in \omega$ we can start a descending chain of at most length n.

Definition 3.13 (Terms over sorted signatures)
Let

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

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

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

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

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

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

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:

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

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

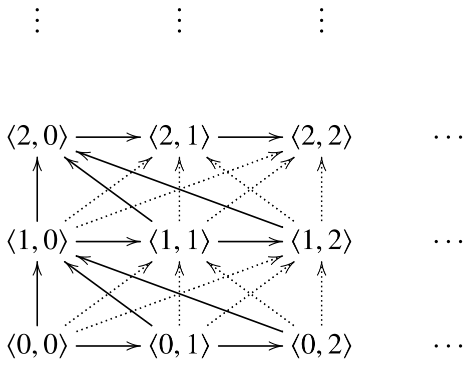


Figure 3.2.: Graph of the lexicographic order relation over pairs of natural numbers.

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

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:

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

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

Example 3.15 (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* $t_1, \ldots, t_n$*, we let:*

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

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

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

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

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

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):*

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

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

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

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

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

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

is a (the only) minimal element of Q*. By Lemma 3.11 the relation is well-founded.*

Example 3.17 (A counterample: Integer numbers)
The usual "strictly less than" relation $<$ *over the set of integer numbers* $\mathbb{N}$ is not well-founded. *In fact it is immediate to define infinite descending chains, such as:*

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

3.1.2. Noether Induction

Theorem 3.18
Let $\prec$ *be a well-founded relation over the set* A *and let* P *be a unary predicate over* A*. Then:*

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

Proof. We show the two implications separately.

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

(⇒): We proceed by contradiction by assuming $\neg(\forall a \in A. P(a))$, i.e., that $\exists a \in A.\ \neg P(a)$. Let us consider the nonempty set $Q = \{\ a \in A \ \mid\ \neg P(a)\ \}$ of all those elements a in A for which $P(a)$ is false. Since $\prec$ is well-founded, we know by Lemma 3.11 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 $\forall b \prec m. b \notin Q$, i.e., $\forall b \prec m. P(b)$. But this leads to a contradiction, because by hypothesis we have $\forall a \in A. (\forall b \prec a. P(b)) \rightarrow P(a)$ and instead the predicate $(\forall b \prec m. P(b)) \rightarrow P(m)$ is false.

Therefore Q must be empty and $\forall a \in A.P(a)$ must hold.

□

3.1.3. Weak Mathematical Induction

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

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

Theorem 3.19 (Weak mathematical induction)

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

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

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

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

3.1.4. Strong Mathematical Induction

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

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

Theorem 3.20 (Strong mathematical induction)

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

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

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

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

3.1.5. Structural Induction

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

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

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

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

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

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

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

Example 3.22
Let us consider the grammar of IMP arithmetic expressions:

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

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

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

In other words, we want to show that given any arithmetic expression a and any memory σ the evaluation of a in σ will always return one and only one value. We 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* $d' \prec d$*, if and only if* d *has the form* $(\{d_1, ..., d_n\} \ / \ y)$ *with* $d_1 \Vdash_R x_1, ..., d_n \Vdash_R x_n$ *and* $(\{x_1, ..., x_n\} \ / \ y) \in R$ *(i.e.* $d \Vdash_R y$*) and* $d' = d_i$ *for some* $1 \leq i \leq n$.

Example 3.25 (Immediate (sub-)derivation)
Let us consider the derivation

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

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* $d' \prec^+ d$.

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

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

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

where $d_1, \ldots, d_n$ are derivation for $x_1, \ldots, x_n$.

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 $(\{x_1, \ldots, x_n\}/y)$ otherwise. Given a set R of such rules, the *set of theorems of* R is defined as

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

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

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

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

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

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

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*

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

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":

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

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

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

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

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

(rule assign): *assuming*

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

we want to show that

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

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

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

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

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

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

We have:

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

But now we can apply the first inductive hypotheses:

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

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

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

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

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

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

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

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

But then, exploiting the inductive hypothesis

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

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

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

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

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

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

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

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

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

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

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

But now we can apply the first inductive hypotheses:

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

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

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

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

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

Let us write the definition in a slightly different way

$$\begin{array}{l} p_y(0) = 0 \\ p_y(x+1) = y + p_y(x) \end{array}$$

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 $\prec \subseteq B \times B$, *the set of* predecessors *of a set* $I \subseteq B$ *is the set*

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

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

Theorem 3.30 (Well-founded recursion)
Let $(B, \prec)$ *a well-founded relation over* B. *Let us consider a function* F *with* $F(b,h) \in C$, *where*

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

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

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

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

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

Example 3.31 (Primitive recursion)
Let us recast the Peano formula seen above (Example 3.28) to the formal scheme of primitive recursion.

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

Example 3.32 (Structural recursion)
Let us consider the signature Σ *for binary trees* $B = T_\Sigma$, *where* $\Sigma_0 = \{0, 1, ...\}$ *and* $\Sigma_2 = \mathsf{cons}$. *Take the*

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

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

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

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

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

or more formally

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

Example 3.33 (Ackermann function)
The Ackermann function $ack(z, x, y) = ack_y(z, x)$ *is defined by well-founded recursion (exploiting the lexicographic order over pair of natural numbers) by letting*

$$\left\{\begin{array}{lcccccccl} ack(& 0 & , & 0 & , & y &) = & y \\ ack(& 0 & , & x+1 & , & y &) = & ack(0, x, y) + 1 \\ ack(& 1 & , & 0 & , & y &) = & 0 \\ ack(& z+2 & , & 0 & , & y &) = & 1 \\ ack(& z+1 & , & x+1 & , & y &) = & ack(z, ack(z+1, x, y), y) \end{array}\right.$$

We have

$$\left\{\begin{array}{rcl} ack(0, 0, y) & = & y \\ ack(0, x+1, y) & = & ack(0, x, y) + 1 \end{array}\right. \quad \Longrightarrow \quad ack(0, x, y) = y + x$$

$$\left\{\begin{array}{rcl} ack(1, 0, y) & = & 0 \\ ack(1, x+1, y) & = & ack(0, ack(1, x, y), y) = ack(1, x, y) + y \end{array}\right. \quad \Longrightarrow \quad ack(1, x, y) = yx$$

$$\left\{\begin{array}{rcl} ack(2, 0, y) & = & 1 \\ ack(2, x+1, y) & = & ack(1, ack(2, x, y), y) = ack(2, x, y)y \end{array}\right. \quad \Longrightarrow \quad ack(2, x, y) = y^x$$

and so on.

4. Partial Orders and Fixpoints

4.1. Orderings and Continuous Functions

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

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

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

4.1.1. Orderings

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1.2. Hasse Diagrams

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

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

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

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

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

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

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.

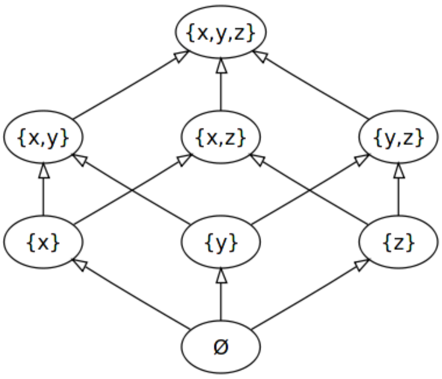


Figure 4.1.: Hasse diagram for the powerset over $\{x, y, z\}$ ordered by inclusion

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 $\sqsubseteq$:

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

We have:

$$H^* = \sqsubseteq$$

Note that the rule

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

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

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

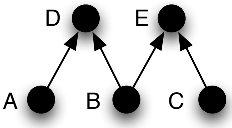
The above theorem only allows to represent finite orders.

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

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

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

Let us consider the following order:



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

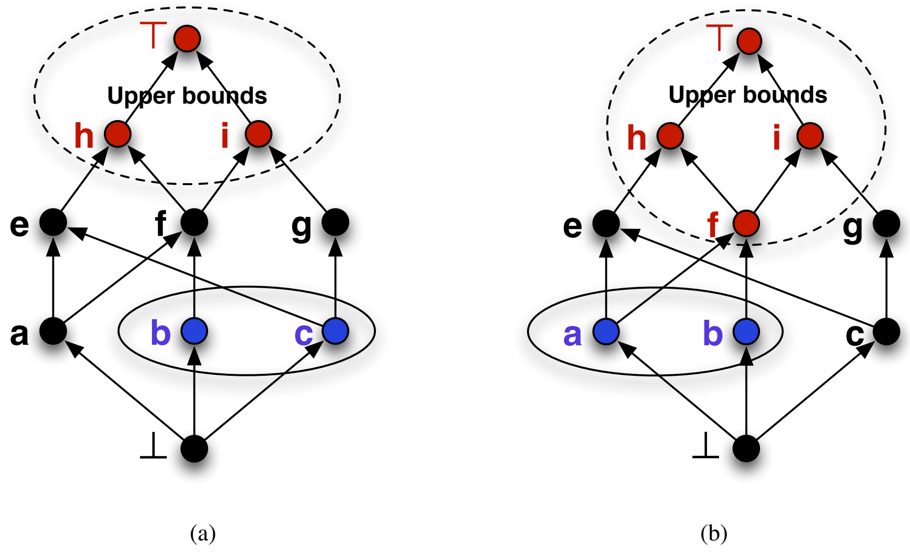


Figure 4.2.: Two subsets of a poset, the right one with LUB and the left one without LUB

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

4.1.3. Chains

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

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

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

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

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

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

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

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

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

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

4.1.4. Complete Partial Orders

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

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

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

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

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

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

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

$$(\omega, \leq)$$

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

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

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

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

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

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

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

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

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:

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

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 $R_0 \subseteq R_1 \subseteq R_2 \subseteq \ldots$ with $\forall i.\ nR_im \wedge nR_im' \implies m = m'$ 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 $\exists k, k'.nR_km \wedge nR_{k'}m'$, we take $k'' = max\{k, k'\}$ then it holds $nR_{k''}m \wedge nR_{k''}m'$ then by hypothesis $m = m'$. □

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

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

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

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

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

Example 4.28 (Limit of a chain of partial functions)

Let $\{f_i\}_{i\in\omega}$ be a chain on P' such that:

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

Let us consider some application of the functions:

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

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

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

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

4.2. Continuity and Fixpoints

4.2.1. Monotone and Continuous Functions

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

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

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

We 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:*

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

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

We will say that a continuous function 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

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

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 gfp(f) the greatest fixpoint of f and with 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 $f : D \to D$ 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:

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

Now we have a new chain :

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

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 $\mathrm{lfp}(Id)$ *we need a bottom element in the CPO.*

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

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

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

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

Note that f has at least one fixpoint, indeed:

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

But nothing ensures that the fixpoint is reachable.

4.3. Immediate Consequence Operator

4.3.1. The $\hat{R}$ Operator

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

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

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

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

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

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

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

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

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

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

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

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

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

1. Let y be an element of $\bigcup_{n\in\omega} \hat{R}(B_n)$ so there exists a natural number m such that $y \in \hat{R}(B_m)$. Since $B_m \subseteq \bigcup_{n\in\omega} B_n$ by monotonicity $\hat{R}(B_m) \subseteq \hat{R}(\bigcup_{n\in\omega} B_n)$ so $y \in \hat{R}(\bigcup_{n\in\omega} B_n)$.
2. Let y be an element of $\hat{R}(\bigcup_{n\in\omega} B_n)$ so there exists a rule X/y with $X \subseteq \bigcup_{n\in\omega} B_n$.
   Since X is finite there exists a natural number k such that $X \subseteq B_k$, otherwise $X \nsubseteq \bigcup_{n\in\omega} B_n$. In fact if $X = \{x_i\}_{i=1,\ldots,N}$, then $k_i = \min\{j \mid x_i \in B_j\}$ and $k = max\{k_i\}_{i=1,\ldots N}$. So $y \in \hat{R}(B_k)$ then $y \in \bigcup_{n\in\omega} \hat{R}(B_n)$ as required.

□

4.3.2. Fixpoint of $\hat{R}$

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

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

where

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

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

Theorem 4.41
Let R a rule system, it holds:

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

Proof. By induction on n

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

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

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

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

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

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

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

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

as required.

□

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

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

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

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

We have :

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

Then we have:

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

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

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

$$fix(\hat{R}) = \bigcup_{n\in\omega} R^n(\emptyset) = \{P(1), P(2), P(3), P(4), \ldots\}$$
$$\hat{R}(fix(\hat{R})) = \{P(0), P(1), P(2), P(3), P(4), \ldots\}$$

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

Example 4.44 (String)
Let us consider the grammar

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

Obviously using the inference rule formalism we can write:

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

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

$$\begin{aligned}
S_0 &= \hat{R}^0(\emptyset) = \emptyset \\
S_1 &= \hat{R}(S_0) = \lambda + (\emptyset) + \emptyset\emptyset = \{\lambda\} \\
S_2 &= \hat{R}(S_1) = \lambda + (\lambda) + \lambda\lambda = \{\lambda,\ (\,)\} \\
S_3 &= \hat{R}(S_2) = \lambda + (\lambda) + ((\,)) + (\,)(\,) = \{\lambda,\ (\,),\ ((\,)),\ (\,)(\,)\} \\
&\dots
\end{aligned}$$

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

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:

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

Where x is a variable.

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

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

All the notions used in this definition, like "True" and "False" can be formalized in lambda notation only by using lambda abstraction, but this development is beyond the scope of the course.

Definition 5.2
Let t, t' and t'' be three lambda terms, we define:

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

Lambda abstraction $\lambda x.t$ is the main feature. It allows to define functions, where x represents the parameter of the function and t is the lambda term which represents the body of the function. For example the term $\lambda x.x$ is the identity function. Usually we equip the lambda notation with some equations. Before introducing these equations we will present the notions of *substitution* and *free variable*. Substitution allows to systematically substitute a lambda term for a variable.

Definition 5.3
Let t and t' be two lambda terms, we define:

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

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:

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

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.

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

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

5.2.1. Function $\mathscr{A}$

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

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

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

Notation
We will rely on definitions of the form

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

with the following meaning:

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

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

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

instead of

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

In this way, we simplify the notation in the right-hand side. Notice that 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:

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

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

5.2.2. Function $\mathscr{B}$

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

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

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

5.2.3. Function $\mathscr{C}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Obviously this definition is not a structural recursion definition rule. Indeed structural recursion allows only for subterms, while here “while b do c” appears on both sides. To solve this issue we will reduce the problem of defining the semantics of “while” to a fixpoint calculation. Let us define a function $\Gamma_{b,c}$.

$$\Gamma_{b,c} = \underbrace{\lambda\varphi.\,\underbrace{\lambda\sigma.\,\underbrace{\mathscr{B}[\![b]\!]\,\sigma \to \varphi^*(\mathscr{C}[\![c]\!]\,\sigma),\sigma}_{\Sigma_\perp}}_{\Sigma\to\Sigma_\perp}}_{(\Sigma\to\Sigma_\perp)\to\Sigma\to\Sigma_\perp}$$

As we can see the function $\Gamma_{b,c}$ is of type $(\Sigma \to \Sigma_\perp) \to \Sigma \to \Sigma_\perp$ and contains only subterms of “**while** b **do** c”. We want to define the semantics of the “while” command as the least fixpoint of $\Gamma_{b,c}$. Now we will show that $\Gamma_{b,c}$ is a monotone continuous function, so that we can prove that $\Gamma_{b,c}$ has a least fixpoint and that:

$$\mathscr{C}[\![\textbf{while}\ b\ \textbf{do}\ c]\!] = fix\,\Gamma_{b,c} = \bigsqcup_{n\in\omega} \Gamma^n_{b,c}(\perp_{\Sigma\to\Sigma_\perp})$$

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

$$R_{\Gamma_{b,c}} = \left\{ \frac{\mathscr{B}[\![b]\!]\,\sigma \quad \mathscr{C}[\![c]\!]\,\sigma = \sigma'' \quad \sigma'' \to \sigma'}{\sigma \to \sigma'}, \qquad \frac{\neg\mathscr{B}[\![b]\!]\,\sigma}{\sigma \to \sigma} \right\} \text{ with } \hat{R}_{\Gamma_{b,c}} = \Gamma_{b,c}$$

Notice that here the only well formed formulas are $\sigma'' \longrightarrow \sigma'$, $\sigma \longrightarrow \sigma'$ and $\sigma \longrightarrow \sigma$. An instance of the first rule schema is obtained by picking up a value of σ such that $\mathscr{B}[\![b]\!]\sigma$ is true, and a (the) value of σ'' such that $\mathscr{C}[\![c]\!]\sigma = \sigma''$. Then for every σ'' such that $\sigma'' \to \sigma'$ we can imply $\sigma \to \sigma'$. Similarly for the second rule schema.

Since all the rules obtained in this way have a finite number of premises (actually one or none), we can apply th.4.40, which ensures the continuity of $\hat{R}_{\Gamma_{b,c}}$. Now by using th.4.3.2 we have:

$$fix\,\hat{R}_{\Gamma_{b,c}} = I_{R_{\Gamma_{b,c}}} = fix\,\Gamma_{b,c}$$

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:

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

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

$$\Gamma_{\textbf{true},\textbf{skip}}\varphi\sigma \;=\; \mathscr{B}[\![\textbf{true}]\!]\,\sigma \to \varphi^*\,(\mathscr{C}[\![\textbf{skip}]\!]\,\sigma)\,,\sigma \;=\; \varphi^*\,(\mathscr{C}[\![\textbf{skip}]\!]\,\sigma) \;=\; \varphi^*\sigma \;=\; \varphi\sigma$$

So we have $\Gamma_{\textbf{true},\textbf{skip}}\varphi = \varphi$, *that is* $\Gamma_{\textbf{true},\textbf{skip}}$ *is the identity function. Then each function* φ *is a fixpoint of* $\Gamma_{\textbf{true},\textbf{skip}}$, *but we are looking for the least fixpoint. This means that the sought solution is the least function in the* $CPO_\perp$ *of functions. Then we have* $fix\,\Gamma_{\textbf{true},\textbf{skip}} = \lambda\sigma.\perp_{\Sigma_\perp}$.

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 $w =$ **while** b **do** c *and* **if** b **then** $c; w$ **else skip** *. Since* $\mathscr{C}\llbracket w \rrbracket$ *is a fixpoint we have:*

$$\mathscr{C}\llbracket w \rrbracket = \Gamma(\mathscr{C}\llbracket w \rrbracket) = \lambda\sigma.\mathscr{B}\llbracket\, b \,\rrbracket\sigma \rightarrow \mathscr{C}\llbracket w \rrbracket^{*}(\mathscr{C}\llbracket c \rrbracket\sigma), \sigma$$

Moreover we have:

$$\begin{aligned} \mathscr{C}\llbracket \textbf{if } b \textbf{ then } c; w \textbf{ else skip} \rrbracket &= \lambda\sigma.\mathscr{B}\llbracket\, b \,\rrbracket\sigma \rightarrow \mathscr{C}\llbracket c;w \rrbracket\sigma, \mathscr{C}\llbracket \textbf{skip} \rrbracket\sigma \\ &= \lambda\sigma.\mathscr{B}\llbracket\, b \,\rrbracket\sigma \rightarrow \mathscr{C}\llbracket w \rrbracket^{*}(\mathscr{C}\llbracket c \rrbracket\sigma), \sigma \end{aligned}$$

Example 5.12

Let us consider the command:

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

we have:

$$\mathscr{C}\llbracket c \rrbracket = fix\,\Gamma$$

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

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

Now we have the followings possibilities:

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

So we have:

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

We can now attempt to formulate a conjecture:

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

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

The base case is trivial, indeed $\varphi_0 = \Gamma^0\ \bot = \lambda\,\sigma.\bot_{\Sigma_\bot}$.

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

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

and prove:

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

By definition:

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

we have:

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

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

Then:

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

Finally we have:

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

5.3. Equivalence Between Operational and Denotational Semantics

This section deals with the issue of equivalence between the two semantics of IMP introduced up to now. As we will show the denotational and operational semantics agree. As usual we will handle first arithmetic and boolean expressions, then assuming the proved equivalences we will show that operational and denotational semantics agree also on commands.

5.3.1. Equivalence Proofs for $\mathscr{A}$ and $\mathscr{B}$

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

$$P(a) \overset{\text{def}}{=} \langle a,\sigma\rangle \to \mathscr{A}\llbracket a\rrbracket\sigma \quad \forall a \in Aexpr$$

That is, the results of evaluating an arithmetic expression both by operational and denotational semantics are the same. If we regard the operational semantics as an interpreter and the denotational semantics as a compiler we are proving that interpreting an IMP program and executing its compiled version starting from the same memory leads to the same result. The proof is by structural induction on the arithmetic expressions.

Constants $P(n) \overset{\text{def}}{=} \langle n,\sigma\rangle \to \mathscr{A}\llbracket n\rrbracket\sigma$ Since $\mathscr{A}\llbracket n\rrbracket\sigma = n$ and $\langle n,\sigma\rangle \to n$.

Variables $P(x) \overset{\text{def}}{=} \langle x,\sigma\rangle \to \mathscr{A}\llbracket x\rrbracket\sigma$ Since $\mathscr{A}\llbracket x\rrbracket\sigma = \sigma(x)$ and $\langle x,\sigma\rangle \to \sigma(x)$.

Binary operators Let us generalize the proof for the binary operations of arithmetic expressions. Consider two arithmetic expressions a_0 and a_1 and a binary operator $\circ$ of IMP, whose corresponding semantic operator is $\odot$. We would like to prove

$$P(a_0) \wedge P(a_1) \Rightarrow P(a_0 \circ a_1)$$

So as usual we assume $P(a_0) \overset{\text{def}}{=} \langle a_0,\sigma\rangle \to \mathscr{A}\llbracket a_0\rrbracket\sigma$ and $P(a_1) \overset{\text{def}}{=} \langle a_1,\sigma\rangle \to \mathscr{A}\llbracket a_1\rrbracket\sigma$ and we prove $P(a_0 \circ a_1) \overset{\text{def}}{=} \langle a_0 \circ a_1,\sigma\rangle \to \mathscr{A}\llbracket a_0 \circ a_1\rrbracket\sigma$.
We have:

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

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 $\mathscr{C}$

Central to the proof of equivalence between denotational and operational semantics is the case of commands. Operational and denotational semantics are defined in very different 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:

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

As usual we will divide the proof in two parts:

- $P\left(\langle c,\sigma\rangle \to \sigma'\right) \overset{\text{def}}{=} \mathscr{C}\llbracket c\rrbracket\sigma = \sigma'$ (Completeness)
- $P\left(c\right) \overset{\text{def}}{=} \forall \sigma,\sigma'.\ \mathscr{C}\llbracket c\rrbracket\sigma = \sigma' \Rightarrow \langle c,\sigma\rangle \to \sigma'$ (Correctness)

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

5.3.2.1. Completeness of the Denotational Semantics

Let us prove the first part of the theorem: $P(\langle c, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}\,[\![c]\!]\,\sigma = \sigma'$.
Since the operational semantic is defined by a rule system we will proceed by rule induction. So for each rule we will assume the property on the premises and we will prove the property on the consequence.

Skip We would like to prove:

- $P(\langle \mathbf{skip}, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \mathscr{C}\,[\![\mathbf{skip}]\!]\,\sigma = \sigma$

Obviously the proposition is true by definition of denotational semantic.

Assignment Let us consider the assignment command:

$$\frac{\langle a, \sigma\rangle \to m}{\langle x := a, \sigma\rangle \to \sigma\left[{}^{m}/_{x}\right]}$$

We assume:

- $P(\langle a, \sigma\rangle \to m) \stackrel{\text{def}}{=} \mathscr{A}\,[\![a]\!]\,\sigma = m$.

We will prove:

- $P(\langle x := a, \sigma\rangle \to \sigma\left[{}^{m}/_{x}\right]) \stackrel{\text{def}}{=} \mathscr{C}\,[\![x := a]\!]\,\sigma = \sigma\left[{}^{m}/_{x}\right]$

We have by the definition of denotational semantics:

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

Concatenation Let us consider the concatenation rule:

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

We assume:

- $P(\langle c_0, \sigma\rangle \to \sigma'') \stackrel{\text{def}}{=} \mathscr{C}\,[\![c_0]\!]\,\sigma = \sigma''$
- $P(\langle c_1, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}\,[\![c_1]\!]\,\sigma'' = \sigma'$.

We will prove:

- $P(\langle c_0; c_1, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}\,[\![c_0; c_1]\!]\,\sigma = \sigma'$

We have by definition:

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

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

Conditional For the conditional command we have two rules:

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

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

- $P(\langle b, \sigma\rangle \to \mathbf{true}) \stackrel{\text{def}}{=} \mathscr{B}\,[\![b]\!]\,\sigma = \mathbf{true}$
- $P(\langle c_0, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}\,[\![c_0]\!]\,\sigma = \sigma'$.

And we will prove:

- $P(\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}\,[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]\,\sigma = \sigma'$

We have:

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

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

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

Let us consider the first rule. We assume:

- $P(\langle b,\sigma\rangle \to \mathbf{false}) \stackrel{\text{def}}{=} \mathscr{B}\,[\![b]\!]\,\sigma = \mathbf{false}$

We will prove:

- $P\,(\langle \mathbf{while}\, b\, \mathbf{do}\, c,\sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \mathscr{C}\,[\![\mathbf{while}\, b\, \mathbf{do}\, c]\!]\,\sigma = \sigma$

We have by the fixpoint property of the denotational semantics:

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

For the second rule we assume:

- $P(\langle b,\sigma\rangle \to \mathbf{true}) \stackrel{\text{def}}{=} \mathscr{B}\,[\![b]\!]\,\sigma = \mathbf{true}$
- $P(\langle c,\sigma\rangle \to \sigma'') \stackrel{\text{def}}{=} \mathscr{C}\,[\![c]\!]\,\sigma = \sigma''$
- $P(\langle \mathbf{while}\, b\, \mathbf{do}\, c,\sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}\,[\![\mathbf{while}\, b\, \mathbf{do}\, c]\!]\,\sigma'' = \sigma'$

We will prove:

- $P(\langle \mathbf{while}\, b\, \mathbf{do}\, c,\sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}\,[\![\mathbf{while}\, b\, \mathbf{do}\, c]\!]\,\sigma = \sigma'$

By the definition of the denotational semantics:

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

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

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:

$$P(c) \stackrel{\text{def}}{=} \forall \sigma, \sigma'\ \mathscr{C}\,[\![c]\!]\,\sigma = \sigma' \Rightarrow \langle c,\sigma\rangle \to \sigma'$$

Skip We will prove:

- $P(\mathbf{skip}) \stackrel{\text{def}}{=} \forall \sigma, \sigma'\ \mathscr{C}\,[\![\mathbf{skip}]\!]\,\sigma = \sigma' \Rightarrow \langle \mathbf{skip},\sigma\rangle \to \sigma'$

By definition we have $\mathscr{C}\,[\![\mathbf{skip}]\!]\,\sigma = \sigma$ and $\langle \mathbf{skip},\sigma\rangle \to \sigma$.

Assignment We will prove:

- $P(x := a) \stackrel{\text{def}}{=} \forall \sigma, \sigma'\ \mathscr{C}\,[\![x := a]\!]\,\sigma = \sigma' \Rightarrow \langle x := a,\sigma\rangle \to \sigma'$

By definition have $\sigma' = \sigma[^n/_x]$ where $\mathscr{A}\,[\![a]\!]\,\sigma = n$ so we have $\langle a,\sigma\rangle \to n$ and thus we can apply the rule:

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

Concatenation We will prove:

- $P(c_0;c_1) \overset{\text{def}}{=} \forall\sigma,\sigma'\ \mathscr{C}\,[\![c_0;c_1]\!]\,\sigma = \sigma' \Rightarrow \langle c_0;c_1,\sigma\rangle \to \sigma'$

We assume:

- $P(c_0) \overset{\text{def}}{=} \forall\sigma,\sigma''\ \mathscr{C}\,[\![c_0]\!]\,\sigma = \sigma'' \Rightarrow \langle c_0,\sigma\rangle \to \sigma''$
- $P(c_1) \overset{\text{def}}{=} \forall\sigma'',\sigma'\ \mathscr{C}\,[\![c_1]\!]\,\sigma'' = \sigma' \Rightarrow \langle c_1,\sigma\rangle \to \sigma'$

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

$$\frac{\langle c_0,\sigma\rangle \to \sigma'' \qquad \langle c_1,\sigma''\rangle \to \sigma'}{\langle c_0;c_1,\sigma\rangle \to \sigma'}$$

Conditional We will prove:

- $P(\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1) \overset{\text{def}}{=} \forall\sigma,\sigma'\ \mathscr{C}\,[\![\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1]\!]\,\sigma = \sigma' \Rightarrow \langle\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1,\sigma\rangle \to \sigma'$

As usual we must distinguish two cases, let us consider only the case with $\mathscr{B}\,[\![b]\!]\,\sigma = \textbf{false}$, namely $\langle b,\sigma\rangle \to \textbf{false}$
We assume:

- $P(c_0) \overset{\text{def}}{=} \forall\sigma,\sigma'\ \mathscr{C}\,[\![c_0]\!]\,\sigma = \sigma' \Rightarrow \langle c_0,\sigma\rangle \to \sigma'$
- $P(c_1) \overset{\text{def}}{=} \forall\sigma,\sigma'\ \mathscr{C}\,[\![c_1]\!]\,\sigma = \sigma' \Rightarrow \langle c_1,\sigma\rangle \to \sigma'$
- $\mathscr{C}\,[\![\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1]\!]\,\sigma = \sigma'$

By definition and since $\mathscr{B}\,[\![b]\!]\,\sigma = \textbf{false}$ we have $\mathscr{C}\,[\![\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1]\!]\,\sigma = \mathscr{B}\,[\![b]\!]\,\sigma \to \mathscr{C}\,[\![c_0]\!]\,\sigma, \mathscr{C}\,[\![c_1]\!]\,\sigma = \mathscr{C}\,[\![c_1]\!]\,\sigma = \sigma'$ this implies by hypothesis $\langle c_1,\sigma\rangle \to \sigma'$. Thus we can apply the rule:

$$\frac{\langle c_1,\sigma\rangle \to \sigma' \qquad \langle b,\sigma\rangle \to \textbf{false}}{\langle\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1,\sigma\rangle \to \sigma'}$$

While We will prove:

$$P(\textbf{while}\ b\ \textbf{do}\ c) \overset{\text{def}}{=} \forall\sigma,\sigma'\ \mathscr{C}\,[\![\textbf{while}\ b\ \textbf{do}\ c]\!]\,\sigma = \sigma' \Rightarrow \langle\textbf{while}\ b\ \textbf{do}\ c,\sigma\rangle \to \sigma'$$

We assume by structural induction:

- $P(c) \overset{\text{def}}{=} \forall\sigma,\sigma''\ \mathscr{C}\,[\![c]\!]\,\sigma = \sigma'' \Rightarrow \langle c,\sigma\rangle \to \sigma''$

By definition $\mathscr{C}\,[\![\textbf{while}\ b\ \textbf{do}\ c]\!]\,\sigma = fix\ \Gamma_{b,c}\ \sigma$ so:

$$\begin{aligned}
P(\textbf{while}\ b\ \textbf{do}\ c) &= \mathscr{C}\,[\![\textbf{while}\ b\ \textbf{do}\ c]\!]\,\sigma = \sigma' \Rightarrow \langle\textbf{while}\ b\ \textbf{do}\ c,\sigma\rangle \to \sigma' \\
&= fix\ \Gamma_{b,c}\ \sigma = \sigma' \Rightarrow \langle\textbf{while}\ b\ \textbf{do}\ c,\sigma\rangle \to \sigma' \\
&= \left(\bigsqcup_{n\in\omega} \Gamma^n_{b,c}\bot\right)\sigma = \sigma' \Rightarrow \langle\textbf{while}\ b\ \textbf{do}\ c,\sigma\rangle \to \sigma' \\
&= \forall n.(\Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle\textbf{while}\ b\ \textbf{do}\ c,\sigma\rangle \to \sigma')
\end{aligned}$$

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

P(0) We have to prove:

$$\Gamma^0_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'$$

Obviously:

$$\bot\sigma = \sigma' \Rightarrow \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'$$

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

P(n) $\Rightarrow$ **P(n+1)** Let us assume $A(n) \overset{\text{def}}{=} \Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'$. We will show that $A(n+1) \overset{\text{def}}{=} \Gamma^{n+1}_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'$. We assume the premise of the implication, i.e. $\Gamma_{b,c}\left(\Gamma^n_{b,c}\bot\right)\sigma = \sigma'$, that is

$$\mathscr{B}\,[\![b]\!]\,\sigma \to \left(\Gamma^n_{b,c}\bot\right)^*(\mathscr{C}\,[\![c]\!]\,\sigma)\,, \sigma = \sigma'$$

Now we distinguish two cases $\mathscr{B}\,[\![b]\!]\,\sigma = \textbf{false}$ and $\mathscr{B}\,[\![b]\!]\,\sigma = \textbf{true}$.

- if $\mathscr{B}\,[\![b]\!]\,\sigma = \textbf{false}$, we have $\sigma' = \sigma$. As proved in the previous sections $\mathscr{B}\,[\![b]\!]\,\sigma = \textbf{false} \Leftrightarrow \langle b, \sigma\rangle \to \textbf{false}$. Now by using the rule:

$$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma}$$

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

- if $\mathscr{B}\,[\![b]\!]\,\sigma = \textbf{true}$ as for the false case $\langle b, \sigma\rangle \to \textbf{true}$. The premise of the implication becomes $\sigma' = \left(\Gamma^n_{b,c}\bot\right)^*(\mathscr{C}\,[\![c]\!]\,\sigma)$. We can omit the lifting operator since we are working with terminating commands. So we have $\mathscr{C}\,[\![c]\!]\,\sigma = \sigma''$ and by structural induction $\langle c, \sigma\rangle \to \sigma''$. Now we have by mathematical induction hypothesis $A(n) = \Gamma^n_{b,c}\bot\sigma'' = \sigma' \Rightarrow \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma''\rangle \to \sigma'$. Finally by using the rule:

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \qquad \langle c, \sigma\rangle \to \sigma'' \qquad \langle \textbf{while}\, b\, \textbf{do}\, c, \sigma''\rangle \to \sigma'}{\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'}$$

  we have $\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'$ as required.

5.4. Computational Induction

Up to this time the denotational semantics is less powerful than the operational, since we are not able to prove properties on fixpoints. To fill this gap we introduce *computational induction*, which applies to a class of properties corresponding to inclusive sets.

Definition 5.13 (Inclusive set)
Let $(D, \sqsubseteq)$ be a CPO, let P be a set, we say that P is an inclusive *set if and only if:*

$$(\forall n \in \omega, d_n \in P) \Rightarrow \bigsqcup_{n\in\omega} d_n \in P$$

A property is inclusive if the set of values on which it holds is inclusive.

Notice that if we consider a subset P of D as equipped with the same partial ordering $\sqsubseteq$ of D, then P is inclusive iff it is complete.

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:*

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

is sound.

Proof. Given the second and the third premises, it is easy to prove by mathematical induction that $\forall n.\ F^n(\perp) \in P$. Then also $\bigsqcup_{n\in\omega} F^n(\perp) = \mathrm{fix}(F) \in P$ since P is inclusive. □

Example 5.16 (Computational induction)

$$\mathscr{C}\,[\![\mathbf{while}\ x \neq 0\ \mathbf{do}\ x := x - 1]\!]\,\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]$$

By definition:

$$\mathscr{C}\,[\![w]\!] = fix(\Gamma) \quad \Gamma\varphi\sigma = \sigma x \neq 0 \to \varphi\sigma[^{\sigma x - 1}/_x], \sigma$$

Let us rewrite the property:

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

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

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

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:

$$\frac{P \textit{ inclusive} \quad \perp \in P \quad \forall\varphi.P(\varphi) \Rightarrow P(\Gamma\,\varphi)}{P(\mathscr{C}\,[\![\mathbf{while}\ x \neq 0\ \mathbf{do}\ x := x - 1]\!])}$$

- $P(\perp)$ *obviously since* $\perp\sigma = \sigma'$ *is always false.*
- $P(\varphi) \overset{?}{\Longrightarrow} P(\Gamma\varphi)$
  Let us assume
  $$P(\varphi) = \forall\sigma.(\varphi\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x])$$
  we will show
  $$P(\Gamma\varphi) = \Gamma\varphi\sigma = \sigma' \overset{?}{\Longrightarrow} \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]$$
  we assume the premise $\Gamma\varphi\sigma = \sigma'$
  $$\Gamma\varphi\sigma = \sigma x \neq 0 \to \varphi\sigma[^{\sigma x - 1}/_x], \sigma = \sigma'$$

  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''

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

we have:

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

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

And the consequence holds also in this case. 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.

$$
\begin{array}{lll}
t ::= & x \quad | & \text{Variable} \\
 & n \quad | & \text{Constant} \\
 & t_1 + t_2 \quad | \quad t_1 - t_2 \quad | \quad t_1 \times t_2 \quad | & \text{Arithmetic Operators} \\
 & \mathbf{if}\, t\, \mathbf{then}\, t_1\, \mathbf{else}\, t_2 \quad | & \text{Conditional (it reads } \mathbf{if}\, t = 0\, \mathbf{then}\, t_1\, \mathbf{else}\, t_2\text{)} \\
 & (t_1, t_2) \quad | \quad \mathbf{fst}(t) \quad | \quad \mathbf{snd}(t) \quad | & \text{Pairing and Projection Operators} \\
 & \lambda x.t \quad | \quad (t_1\ t_2) \quad | & \text{Function Abstraction and Application} \\
 & \mathbf{rec}\, x.t & \text{Recursion}
\end{array}
$$

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 $\mathbf{rec}\, x.t$ defines a term t which can contain variable x, which in turn can be replaced by t.

6.1.1. Typed Terms

Using the definition of pre-term given in the previous section, we can construct ill-formed terms that make little sense (for example we can construct terms like $(\lambda x.0) + 1$). To avoid these constructions we introduce the concepts of *type* and *typed term*. A type is a term constructed by using the following grammar:

$$\tau ::= int \ | \ \tau * \tau \ | \ \tau \to \tau$$

So we allow constant type *int*, the pair type and the function type. Using these constructors we are allowed to define infinitely many types, like $(int * int) \to int$ and $int \to (int * (int \to int))$. Now we define the rule system which allows to say if a pre-term of HOFL is well-formed (i.e. if we can associate a type to the pre-term).

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

Arithmetic operators

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

Pairings

$$\frac{t_0 : \tau_0 \quad t_1 : \tau_1}{(t_0, t_1) : \tau_0 * \tau_1} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{fst}(t) : \tau_0} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{snd}(t) : \tau_1}$$

Functions

$$\frac{x : \tau_0 \quad t : \tau_1}{\lambda x.t : \tau_0 \to \tau_1} \qquad \frac{t_1 : \tau_0 \to \tau_1 \qquad t_0 : \tau_0}{(t_1\ t_0) : \tau_1}$$

Recursion

$$\frac{x : \tau \quad t : \tau}{\mathbf{rec}\ x.t : \tau}$$

Definition 6.1 (Well-Formed Terms of HOFL)
Let t be a pre-term of HOFL, we say that t is well-formed *if there exists a type τ such that $t : \tau$.*

In Section 5.1 we defined the concepts of free variables and substitution for λ-calculus. Now we define the same concepts for HOFL. So by structural recursion we define the set of free-variables of HOFL terms as follows:

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

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

$$\begin{aligned}
&x[^t/_x] = t\\
&y[^t/_x] = y \qquad \text{if } x \neq y\\
&n[^t/_x] = n\\
&(t_1 + t_2)[^t/_x] = t_1[^t/_x] + t_2[^t/_x] \qquad \text{(Analogously for } -, \times, \mathbf{if\ then\ else}, \text{pairing}, \mathbf{fst}, \mathbf{snd}\ )\\
&(\lambda y.t')[^t/_x] = \lambda z.t'[^z/_y][^t/_x] \quad z \notin FV(\lambda y.t'), FV(t), \{x\} \qquad \text{(Analogously for recursion)}
\end{aligned}$$

Note that in the last rule we performed an α-conversion before the substitution. This ensures that the variables in t' are not bound by the substitution operation. As discussed in Section 5.1 the substitution is well-defined if we consider the terms up to α-equivalence (i.e. up to the name equivalence). Obviously we would like to extend these concepts to typed terms. So we are interested in understanding how substitution and α-conversion interact with typing. We have the following results:

Theorem 6.2 (Substitution Respects Types)
Let $t : \tau$ a term of type τ and let $t', y : \tau'$ be two terms of type τ' then we have

$$t[^{t'}/_y] : \tau$$

Proof. We leave as an exercise to prove the property 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:
$$P(\lambda x.t) = \forall y, t', \tau_1, \tau_2, \tau'.\ \lambda x.t : \tau_1 \to \tau_2 \wedge y, t' : \tau' \Rightarrow (\lambda x.t)[t'/y] : \tau_1 \to \tau_2$$
By inductive hypothesis we have $x : \tau_1$, $t : \tau_2$ and $t[^z/_x][^{t'}/_y] : \tau_2$, by the above exercise and by the inductive hypothesis. Then $\lambda z.t[^z/_x][^{t'}/_y] = (\lambda x.t)[t'/y] : \tau_1 \to \tau_2$

□

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 t*0 *should behave like the following infinite list:*

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

Let us show that this term is not typable:

$$\begin{aligned}
t = \mathbf{rec}\ \ p.\lambda x.\ (x, (p\ (x+2))) : \tau &\qquad \nwarrow_{\hat{p}=\tau}\\
\lambda x.\ (x, (p\ (x+2))) : \tau &\qquad \nwarrow_{\tau=\tau_1\to\tau_2}\\
x : \tau_1,\ (x, (p\ (x+2))) : \tau_2 &\qquad \nwarrow_{\hat{x}=\tau_1}\\
(x, (p\ (x+2))) : \tau_2 &\qquad \nwarrow_{\tau_2=\tau_3*\tau_4}\\
x : \tau_3,\ (p\ (x+2)) : \tau_4 &\qquad \nwarrow_{\hat{x}=\tau_3}\\
(p\ (x+2)) : \tau_4 &\qquad \nwarrow_{\hat{p}=int\to\tau_4,\ \hat{x}=int}
\end{aligned}$$

So we have:

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

That is:

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

Thus:

$$int * \tau_4 = \tau_4 \textit{ which is absurd}$$

The above argument is represented more concisely below:

$$t = \mathbf{rec}\ \ p.\lambda\ \underbrace{\underbrace{x}_{int}\ .\ \underbrace{(\underbrace{x}_{int},(\underbrace{\underbrace{p}_{int\to\tau}\ \ \underbrace{(\underbrace{x}_{int}+2)}_{int}}_{\tau}))}_{int*\tau}}_{int\to int*\tau\ =\ int\to\tau\ \Rightarrow\ \tau=int*\tau}$$

So we have no solutions, and the term is not an HOFL term, no matter what is the value of $\hat{p}$ and $\hat{x}$.

6.1.2. Typability and Typechecking

As we said in the last section we will give semantics only to well-formed terms, namely terms which have a type in our type system. Therefore we need an algorithm to say if a term is well-formed. In this section we will present two different solutions to the typability problem, introduced by Church and by Curry, respectively.

6.1.2.1. Church Type Theory

In Church type theory we explicitly associate a type to each variable and deduce the type of each term by structural recursion (i.e. by using the rules in a bottom-up fashion).

Let us show how it works by typing the factorial function:

Example 6.4 (Factorial with Church Types)
Let $x : int$ and $f : int \to int$. So we can type all the subterms:

$$\mathbf{fact} \overset{\text{def}}{=} \mathbf{rec}\ \underbrace{f}_{int\to int} . \underbrace{\lambda \underbrace{x}_{int} . \underbrace{\mathbf{if}\ \underbrace{x}_{int}\ \mathbf{then}\ \underbrace{1}_{int}\ \mathbf{else}\ \underbrace{\underbrace{x}_{int} \times (\underbrace{\underbrace{f}_{int\to int} (\underbrace{\underbrace{x}_{int} - \underbrace{1}_{int}}_{int})}_{int})}_{int}}_{int}}_{int\to int} \qquad : int \to int$$

6.1.2.2. Curry Type Theory

In Curry types we do not need to explicitly declare the type of each variable. We will use the inference rules to calculate type equations (i.e. equations which have types as variables) whose solutions will be all the possible types of the term. This means that the result will be a set of types associated to the typed term. The surprising fact is that this set can be represented as all the instantiation of a single term with variables, where one instantiation is obtained by replacing each variable with any type. We call this term with variables the *principal type* of the term. This construction is made by using the rules in a goal oriented fashion.

Example 6.5 (Identity)
Let us consider the identity function:

$$\lambda x.\ x$$

By using the type system we have:

$$\begin{array}{rl} \lambda x.\ x : \tau & \nwarrow_{\tau=\tau_1\to\tau_2} \\ x : \tau_1\ x : \tau_2 & \nwarrow_{\hat{x}=\tau_2=\tau_1} \\ \square & \end{array}$$

So we have that the principal type is $\lambda x.\ x : \tau_1 \to \tau_1$. Now each solution of the type equation will be an identity function for a specified type. For example if we set $\tau_1 = int$ we have $\tau = int \to int$.

As we will see in the next example and as we already saw in example 6.3, the typing problem reduces to that of resolving a system of type equations. We will also introduce a standard way to find a solution of the system (if it exists).

Example 6.6 (Non-typable term of HOFL)
Let us consider the following function, which computes the factorial without using recursion.

$$\begin{array}{l} \mathbf{begin} \\ \qquad \mathbf{fact}(f, x) \overset{\text{def}}{=} \mathbf{if}\ x = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \times f(f, x-1) \\ \qquad \mathbf{fact}(\mathbf{fact}, n) \\ \mathbf{end} \end{array}$$

It can be written in HOFL as follows:

$$\mathbf{fact} \overset{\text{def}}{=} \lambda \underbrace{y}_{\tau*int} .\ \mathbf{if}\ \underbrace{\mathbf{snd}(\underbrace{y}_{\tau*int})}_{int}\ \mathbf{then}\ \underbrace{1}_{int}\ \mathbf{else}\ \underbrace{\mathbf{snd}(\underbrace{y}_{\tau*int})}_{int} \times (\underbrace{\underbrace{\mathbf{fst}(\underbrace{y}_{\tau*int})}_{\tau=\tau*int\to int}\ \underbrace{(\underbrace{\mathbf{fst}(\underbrace{y}_{\tau*int})}_{\tau},\ \underbrace{\mathbf{snd}(\underbrace{y}_{\tau*int})}_{int} - \underbrace{1}_{int})}_{\tau*int}}_{int})$$

We conclude $\mathbf{fst}(y) : \tau$ *and* $\mathbf{fst}(y) : \tau * int \to int$*. Thus we have* $\tau = \tau * int \to int$ *which is an equation which has no solution.*

We present an algorithm, called *unification*, to solve general systems of type equations. We start from a system of equation of the type:

$$\begin{cases} t_1 = t'_1 \\ t_2 = t'_2 \\ \dots \end{cases}$$

where t's are type terms, with type variables denoted by τ's, then we apply iteratively in any order the following steps:

1) We eliminate all the equations like $\tau = \tau$.

2) For each equation of the form $f(t_1, \dots, t_n) = f'(t'_1, \dots, t'_m)$: if $f \neq f'$, then the system has no solutions. if $f = f'$ then $m = n$ so we must have:

$$t_1 = t'_1, t_2 = t'_2, \dots, t_n = t'_n$$

Then we replace the original equation with these.

3) For each equation of the type $\tau = t$, $t \neq \tau$: we let $\tau \overset{\text{def}}{=} t$ and we replace each occurrence of τ with t in all the other equations. If $\tau \in t$ then the system has no solutions.

Eventually, either the system is recognized as unsolvable, or all the variables in the original equations are assigned to solution terms. Note that the order of the step executions can affect the complexity of the algorithm but not the solution. The best execution strategies yield a complexity linear or quasi linear with the size of the original system of equations.

Example 6.7

Let us now apply the algorithm to the previous example:

$$\tau = \tau * int \to int$$

step 3 fails since type variable τ *appears in the right hand side.*

6.2. Operational Semantics of HOFL

We are now ready to present the (lazy) operational semantics of HOFL. Unlike IMP the operational semantics of HOFL is a simple manipulation of terms. This means that the operational semantics of HOFL defines a method to calculate the *canonical form* of a given closed term of HOFL. Canonical forms are closed terms, which we will consider the results of calculations (i.e. values). For each type we will define a set of terms in canonical form by taking a subset of terms which reasonably represent the values of that type. As shown in the previous section, HOFL has three type constructors: integer, pair and arrow. Obviously, terms which represent the integers are the canonical forms for the integer type. For pair type we will take pairs of terms (note that this choice is arbitrary, for example we could take pairs of terms in canonical form). Finally, since HOFL is a higher-order language, functions are values. So is quite natural to take all abstractions as canonical forms for the arrow type.

Definition 6.8 (Canonical forms)
Let us define a set C_τ of canonical forms for each type τ as follows:

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

$$\frac{t_0 : \tau_0 \quad t_1 : \tau_1 \quad t_0, t_1 \; closed}{(t_0, t_1) \in C_{\tau_0 * \tau_1}}$$

$$\frac{\lambda x.t : \tau_0 \rightarrow \tau_1 \quad \lambda x.t \; closed}{\lambda x.t \in C_{\tau_0 \rightarrow \tau_1}}$$

We now define the rules of the operational semantics, these rules define an evaluation relation:

$$t \longrightarrow c$$

where t is a well-formed closed term of HOFL and c is its canonical form. So we define:

$$\frac{}{c \rightarrow c} \qquad \frac{t_0 \rightarrow n_0 \quad t_1 \rightarrow n_1}{t_0 \text{ op } t_1 \rightarrow n_0 \; \underline{op} \; n_1} \qquad \frac{t_0 \rightarrow 0 \quad t_1 \rightarrow c_1}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \rightarrow c_1} \qquad \frac{t_0 \rightarrow n \quad n \neq 0 \quad t_2 \rightarrow c_2}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \rightarrow c_2}$$

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 **if** t_0 **then** t_1 **else** t_2 stand for **if** $t_0 = 0$ **then** t_1 **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:

$$\frac{t \rightarrow (t_0, t_1) \quad t_0 \rightarrow c_0}{\textbf{fst}(t) \rightarrow c_0} \qquad \frac{t \rightarrow (t_0, t_1) \quad t_1 \rightarrow c_1}{\textbf{snd}(t) \rightarrow c_1}$$

For function application, we give a lazy operational semantics, this means that we do not evaluate the parameters before replacing them with the copy rule in the function body. If the argument is actually needed it may be later evaluated several times.

$$\frac{t_1 \rightarrow \lambda x.t_1' \quad t_1'[^{t_0}/_x] \rightarrow c}{(t_1 \quad t_0) \rightarrow c} \quad \text{(lazy)}$$

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.

$$\frac{t_1 \rightarrow \lambda x.t_1' \quad t_0 \rightarrow c_0 \quad t_1'[^{c_0}/_x] \rightarrow c}{(t_1 \quad t_0) \rightarrow c} \quad \text{(eager)}$$

Finally we have a rule for "Rec":

$$\frac{t[^{\textbf{rec}\, x.t}/_x] \rightarrow c}{\textbf{rec}\, x.t \rightarrow c}$$

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:

$$\begin{array}{rcl}
(\mathbf{fact}\,2) \to c & \nwarrow & \mathbf{fact} \to \lambda x.t \quad t[^2/_x] \to c \\
 & \nwarrow & \lambda x.\, \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (\mathbf{fact}(x-1)) \to \lambda x.t \quad t[^2/_x] \to c \\
 & \nwarrow_{t=\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times \mathbf{fact}(x-1)} & \mathbf{if}\ 2\ \mathbf{then}\ 1\ \mathbf{else}\ 2 \times (\mathbf{fact}(2-1)) \to c \\
 & \nwarrow & 2 \times (\mathbf{fact}(2-1)) \to c \\
 & \nwarrow_{c=c_1 \times c_2} & 2 \to c_1 \quad (\mathbf{fact}(2-1)) \to c_2 \\
 & \nwarrow_{c_1=2} & \mathbf{fact} \to \lambda x.t \quad \underbrace{t[^{2-1}/_x] \to c_2}_{2-1\ \textit{not evaluated}} \\
 & \nwarrow & \mathbf{if}(2-1)\ \mathbf{then}\ 1\ \mathbf{else}(2-1) \times (\mathbf{fact}((2-1)-1)) \to c_2 \\
 & \nwarrow & 2-1 \to n \quad n \neq 0 \quad (2-1) \times \mathbf{fact}((2-1)-1) \to c_2 \\
 & \nwarrow_{n=n_1-n_2} & 2 \to n_1 \quad 1 \to n_2 \quad 2-1 \to n_1 - n_2 \quad n_1 - n_2 \neq 0 \\
 & & (2-1) \times \mathbf{fact}((2-1)-1) \to c_2 \\
 & \nwarrow_{n_1=2 \quad n_2=1} & 2-1 \to 1 \quad 1 \neq 0 \quad (2-1) \times \mathbf{fact}((2-1)-1) \to c_2 \\
 & \nwarrow_{c_2=c_3 \times c_4} & 2-1 \to c_3 \quad \mathbf{fact}((2-1)-1) \to c_4 \\
 & \nwarrow_{c_3=1} & \mathbf{if}(2-1)-1\ \mathbf{then}\ 1\ \mathbf{else}((2-1)-1) \times (\mathbf{fact}(((2-1)-1)-1)) \to c_4 \\
 & \nwarrow^{*} & (2-1)-1 \to 0 \qquad 1 \to c_4 \\
 & \nwarrow_{c_4=1\ c_3=1\ c_2=c_3 \times c_4=1\ c=c_1 \times c_2 \times 1=2} & \square
\end{array}$$

Example 6.10 (Non-equivalence of lazy and eager evaluations)
The aim of this example is to illustrate the difference between lazy and eager semantics.

- ***Lazy evaluation***
  Lazy evaluation evaluates the terms only if needed: if a parameter is never used in a function or in a specific instance of a function it will never be evaluated. Let us show an example:

  $$\begin{array}{rcl}
  ((\lambda x : int.3)\, \mathbf{rec}\, y : int.y) : int \to c & \nwarrow & \lambda x : int.3 \to \lambda x.t \quad t[^{\mathbf{rec}\, y.y}/_x] \to c \\
   & \nwarrow_{t=3} & 3[^{\mathbf{rec}\, y.y}/_x] \to c \\
   & \nwarrow_{c=3} & \square
  \end{array}$$

  So although the term $\mathbf{rec}\, y.y$ *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:

  $$\begin{array}{rcl}
  ((\lambda x : int.3)\, \mathbf{rec}\, y : int.y) : int \to c & \nwarrow & \lambda x : int.3 \to \lambda x.t \quad \mathbf{rec}\, y.y \to c_1 \quad t[^{c_1}/_x] \to c \\
   & \nwarrow_{t=3} & \mathbf{rec}\, y.y \to c_1 \quad 3[^{c_1}/_x] \to c \\
   & \nwarrow_{y=\mathbf{rec}\, y.y} & \mathbf{rec}\, y.y \to c_1 \quad 3[^{c_1}/_x] \to c \\
   & \nwarrow_{y=\mathbf{rec}\, y.y} & \ldots
  \end{array}$$

So the evaluation does not terminate. However if the parameter of a function is used n times, the parameter would be evaluated n times (at most) in the lazy semantics and only once in the eager case.

Theorem 6.11

i) If $t \to c$ and $t \to c'$ then $c = c'$ (if a canonical form exists it is unique)

ii) if $t \to c$ and $t : \tau$ then $c : \tau$ (the evaluation relation respects the types)

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

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[^{t_2}/_x] \to c}{(t_1 \quad t_2) \to c}$$

We will show:

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

As usually let us assume the premise:

$$(t_1 \quad t_2) \longrightarrow c'$$

And the inductive hypothesis:

- $P(t_1 \to \lambda x.t_1') \stackrel{\text{def}}{=} \forall c' \; t_1 \longrightarrow c' \Rightarrow \lambda x.t_1' = c'$
- $P(t_1'[^{t_2}/_x] \to c) \stackrel{\text{def}}{=} \forall c' \; t_1'[^{t_2}/_x] \longrightarrow c' \Rightarrow c = c'$

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

- $t_1 \to \lambda \overline{x}.\overline{t_1}'$
- $\overline{t_1}'[^{t_2}/_{\overline{x}}] \to c'$

then we have by inductive hypothesis:

- $\lambda x.t_1' = \lambda \overline{x}.\overline{t_1}'$
- $t_1'[^{t_2}/_x] = \overline{t_1}'[^{t_2}/_{\overline{x}}]$

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 $CPO_\perp$:

$$\begin{array}{rcl}\mathscr{D} & = & (D, \sqsubseteq_D)\\ \mathscr{E} & = & (E, \sqsubseteq_E)\end{array}$$

Now we construct the Cartesian product $\mathscr{D} \times \mathscr{E} = (D \times E, \sqsubseteq_{D\times E})$ 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 $e \sqsubseteq_D e_2$. 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 $\forall d \in D, e \in E.\ \perp_D \sqsubseteq d \wedge \perp_E \sqsubseteq e$, thus $(\perp_D, \perp_E) \sqsubseteq_{D\times E} (d, e)$. It remains to show the completeness of $\mathscr{D} \times \mathscr{E}$.

Theorem 7.1 (Completeness of $\mathscr{D} \times \mathscr{E}$)
The PO $\mathscr{D} \times \mathscr{E}$ 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 $\bigsqcup_{i\in\omega} e_i \sqsubseteq_D \bar{e}$. 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. □

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

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 $\pi_1 : D \times E \to D$ and $\pi_2 : D \times E \to E$ as follows.

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

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 $\mathscr{D} \times \mathscr{E}$ 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:

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

and

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

Proof. Let us prove the first statement:

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

For π_2 the proof is analogous. □

7.3. Functional Domains

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

Let us consider the CPOs:

$$
\begin{array}{rcl}
\mathscr{D} & = & (D, \sqsubseteq_D) \\
\mathscr{E} & = & (E, \sqsubseteq_E)
\end{array}
$$

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

$$[\mathscr{D} \to \mathscr{E}] = ([D \to E], \sqsubseteq_{[D \to E]})$$

where:

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

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

$$
\begin{array}{ccccccccc}
e_{00} & \sqsubseteq & e_{01} & \sqsubseteq & e_{02} & \sqsubseteq & \cdots & \bigsqcup_{i \in \omega} e_{0i} = e_0 \\
\rotatebox{-90}{$\sqsubseteq$} & & \rotatebox{-90}{$\sqsubseteq$} & & \rotatebox{-90}{$\sqsubseteq$} & & & \rotatebox{-90}{$\sqsubseteq$} \\
e_{10} & \sqsubseteq & e_{11} & \sqsubseteq & e_{12} & \sqsubseteq & \cdots & \bigsqcup_{i \in \omega} e_{1i} = e_1 \\
\rotatebox{-90}{$\sqsubseteq$} & & \rotatebox{-90}{$\sqsubseteq$} & & \rotatebox{-90}{$\sqsubseteq$} & & & \rotatebox{-90}{$\sqsubseteq$} \\
e_{20} & \sqsubseteq & e_{21} & \sqsubseteq & e_{22} & \sqsubseteq & \cdots & \bigsqcup_{i \in \omega} e_{2i} = e_2 \\
\rotatebox{-90}{$\sqsubseteq$} & & \rotatebox{-90}{$\sqsubseteq$} & & \rotatebox{-90}{$\sqsubseteq$} & & & \rotatebox{-90}{$\sqsubseteq$} \\
\vdots & & \vdots & & \vdots & & \ddots & \vdots
\end{array}
$$

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

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

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 $\{\bigsqcup_{m \in \omega} e_{n,m}\}_{n \in \omega}$ has a least element. In fact, $n_1 \leq n_2$ implies $\bigsqcup_{m \in \omega} e_{n_1,m} \sqsubseteq \bigsqcup_{m \in \omega} e_{n_2,m}$, since every upper bound of $\bigsqcup_{m \in \omega} e_{n_2,m}$ is an upper bound of $\bigsqcup_{m \in \omega} e_{n_1,m}$. Therefore $\{\bigsqcup_{m \in \omega} e_{n,m}\}_{n \in \omega}$ is a chain, and thus it has a LUB since E is a CPO. □

Lemma 7.5
Let $\{f_n\}_{n\in\omega}$ *be a chain of functions (not necessarily continuous) in* $\mathscr{D} \to \mathscr{E}$ *the LUB* $\bigsqcup_{n\in\omega} f_n$ *exists and is defined as:*

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

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

Lemma 7.6
Let $\{f_n\}_{n\in\omega}$ *be a chain of functions in* $[\mathscr{D} \to \mathscr{E}]$ *and let* $\{d_n\}_{n\in\omega}$ *be a chain on* $\mathscr{D}$ *then function* $\lambda d. \bigsqcup_{n\in\omega}(f_n(d))$ *is continuous, namely*

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

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

Proof.

$$\begin{array}{ll} \bigsqcup_n (f_n (\bigsqcup_m d_m)) = & \text{by continuity} \\ \bigsqcup_n (\bigsqcup_m (f_n (d_m))) = & \text{by lemma 7.4 (switch lemma)} \\ \bigsqcup_m (\bigsqcup_n (f_n (d_m))) & \end{array}$$

The upper bounds of $\{f_n\}_{n\in\omega}$ in $\mathscr{D} \to \mathscr{E}$ are a larger set then those in $[\mathscr{D} \to \mathscr{E}]$, thus if $\lambda d. \bigsqcup_{n\in\omega}(f_n(d))$ is the LUB in $\mathscr{D} \to \mathscr{E}$, it is also the LUB in $[\mathscr{D} \to \mathscr{E}]$. □

Theorem 7.7 (Completeness of the functional space)
The PO $[\mathscr{D} \to \mathscr{E}]$ *is a* $CPO_\perp$

Proof. The statement follows immediately from the previous Lemmas. □

7.4. Lifting

In IMP we introduced a lifting operator (Chapter 5.2.3) on memories Σ to obtain a CPO $\Sigma_\perp$. 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 $\mathscr{D} = (D, \sqsubseteq_D)$ *be a CPO and let* $\perp$ *be an element not in D, so we define the lifted* $CPO_\perp$ $\mathscr{D}_\perp$ *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 $\lfloor - \rfloor : D \longrightarrow D_\perp$ *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 $\mathscr{D}$ 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 $\mathscr{D}$ be a CPO and let $\mathscr{E}$ be a $CPO_\perp$. We define a lifting operator $\_^* : [D \rightarrow E] \rightarrow [D_\perp \rightarrow E]$ for functions in $[D \rightarrow E]$ as follows:

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

Theorem 7.10

i) If f is continuous in $[D \rightarrow E]$, then f^* is continuous in $[D_\perp \rightarrow E]$.

ii) The operator $\_^*$ is continuous.

Proof.

i) Let $\{d_i\}_{i\in\omega}$ be a chain in $\mathscr{D}_\perp$. 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 $[\mathscr{D} \rightarrow \mathscr{E}]$. 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:

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

□

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:

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

□

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.

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

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 $g_1 : S \longrightarrow D$ and $g_2 : S \longrightarrow E$, then we can prove the continuity of f from the continuity of g_1 and g_2.

Theorem 7.12
Let $f : S \longrightarrow D \times E$ be a function over CPOs and let $g_1 : S \longrightarrow D$ and $g_2 : S \longrightarrow E$ 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

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

Then we have:
f is continuous $\Longleftrightarrow$ g_1 and g_2 are continuous.

Proof.

$\Leftarrow$) 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:

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

$\Rightarrow$) Immediate by Theorem 7.11 (continuity of composition) and Theorem7.3 (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 $f : D_1 \times D_2 \longrightarrow E$ 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 $f : D_1 \times D_2 \to E$ be a function over CPOs. Then f is continuous iff all the functions in the following two classes are continuous.

$$\forall d_1.\ f_{d_1} : D_2 \to E \text{ defined as } f_{d_1} = \lambda d.f(d_1, d)$$
$$\forall d_2.\ f_{d_2} : D_1 \to E \text{ defined as } f_{d_2} = \lambda d.f(d, d_2)$$

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).

⇐) On the other hand we have:

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

□

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 apply : $[D \to E] \times D \to E$ *over domains as follows:*

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

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 apply : $[D \to E] \times D \to E$ *be the function defined above and let* $\{f_i, d_i\}_{i\in\omega}$ *be a chain on the* CPO$_\perp$ $[\mathscr{D} \to \mathscr{E}] \times \mathscr{D}$ *then:*

$$\mathrm{apply}(\bigsqcup_{i\in\omega}(f_i, d_i)) = \bigsqcup_{i\in\omega} \mathrm{apply}(f_i, d_i)$$

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

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

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

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

So apply is a continuous function. □

Definition 7.16 (Curry and un-curry)
We define the function curry : $(D \times F \to E) \to (D \to (F \to E))$ *as:*

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

And we define the un-curry : $(D \to F \to E) \to (D \times F \to E)$ *as:*

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

Theorem 7.17 (Continuity of the curry of a function)
Let curry : $(D \times F \to E) \to (D \to (F \to E))$ *be the function defined above let* $\{d_i\}_{i\in\omega}$ *be a chain on* $\mathscr{D}$ *and let* $g : D \times F \to E$ *a continuous function then* (curry g) *is a continuous function, namely*

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

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:

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

□

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* fix : $([D \to D] \to D)$ *as:*

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

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

Theorem 7.19 (Continuity of fix)
Function fix : $([D \to D] \to D)$ *is continuous, namely* fix : $[[D \to D] \to D]$.

Proof. We know that $[[D \to D] \to D]$ is complete, thus if for all i $\lambda f.\ f^i(\perp_D)$ is continuous, then $\bigsqcup_{i\in\omega} \lambda f.\ f^i(\perp_D) =$ 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:

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

□

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

$$\textbf{let}\ \ x \Leftarrow d_\perp.\ e \overset{\text{def}}{=} \underbrace{\underbrace{\underbrace{(\lambda x.\ e)}_{D\to E}{}^*}_{D_\perp\to E}\ \underbrace{(d_\perp)}_{D_\perp}}_{E} = \begin{cases} \perp_E & \mathit{if}\ d_\perp = \perp_{D_\perp} \\ e\left[{}^d/_x\right] & \mathit{if}\ d_\perp = \lfloor d \rfloor \end{cases}$$

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 $[\![t]\!]$. However, since the terms are typed, the evaluation function is parametrised by the types. We have

$$[\![t : \tau]\!] : Env \longrightarrow (V_\tau)_\perp$$

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:

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

Notice that the recursive definition takes advantage of the domain constructors we defined in Chapter 7. While the lifiting $\mathbb{N}_\perp$ 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:

$$[\![n]\!]\rho = \lfloor n \rfloor$$

8.1.2. Variables

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

$$[\![x]\!]\rho = \rho x$$

8.1.3. Binary Operators

We give the generic semantics of a binary operator as:

$$[\![t_0 \ \mathbf{op} \ t_1]\!]\rho = [\![t_0]\!]\rho \ \underline{op}_\perp \ [\![t_1]\!]\rho$$

where

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

$$x_1 \underline{op}_\perp x_2 = \left| \begin{array}{l} Case(x_1, x_2) \\ (\lfloor n_1 \rfloor, \lfloor n_2 \rfloor) : \lfloor n_1 \ \underline{op} \ n_2 \rfloor \\ \perp_{\mathbb{N}_\perp} \text{ otherwise} \end{array} \right.$$

notice that for every operator $op \in \{+, -, \times, \ldots\}$ 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:

$$[\![\mathbf{if} \ t_0 \ \mathbf{then} \ t_1 \ \mathbf{else} \ t_2]\!]\rho = Cond\,([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho)$$

where

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

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

8.1.5. Pairing

For the pairing operator we have:

$$[\![(t_0, t_1)]\!]\rho = \lfloor([\![t_0]\!]\rho, [\![t_1]\!]\rho)\rfloor$$

8.1.6. Projections

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

$$\begin{aligned} [\![\mathbf{fst}(t)]\!]\rho &= \mathbf{let}\, d \Leftarrow [\![t]\!]\rho.\ \pi_1 d = (\lambda d.\pi_1 d)^* \ [\![t]\!]\rho \\ [\![\mathbf{snd}(t)]\!]\rho &= \mathbf{let}\, d \Leftarrow [\![t]\!]\rho.\ \pi_2 d = (\lambda d.\pi_2 d)^* \ [\![t]\!]\rho \end{aligned}$$

as we said in the previous chapter the "let" operator allows to *de-lift* $[\![t]\!]\rho$ 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:

$$[\![\lambda x.t]\!]\rho = \left\lfloor \lambda d.\ [\![t]\!]\rho[^d/_x] \right\rfloor$$

8.1.8. Function Application

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

$$[\![(t_1 \ t_0)]\!]\rho = \mathbf{let}\, \varphi \Leftarrow [\![t_1]\!]\rho.\ \varphi([\![t_0]\!]\rho) = (\lambda\varphi.\varphi\,([\![t_0]\!]\rho))^* \ [\![t_1]\!]\rho$$

8.1.9. Recursion

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

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

8.2. Typing the Clauses

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

Constants

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

as required.

Variables

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

Binary operations

$$\underbrace{\llbracket t_0\ op\ t_1 : int \rrbracket \rho}_{(V_{int})_\perp} = \underbrace{\underbrace{\llbracket t_0 \rrbracket\, \rho}_{(V_{int})_\perp} \quad \underbrace{\underline{op}_\perp}_{(V_{int})_\perp \times (V_{int})_\perp \to (V_{int})_\perp} \quad \underbrace{\llbracket t_1 \rrbracket \rho}_{(V_{int})_\perp}}_{(V_{int})_\perp}$$

Conditional

$$\underbrace{\llbracket \mathbf{if}\ \ t_0 : int\ \ \mathbf{then}\ \ t_1 : \tau\ \ \mathbf{else}\ \ t_2 : \tau \rrbracket \rho}_{(V_\tau)_\perp} = \underbrace{\underbrace{Cond_\tau}_{\mathbb{N}_\perp \times (V_\tau)_\perp \times (V_\tau)_\perp \longrightarrow (V_\tau)_\perp} \quad (\underbrace{\llbracket t_0 \rrbracket\, \rho,}_{(V_{int})_\perp} \underbrace{\llbracket t_1 \rrbracket\, \rho,}_{(V_\tau)_\perp} \underbrace{\llbracket t_2 \rrbracket\, \rho}_{(V_\tau)_\perp})}_{(V_\tau)_\perp}$$

Pairing

$$\underbrace{\llbracket (t_0 : \tau_0, t_1 : \tau_1) \rrbracket \rho}_{(V_{\tau_0 * \tau_1})_\perp} = \underbrace{\lfloor \underbrace{(\underbrace{\llbracket t_0 \rrbracket \rho}_{(V_{\tau_0})_\perp}, \underbrace{\llbracket t_1 \rrbracket \rho}_{(V_{\tau_1})_\perp})}_{(V_{\tau_0})_\perp \times (V_{\tau_1})_\perp} \rfloor}_{((V_{\tau_0})_\perp \times (V_{\tau_1})_\perp)_\perp}$$

Projections

$$\underbrace{\llbracket \mathbf{fst}(t : \tau_0 * \tau_1) \rrbracket \rho}_{(V_{\tau_0})_\perp} = \mathbf{let}\quad \underbrace{d}_{(V_{\tau_0})_\perp \times (V_{\tau_1})_\perp} \quad \Leftarrow \quad \underbrace{\llbracket t \rrbracket \rho}_{(V_{\tau_0 * \tau_1})_\perp} \quad . \quad \underbrace{\underbrace{\pi_1}_{(V_{\tau_0})_\perp \times (V_{\tau_1})_\perp \to (V_{\tau_0})_\perp} \quad d}_{(V_{\tau_0})_\perp}$$

Lambda abstraction

$$\underbrace{\llbracket \lambda x : \tau_0.t : \tau_1 \rrbracket \rho}_{(V_{\tau_0 \longrightarrow \tau_1})_\perp} = \underbrace{\lfloor \lambda \quad \underbrace{\underbrace{d}_{(V_{\tau_0})_\perp} \quad . \underbrace{\llbracket t \rrbracket \rho[{}^{d}/_{x}]}_{(V_{\tau_1})_\perp}}_{[(V_{\tau_0})_\perp \to (V_{\tau_1})_\perp]} \rfloor}_{[(V_{\tau_0})_\perp \to (V_{\tau_1})_\perp]_\perp}$$

Function application

$$\underbrace{[\![(t_1 : \tau_0 \longrightarrow \tau_1\ t_0 : \tau_0)]\!]\rho}_{(V_{\tau_1})_\perp} = \mathbf{let}\ \underbrace{\varphi}_{[(V_{\tau_0})_\perp \to (V_{\tau_1})_\perp]} \Leftarrow \underbrace{[\![t_1]\!]\rho}_{(V_{\tau_0\to\tau_1})_\perp}\ .\, \underbrace{\varphi(\underbrace{[\![t_0]\!]\rho}_{(V_{\tau_0})_\perp})}_{(V_{\tau_1})_\perp}$$

Recursion

$$\underbrace{[\![\mathbf{rec}\,x : \tau.t : \tau]\!]\rho}_{(V_\tau)_\perp} = \underbrace{\underbrace{fix}_{[[(V_\tau)_\perp\to(V_\tau)_\perp]\to(V_\tau)_\perp]}\ \underbrace{\underbrace{\lambda d}_{(V_\tau)_\perp}\ .\, \underbrace{[\![t]\!]\rho[^d/_x]}_{(V_\tau)_\perp}}_{[(V_\tau)_\perp\to(V_\tau)_\perp]}}_{(V_\tau)_\perp}$$

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. $f \overset{\text{def}}{=} \lambda x : int.3$
2. $g \overset{\text{def}}{=} \lambda x : int.\,\mathbf{if}\ x\ \mathbf{then}\ 3\ \mathbf{else}\ 3$
3. $h \overset{\text{def}}{=} \mathbf{rec}\, y : int \to int.\lambda x : int.3$

1. $[\![f]\!]\rho = [\![\lambda x : int.3]\!]\rho = \lfloor \lambda d.[\![3]\!]\rho[^d/_x]\rfloor = \lfloor \lambda d.\lfloor 3\rfloor\rfloor$
2. $[\![g]\!]\rho = [\![\lambda x : int.\,\mathbf{if}\ x\ \mathbf{then}\ 3\ \mathbf{else}\ 3]\!]\rho = \lfloor \lambda d.[\![\mathbf{if}\ x\ \mathbf{then}\ 3\ \mathbf{else}\ 3]\!]\rho[^d/_x]\rfloor = \lfloor \lambda d.Cond(d, \lfloor 3\rfloor, \lfloor 3\rfloor)\rfloor = \lfloor \lambda d.\,\mathbf{let}\, x \Leftarrow d.\lfloor 3\rfloor\rfloor$
3. $[\![h]\!]\rho = [\![\mathbf{rec}\, y : int \to int.\lambda x : int.3]\!]\rho = \mathbf{fix}\,\lambda d.[\![\lambda x.3]\!]\rho[^d/_x] = \mathbf{fix}\,\lambda d.\lfloor \lambda d'.\lfloor 3\rfloor\rfloor$
   - $d_0 = \perp_{[\mathbb{N}_\perp\to\mathbb{N}_\perp]_\perp}$
   - $d_1 = (\lambda d.\lfloor \lambda d'.\lfloor 3\rfloor\rfloor)\perp = \lfloor \lambda d'.\lfloor 3\rfloor\rfloor$
   - $d_2 = (\lambda d.\lfloor \lambda d'.\lfloor 3\rfloor\rfloor)\lfloor \lambda d'.\lfloor 3\rfloor\rfloor = \lfloor \lambda d'.\lfloor 3\rfloor\rfloor = d_1 = [\![h]\!]\rho$

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:

- $\underline{op}_\perp$
- $Cond$
- $(\_, \_)$
- $\pi_1,\ \pi_2$
- *let*
- *apply*
- *fix*

Proof.

- $\underline{op}_\perp$: Since $\underline{op}_\perp$ 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 $\perp_{\mathbb{N}_\perp}$
  - $v = 0$ 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:

- $(\lambda d : (V_{\tau_1})_\perp . [\![t : \tau_2]\!]\rho[^d/_x]) : (V_{\tau_1})_\perp \longrightarrow (V_{\tau_2})_\perp$ *is a continuous function.*
- fix $\lambda d.[\![t]\!]\rho[^d/_x]$ *is a continuous function.*

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

- $t = x$: $\lambda d.[\![x]\!]\rho[^d/_x]$ is equal to $\lambda d.d$ which is obviously continuous.
- $t = t_1$ **op** t_2: $\lambda d.[\![t_1\ \mathbf{op}\ t_2]\!]\rho[^d/_x] = \lambda d.[\![t_1]\!]\rho[^d/_x]\ \underline{op}_\perp\ [\![t_2]\!]\rho[^d/_x]$ it is continuous since $\underline{op}_\perp$, $[\![t_1]\!]\rho[^d/_x]$ and $[\![t_2]\!]\rho[^d/_x]$ 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.
- $t = \lambda y.t'$: $\lambda d.[\![\lambda y.t']\!]\rho[^d/_x]$ is obviously continuous if $x = y$. Otherwise if $x \neq y$ we have by induction hypothesis that $\lambda(d, d').[\![t']\!]\rho[^d/_x, ^{d'}/_y]$ is continuous, then:

$$\begin{array}{ll} \text{curry}(\lambda(d, d').[\![t']\!]\rho[^d/_x, ^{d'}/_y]) = & \text{is continuous since curry is continuous} \\ \lambda d.\lambda d'.[\![t']\!]\rho[^d/_x][^{d'}/_y] = & \\ \lambda d.[\![\lambda y.t']\!]\rho[^d/_x] & \end{array}$$

We leave the remaining cases as an exercise.

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

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*

$$\left[\!\left[t[^{t'}/_{x}] \right]\!\right] \rho = [\![t]\!] \rho[^{[\![t']\!]\rho}/_{x}]$$

Proof. By structural induction. □

The substitution lemma is an important result, as it implies the compositionality of denotational semantics:

$$[\![t_1]\!]\rho = [\![t_2]\!]\rho \Rightarrow \left[\!\left[t[^{t_1}/_{x}] \right]\!\right]\rho = \left[\!\left[t[^{t_2}/_{x}] \right]\!\right]\rho$$

In words, replacing a variable x with a term t' in a term t returns a term $t[^{t'}/_{x}]$ whose denotational semantics $\left[\!\left[t[^{t'}/_{x}] \right]\!\right] \rho = [\![t]\!] \rho[^{[\![t']\!]\rho}/_{x}]$ depends only on the denotational semantics $[\![t']\!]\rho$ of t' and *not* on t itself.

Theorem 8.5
Let t be a well-defined term of HOFL. Let $\rho, \rho' \in Env$ *such that* $\forall\ x \in FV(t).\ \rho(x) = \rho'(x)$ *then:*

$$[\![t]\!]\rho = [\![t]\!]\rho'$$

Proof. By structural induction. □

Theorem 8.6
Let $c \in C_\tau$ *be a closed term in canonical form of type* τ*. Then we have:*

$$\forall\ \rho \in Env.\ [\![c]\!]\rho \neq \bot_{(V_\tau)_\bot}$$

Proof. Immediate, by inspection of the clauses for terms in canonical forms. □

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 $\Longleftarrow$ 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* $x : int$*, then we have:*

$$\llbracket t \rrbracket \rho = \llbracket c \rrbracket \rho$$

but

$$t \nrightarrow c$$

In fact we have:

$$\llbracket t \rrbracket \rho = \llbracket \lambda x.\ x + 0 \rrbracket \rho = \lfloor \lambda d.\ d \underline{+}_\perp \lfloor 0 \rfloor \rfloor = \lfloor \lambda d.d \rfloor = \llbracket \lambda x.\ x \rrbracket \rho = \llbracket c \rrbracket \rho$$

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 $int \to int$*, there are different canonical forms with the same denotational semantics, namely terms which compute the same function* $[\mathbb{N}_\perp \to \mathbb{N}_\perp]_\perp$*. 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:

$$t \Downarrow \iff \exists v \in V_\tau.\ [\![t]\!]\rho = \lfloor v \rfloor$$

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:

$$t \downarrow \Longrightarrow t \Downarrow$$

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:

$$t \to c \Longrightarrow \forall \rho \in Env\ \ [\![t]\!]\rho = [\![c]\!]\rho$$

Proof. As usual we proceed by rule induction. So we will prove $P(t \to c) \equiv \forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho$ on each rule by assuming the premises.

$$c \to c$$

We have to prove $P(c \to c)$ that is by definition $\forall \rho.\ [\![c]\!]\rho = [\![c]\!]\rho$, which is obviously true.

$$\frac{t_1 \to n_1 \quad t_2 \to n_2}{t_1 \text{ op } t_2 \to n_1\ \underline{op}\ n_2}$$

We have to prove $P(t_1 \text{ op } t_2 \to n_1\ \underline{op}\ n_2) \equiv \forall \rho.\ [\![t_1 \text{ op } t_2]\!]\rho = \left[\!\left[n_1\ \underline{op}\ n_2\right]\!\right]\rho$.

Let us assume the property on the premises:

$P(t_1 \to n_1) \equiv \forall \rho.\ [\![t_1]\!]\rho = [\![n_1]\!]\rho = \lfloor n_1 \rfloor$ and $P(t_2 \to n_2) \equiv \forall \rho.\ [\![t_2]\!]\rho = [\![n_2]\!]\rho = \lfloor n_2 \rfloor$.

We have $[\![t_1 \text{ op } t_2]\!]\rho = [\![t_1]\!]\rho\ \underline{op}_\perp\ [\![t_2]\!]\rho = \lfloor n_1 \rfloor\ \underline{op}_\perp \lfloor n_2 \rfloor = \lfloor n_1 \underline{op}\ n_2 \rfloor = \left[\!\left[n_1\ \underline{op}\ n_2\right]\!\right]\rho$.

$$\frac{t_0 \to 0 \quad t_1 \to c_1}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to c_1}$$

We will prove $P(\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to c_1) \equiv \forall \rho.\ [\![\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2]\!]\rho = [\![c_1]\!]\rho$.

We assume $P(t_0 \to 0) \equiv \forall \rho.\ [\![t_0]\!]\rho = [\![0]\!]\rho = \lfloor 0 \rfloor$ and $P(t_1 \to c_1) \equiv \forall \rho.\ [\![t_1]\!]\rho = [\![c_1]\!]\rho$.

We have $[\![\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2]\!]\rho = Cond([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho) = Cond(\lfloor 0 \rfloor, [\![t_1]\!]\rho, [\![t_2]\!]\rho) = [\![t_1]\!]\rho = [\![c_1]\!]\rho$.

The same construction holds for the second rule of the conditional operator.

$$\frac{t \to (t_1, t_2) \quad t_1 \to c_1}{\textbf{fst}(t) \to c_1}$$

We prove $P(\textbf{fst}(t) \to c_1) \equiv \forall \rho.\ [\![\textbf{fst}(t)]\!]\rho = [\![c_1]\!]\rho$.

We assume $P(t \to (t_1, t_2)) \equiv \forall \rho.\ [\![t]\!]\rho = [\![(t_1, t_2)]\!]\rho$ and $P(t_1 \to c_1) \equiv \forall \rho.\ [\![t_1]\!]\rho = [\![c_1]\!]\rho$.

By definition $[\![\mathbf{fst}(t)]\!]\rho = \mathbf{let}\, v \Leftarrow [\![t]\!]\rho.\ \pi_1 v$.

By inductive hypothesis $\mathbf{let}\, v \Leftarrow [\![t]\!]\rho.\ \pi_1 v = \mathbf{let}\, v \Leftarrow [\![(t_1,t_2)]\!]\rho.\ \pi_1 v$.

By definition $\mathbf{let}\, v \Leftarrow [\![(t_1,t_2)]\!]\rho.\ \pi_1 v = \mathbf{let}\, v \Leftarrow \lfloor [\![t_1]\!]\rho, [\![t_2]\!]\rho \rfloor.\ \pi_1 v = [\![t_1]\!]\rho = [\![c_1]\!]\rho$.

The same holds for the **snd** operator.

$$\frac{t_1 \to \lambda x.t_1' \qquad t_1'[^{t_2}/_x] \to c}{(t_1 \quad t_2) \to c}$$

We prove $P((t_1 \quad t_2) \to c) \equiv \forall \rho.\ [\![(t_1 \quad t_2)]\!]\rho = [\![c]\!]\rho$.

We assume $P(t_1 \to \lambda x.\ t_1') \equiv \forall \rho.\ [\![t_1]\!]\rho = \left[\!\!\left[\lambda x.\ t_1'\right]\!\!\right]\rho$ and $P(t_1'[^{t_2}/_x] \to c) \equiv \forall \rho.\ \left[\!\!\left[t_1'[^{t_2}/_x]\right]\!\!\right]\rho = [\![c]\!]\rho$.

By definition $[\![(t_1 \quad t_2)]\!]\rho = \mathbf{let}\, \varphi \Leftarrow [\![t_1]\!]\rho.\ \varphi([\![t_2]\!]\rho)$.

By inductive hypothesis $\mathbf{let}\, \varphi \Leftarrow [\![t_1]\!]\rho.\ \varphi([\![t_2]\!]\rho) = \mathbf{let}\, \varphi \Leftarrow \left[\!\!\left[\lambda x.t_1'\right]\!\!\right]\rho.\ \varphi([\![t_2]\!]\rho)$.

By denotational semantics:

$$\begin{aligned}
\mathbf{let}\, \varphi \Leftarrow [\![\lambda x.t_1']\!]\rho.\ \varphi([\![t_2]\!]\rho) \quad &= \quad \mathbf{let}\, \varphi \Leftarrow \lfloor \lambda d.\ [\![t_1']\!]\rho[^d/_x] \rfloor.\ \varphi([\![t_2]\!]\rho) \\
&= \quad (\lambda d.\ [\![t_1']\!]\rho[^d/_x])([\![t_2]\!]\rho) \\
&= \quad [\![t_1']\!]\rho[^{[\![t_2]\!]\rho}/_x].
\end{aligned}$$

Finally, by using the substitution lemma and by the second inductive hypothesis we have:

$$[\![t_1']\!]\rho[^{[\![t_2]\!]\rho}/_x] = \left[\!\!\left[t_1'[^{t_2}/_x]\right]\!\!\right]\rho = [\![c]\!]\rho.$$

$$\frac{t[^{\mathbf{rec}\, x.t}/_x] \to c}{\mathbf{rec}\, x.t \to c}$$

We prove $P(\mathbf{rec}\, x.t \to c) \equiv \forall \rho.\ [\![\mathbf{rec}\, x.t]\!]\rho = [\![c]\!]\rho$.

We assume $P(t[^{\mathbf{rec}\, x.t}/_x] \to c) \equiv \forall \rho.\ [\![t[^{\mathbf{rec}\, x.t}/_x]]\!]\rho = [\![c]\!]\rho$.

By definition we have $[\![\mathbf{rec}\, x.t]\!]\rho = \mathrm{fix}\ \lambda d.\ [\![t]\!]\rho[^d/_x]$.

By the fixpoint property $\mathrm{fix}\ \lambda d.\ [\![t]\!]\rho[^d/_x] = [\![t]\!]\rho[^{[\![\mathbf{rec}\, x.t]\!]\rho}/_x]$.

By the substitution lemma and by induction hypothesis $[\![t]\!]\rho[^{[\![\mathbf{rec}\, x.t]\!]\rho}/_x] = [\![t[^{\mathbf{rec}\, x.t}/_x]]\!]\rho = [\![c]\!]\rho$.

□

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:

$$t \downarrow \Longrightarrow t \Downarrow$$

Proof. If $t \longrightarrow c$, then $[\![t]\!]\rho = [\![c]\!]\rho$ for the previous theorem. But $[\![c]\!]\rho$ is always a lifted value, and thus it is not $\perp$.

□

We give some insight on the reason why the usual structural induction does not work for proving $t \Downarrow \Longrightarrow t \downarrow$. Let us consider function application $(t_1\ t_2)$. We assume by structural induction $t_1 \Downarrow \Rightarrow t_1 \downarrow$ and $t_2 \Downarrow \Rightarrow t_2 \downarrow$. 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 $\llbracket (t_1\ t_2) \rrbracket \rho \neq \bot$ requires $\llbracket t_1 \rrbracket \rho \neq \bot$. 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:

$$\begin{aligned}
\llbracket (t_1\ t_2) \rrbracket \rho &= \mathbf{let}\, \varphi \Leftarrow \lfloor \lambda d. \llbracket t_1' \rrbracket \rho[^d/_x] \rfloor .\ \varphi(\llbracket t_2 \rrbracket \rho) \\
&= (\lambda d. \llbracket t_1' \rrbracket \rho[^d/_x])(\llbracket t_2 \rrbracket \rho) \\
&= (\llbracket t_1' \rrbracket \rho[^{\llbracket t_2 \rrbracket \rho}/_x]) \\
&= (\llbracket t_1'[^{t_2}/_x] \rrbracket \rho)
\end{aligned}$$

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:

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[^{t_2}/_x] \to c}{(t_1 \quad t_2) \to c}$$

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:

$$t_1 \equiv_{op} t_2 \Longleftrightarrow (t_1 \uparrow\ \wedge t_2 \uparrow) \vee (t_1 \to c\ \wedge\ t_2 \to c)$$

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:

$$t_1 \equiv_{den} t_2 \Longleftrightarrow \forall \rho.\ \llbracket t_1 \rrbracket \rho = \llbracket t_2 \rrbracket \rho$$

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 then the operational one. Note that if we assume $t_1 \equiv_{den} t_2$ and $t_1, t_2 \neq \bot$ then we can only conclude that $t_1 \longrightarrow c_1$ and $t_2 \longrightarrow c_2$ 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:

$$\llbracket t \rrbracket \rho = \lfloor n \rfloor \Longleftrightarrow t \longrightarrow n$$

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 Theorem9.4, 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:

$$D_{int} = \mathbf{N}_\perp$$

$$D_{\tau_1 * \tau_2} = D_{\tau_1} \times D_{\tau_2}$$

$$D_{\tau_1 \to \tau_2} = [D_{\tau_1} \to D_{\tau_2}]$$

So we can simply define the interpretation function as follows:

$$\begin{array}{ll}
\llbracket n \rrbracket \rho & = \lfloor n \rfloor \\
\llbracket x \rrbracket \rho & = \rho\, x \\
\llbracket t_1 \ op \ t_2 \rrbracket \rho & = \llbracket t_1 \rrbracket \rho \ \underline{op}_\perp \ \llbracket t_2 \rrbracket \rho \\
\llbracket if \ t_0 \ then \ t_1 \ else \ t_2 \rrbracket \rho & = Cond(\llbracket t_0 \rrbracket \rho, \llbracket t_1 \rrbracket \rho, \llbracket t_2 \rrbracket \rho) \\
\llbracket (t_1, t_2) \rrbracket \rho & = (\llbracket t_1 \rrbracket \rho, \llbracket t_2 \rrbracket \rho) \\
\llbracket \mathbf{fst}(t) \rrbracket \rho & = \pi_1(\llbracket t \rrbracket \rho) \\
\llbracket \mathbf{snd}(t) \rrbracket \rho & = \pi_2(\llbracket t \rrbracket \rho) \\
\llbracket \lambda x.t \rrbracket \rho & = \lambda d.\, \llbracket t \rrbracket \rho[^d/_x] \\
\llbracket (t_1 \ t_2) \rrbracket \rho & = (\llbracket t_1 \rrbracket \rho)(\llbracket t_2 \rrbracket \rho) \\
\llbracket rec x.t \rrbracket \rho & = \mathrm{fix}\, \lambda d.\, \llbracket t \rrbracket \rho[^d/_x]
\end{array}$$

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 then the previous and does not express some interesting properties. For instance, consider the two HOFL terms:

$$t_1 = rec\ x.x : int \longrightarrow int \quad \text{and} \quad t_2 = \lambda x.rec\ y.y : int \longrightarrow int$$

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 $t : int \longrightarrow int$, 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.

Part III.

Concurrency and Logic

10. CCS, the Calculus for Communicating Systems

In the last decade computer science technologies have boosted the growth of large scale distributed and concurrent systems. Their formal study introduces several aspects which are not present in the case of sequential systems. In particular, it emerges the necessity to deal with non-determinism, parallelism, interaction and infinite behaviour. Non-determinism is needed to model time races between different signals and to abstract from programming details which are irrelevant for the interaction behaviour of systems. Parallelism allows agents to perform tasks independently. For our purposes, this will be modelled by using non-determinism. Interaction allows to describe the behaviour of the system from the observational point of view (i.e., the behaviour that the system shows to an external observer). Infinite behaviour allows to study the semantics of non-terminating processes useful in many different contexts (e.g., think about the modelling of operating systems). Accordingly, from the theoretical point of view, some additional efforts must be spent to extend the semantics of sequential systems to that of concurrent systems in a proper way.

As we saw in the previous chapters, the study of sequential programming languages brought to different semantics which allows to prove many different properties. In this chapter we introduce CCS, a specification language which allows to describe concurrent communicating systems. Such systems are composed of agents (i.e., processes) performing tasks by communicating each other through channels.

While infinite behaviour is accounted for also in IMP and HOFL (consider, e.g., the programs **rec** $x.\ x$ and **while** `true` **do** **skip**), unlike the sequential languages, CCS does not assign the same semantics to all the infinite behaviours (recall that if a sequential program does not terminate its semantics is equal to $\perp$ in the denotational semantics).

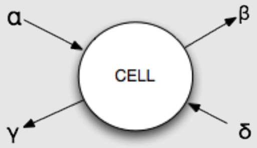
The semantics of sequential languages can be given by defining functions. In the presence of non-deterministic behaviours functions do not seem to provide the right tool to abstract the behaviour of concurrent systems. As we will see this problem is worked out by modelling the system behaviour as a *labelled transition system*, i.e. as a set of states equipped with a transition relation which keeps track of the interactions between the system and its environment. As a consequence, it makes little sense to talk about denotational semantics of CCS. In addition, recall that the denotational semantics is based on fix point theory over CPOs, while it turns out that several interesting properties of non-deterministic systems with non-trivial infinite behaviours are not inclusive (as it is the case of fairness, described in Example 5.14), thus the principle of computational induction does not apply to such properties. Moreover labelled transition systems are often equipped with a modal logic counterpart, which allows to express and prove the relevant properties of the modelled system.

Let us show how CCS works with an example.

Example 10.1 (Dynamic concurrent stack)
Let us consider the problem of modelling an extensible stack. The idea is to represent the stack as a collection of cells that communicate by sending and receiving data over some channels:

- *the* send *operation of data x over channel α is denoted by $\bar{\alpha}x$;*
- *the* receive *operation of data x over channel α is denoted by αx.*

We have one so-called process *(or* agent*) for each cell of the stack. Each process can store one or two values or send a stored value to other processes. All processes involved in the stack have basically the same structure. We represent graphically one of such processes as follow:*



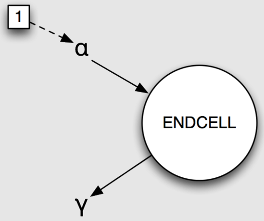
The figure shows that a cell has four channels $\alpha, \beta, \gamma, \delta$ *that can be used to communicate with other cells. In general, a process can perform bidirectional operation on its channels. In this particular case, each cell will use each channel for either input or output operations:*

- α *is the input channel to receive data from either the external environment or the left cell;*
- γ *is the channel used to send data to either the external environment or the left cell;*
- β *is the channel used to send data to the right cell and to manage the end of the stack;*
- δ *is the channel used to receive data from the right cell and to manage the end of the stack.*

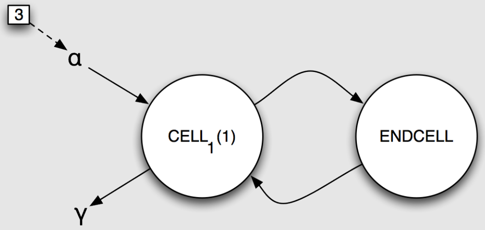
In the following we specify the possible states that a cell can have, each corresponding to some specific behaviour. Note that some states are parametric to certain values that represent, e.g., the particular values stored in that cell. The four possible states are described below:

- $CELL_0 = \delta x.\,\textbf{if}\ x = \$\ \textbf{then}\ ENDCELL\ \textbf{else}\ CELL_1(x)$
  This state represents the empty cell. The agent waits for data from the channel δ*, when a value is received the agent controls if it is equal to a special termination character* \$. *If the received data is* \$ *this means that the agent is the last cell, so it switches to the* $ENDCELL$ *state. Otherwise, if* x *is a new value, the agent stores it by moving to the state* $CELL_1(x)$.
- $CELL_1(y) = \alpha x.CELL_2(x,y) + \bar{\gamma} y.CELL_0$
  This state represents an agent which contains a value y*. In this case the cell can non-deterministically wait for new data on* α *or send the stored data on* γ*. In the first case, the cell must store the new value and send the old value to another agent: this task is performed by* $CELL_2(x,y)$*. In the second case, it is assumed that some other agent wants to extract the stored value; then the cell becomes empty by switching to the state* $CELL_0$*. Note that the* $+$ *operator represents a non-deterministic choice performed by the agent. However a particular choice could be forced on a cell by the behaviour of the other cells.*
- $CELL_2(x,y) = \bar{\beta} y.CELL_1(x)$
  In this case the cell has currently two parameters x *(the newly received value) and* y *(the previously stored value). The agent must cooperate with its neighbors cells in order to perform a right shift of the data. In order to do that the agent communicates to the right neighbour the old stored value* y *and moves to state* $CELL_1(x)$.
- $ENDCELL = \alpha x.(CELL_1(x)\ \underbrace{\smile ENDCELL)}_{\text{a new bottom cell}} + \bar{\gamma}\$.\underbrace{\textbf{nil}}_{\text{termination}}$
  This state represents the bottom of the stack. An agent in this state can perform two actions in a non-deterministic way. First it can wait for a new value (in order to perform a right shift), then store the new data and generate a new agent witch represents the new bottom element. Note that the newly created cell $ENDCELL$ *will be able to communicate with* $CELL_1(x)$ *only, because they will have dedicated channels. We will explain later how this can be achieved, when giving the exact definition of the operation* $\smile$*. Alternatively, the agent can send the special character* \$ *to the left cell, provided it is able to receive this character. If so, than the left cell is empty and after receiving the* \$ *character it becomes the new* $ENDCELL$*. Then the present agent terminates.*

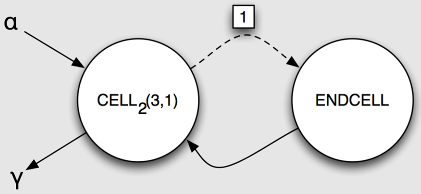
Now we will show how the stack works. Let us start from an empty stack. We have only one cell in the state $ENDCELL$*, whose channels* β *and* δ *are made private, written* $ENDCELL \backslash \beta \backslash \delta$*, because on the "right" side there will be no communication. We perform a push operation in order to fill the stack with a new value. So we send the new value (1 in this case) through the channel* α *of the cell.*



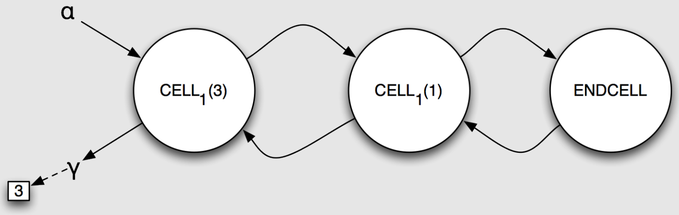
Once the cell receives the new value it generates a new bottom for the stack and changes its state to $CELL_1(1)$ *storing the new value. The result of this operation is the following:*



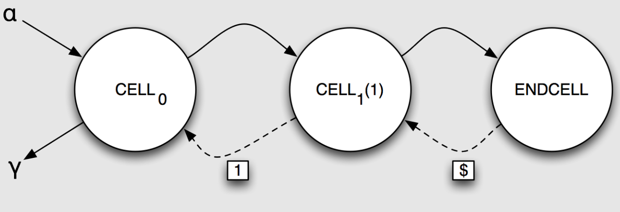
When the stack is stabilized we perform another push, this time with value 3*. In this case the first cell changes its state to* $CELL_2(3,1)$ *in order to perform a right shift.*



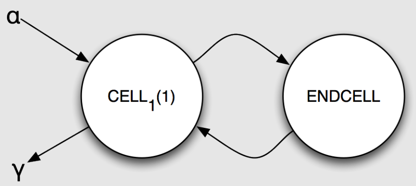
Then, when the second cell receives the value 1 *on his* α' *channel changing its state to* $CELL_1(1)$*, the first cell can stabilize itself on the state* $CELL_1(3)$*. Now we perform a pop operation, which will return the last value pushed into the stack (i.e.* 3*).*



In order to do this we read the value 3 *from the channel* γ *of the first cell. In this case the first cell changes its state to* $CELL_0$*, waiting for a value through the channel* δ*.*



When the second cell become aware of the reading performed by the first cell, it changes its state to $CELL_0$*, and reads the value sent from the third cell. Then, since the received value from* $ENDCELL$ *is* \$*, it changes its state to* $ENDCELL$*. Finally, since a reading operation on* γ'' *have been performed by the second cell, the third cell reduces to* **nil***. The situation reached after the stabilization of the system is the following:*



The above example shows that processes can synchronize in pairs, by performing dual (input/output) operations. In the following we will present a *pure* version of CCS, where we abstract away from the values communicated on channels.

10.1. Syntax of CCS

The CCS process algebra was introduced by Robin Milner in the early eighties. When presenting the syntax of CCS we will use the following conventions:

$\Delta ::= \alpha, \beta, ...$	Channels and (by coercion) input actions on channels
$\overline{\Delta} ::= \overline{\alpha}, \overline{\beta}, ...$ with $\overline{\Delta} \cap \Delta = \emptyset$	Output actions on channels
$\Lambda ::= \Delta \cup \overline{\Delta}$	Observable actions
$\tau \notin \Lambda$	Unobservable action

We extend the "bar" operation to all the elements in Λ by letting $\overline{\overline{\alpha}} = \alpha$ for all $\alpha \in \Delta$. As we have seen in the example, pairs of dual actions (e.g., α and $\overline{\alpha}$) are used to synchronize two processes. The unobservable action τ denotes a special action that is internal to some agent and that cannot be used to synchronize. Moreover we will use the following convention:

$\mu \in \Lambda \cup \{\tau\}$	generic action
$\lambda \in \Lambda$	generic channel
$\overline{\lambda} \in \Lambda$	generic dual channel

Now we are ready to present the syntax of CCS.

$$p, q ::= x \quad | \quad \mathbf{nil} \quad | \quad \mu.p \quad | \quad p\backslash\alpha \quad | \quad p[\phi] \quad | \quad p + q \quad | \quad p \mid q \quad | \quad \mathbf{rec}\, x.p$$

x represents a process name;

nil is the empty (*inactive*) process;

$\mu.p$ is a process p *prefixed* by the action μ;

$p\backslash\alpha$ is a *restricted* process; it allows to hide the channel α from an observer external to p;

$p[\phi]$ is a process that behaves like the process obtained from p by applying to it the permutation (a bijective substitution) ϕ of its channel names. However this operation is part of the syntax and $p[\phi]$ is syntactically different than p with the substitution performed on it. Notice that $\phi(\tau) = \tau$;

$p + q$ is a process that can choose non-deterministically to execute either the process p or q;

$p|q$ is the process obtained as the parallel composition of p and q; the actions of p and q can be interleaved and also synchronized;

rec $x.p$ is a recursively defined process.

As usual we will consider the closed terms of this language, i.e., the processes whose process names x are all bound by recursive definitions.

10.2. Operational Semantics of CCS

The operational semantics of CCS is defined by a suitable *Labelled Transition System* (LTS) whose states are CCS (closed) processes and whose transitions are labelled by actions in $\Lambda \cup \{\tau\}$. The LTS is defined by a rule system whose formulas take the form $p_1 \xrightarrow{\mu} p_2$ meaning that the process p_1 can perform the action μ and reduce to p_2. While the LTS is the same for all CCS closed terms, starting from a CCS closed term p and using the rules we can define the LTS which represents the operational behaviour of p by considering only processes that are reachable from the state p. Although a term can be the parallel composition of many processes, its operational semantics is represented by a single global state in the LTS. Therefore concurrency and interaction between cooperating agents are not adequately represented in our CCS semantics. Now we introduce the inference rules for CCS:

$$\text{(Act)} \quad \frac{}{\mu.p \xrightarrow{\mu} p}$$

There is only one axiom in the rule system, related to the action prefix operator. It states that the process $\mu.p$ can perform the action μ and reduce to p.

$$\text{(Res)} \quad \frac{p \xrightarrow{\mu} q}{p\backslash\alpha \xrightarrow{\mu} q\backslash\alpha}\ \mu \neq \alpha, \bar{\alpha}$$

If the process is executed under a restriction, then it can perform only actions that do not involve the restricted name. Note that this restriction does not affect the communication internal to the processes, i.e., when $\mu = \tau$ the move cannot be blocked by the restriction.

$$\text{(Rel)} \quad \frac{p \xrightarrow{\mu} q}{p[\phi] \xrightarrow{\phi(\mu)} q[\phi]}$$

For ϕ a permutation of channel names, if p can evolve to q by performing μ, then $p[\phi]$ can evolve to $q[\phi]$ by performing $\phi(\mu)$, i.e., the action μ renamed according to ϕ. We remind that the unobservable action cannot be renamed, i.e., $\phi(\tau) = \tau$ for any ϕ.

$$\text{(Sum)} \quad \frac{p \xrightarrow{\mu} p'}{p + q \xrightarrow{\mu} p'} \qquad \frac{q \xrightarrow{\mu} q'}{p + q \xrightarrow{\mu} q'}$$

This pair of rules deals with non-deterministic choice: process $p + q$ can choose non-deterministically to behave like either process p or q. Moreover note that the choice can be performed only during communication, so in order to discard, e.g., process q, process p must be capable to perform an action μ. Note that axioms of the form $p + q \rightarrow p$ and $p + q \rightarrow q$ would yield a rather different semantics.

$$\text{(Com)} \quad \frac{p \xrightarrow{\mu} p'}{p|q \xrightarrow{\mu} p'|q} \qquad \frac{q \xrightarrow{\mu} q'}{p|q \xrightarrow{\mu} p|q'}$$

Also in the case of parallel composition some form of non-determinism appears. But unlike the previous case, here non-determinism is needed to simulate the parallel behaviour of the system: in the previous rule non-determinism was a characteristic of the modelled system, in this case it is a characteristic of the semantic style that allows p and q to interleave their actions in $p|q$.

$$\frac{p_1 \xrightarrow{\lambda} p_2 \quad q_1 \xrightarrow{\overline{\lambda}} q_2}{p_1|q_1 \xrightarrow{\tau} p_2|q_2}$$

There is a third rule for parallel composition, which allows processes to perform internal synchronizations. The processes p_1 and p_2 communicate by using the channel λ, which is hidden after the synchronization by using the action τ. In general, if p_1 and p_2 can perform α and $\overline{\alpha}$, respectively, then their parallel composition can perform α, $\overline{\alpha}$ or τ. When parallel composition is used in combination with the restriction operator, like in $(p_1|p_2)\backslash\alpha$, then we can force synchronization on α.

$$(\text{Rec}) \qquad \frac{p[\mathbf{rec}\, x.p/x] \xrightarrow{\mu} q}{\mathbf{rec}\, x.p \xrightarrow{\mu} q}$$

The semantics of recursion is similar to the one we have presented for HOFL: to see which moves $\mathbf{rec}\, x.p$ can perform we inspect the process $p[\mathbf{rec}\, x.p/x]$ obtained from p by replacing all free occurrences of the process name x with its full recursive definition $\mathbf{rec}\, x.p$.

We will restrict our attention to the class of *guarded agents*, namely agents in which in case of recursive terms of the form $\mathbf{rec}\, x.p$, each free occurrence of x in p occurs under an action prefix. This allows us to exclude terms like $\mathbf{rec}\, x.(x \mid p)$ which can lead (in one step) to infinitely many parallel repetitions of the same agent, making the LTS infinitely branching.

Example 10.2 (Derivation)

Let us show an example of the use of the derivation rules which we have just introduced. Take the following CCS term:

$$(((\mathbf{rec}\, x.\ \alpha.x + \beta.x) \mid (\mathbf{rec}\, x.\ \alpha.x + \gamma.x)) \mid \mathbf{rec}\, x.\ \bar{\alpha}.x)\backslash\alpha$$

First, let us focus on the behaviour of the (deterministic) agent $\mathbf{rec}\, x.\ \bar{\alpha}.x$*.*

$$\begin{array}{rl} \mathbf{rec}\, x.\ \bar{\alpha}.x \xrightarrow{\bar{\alpha}} q & \nwarrow_{Rec} \\ \bar{\alpha}.(\mathbf{rec}\, x.\ \bar{\alpha}.x) \xrightarrow{\bar{\alpha}} q & \nwarrow_{Act,\ q=\mathbf{rec}\, x.\ \bar{\alpha}.x} \\ \square & \end{array}$$

Thus:

$$\mathbf{rec}\, x.\ \bar{\alpha}.x \xrightarrow{\bar{\alpha}} \mathbf{rec}\, x.\ \bar{\alpha}.x$$

There are no other rules applicable during the above derivation; thus, the LTS associated with $\mathbf{rec}\, x.\ \bar{\alpha}.x$ *consists of a single state and one looping arrow with label* $\bar{\alpha}$*. Correspondingly, the agent is able to perform the action* $\bar{\alpha}$ *indefinitely. However, when embedded in the larger system above, then the action* $\bar{\alpha}$ *is blocked by the topmost restriction* $\_\ \backslash\alpha$*. Therefore, the only opportunity for* $\mathbf{rec}\, x.\ \bar{\alpha}.x$ *to act is by synchronizing on channel* α *with either one or the other of the two non-deterministic agents* $\mathbf{rec}\, x.\ \alpha.x{+}\beta.x$ *and* $\mathbf{rec}\, x.\ \alpha.x{+}\gamma.x$*. In fact the synchronization produces an action* τ *which cannot be blocked by* $\_\ \backslash\alpha$*. Note that each of the two non-deterministic agents is also available to interact with some external agent on another non-restricted channel, respectively* β *or* γ*.*

By using the rules of the operational semantics of CCS we have, e.g.:

$$
\begin{array}{rl}
(((\mathbf{rec}\,x.\ \alpha.x+\beta.x) \mid (\mathbf{rec}\,x.\ \alpha.x+\gamma.x)) \mid \ \mathbf{rec}\,x.\ \bar{\alpha}.x) \setminus \alpha \xrightarrow{\mu} q & \nwarrow_{\mathit{Res},\ q=q'\setminus\alpha} \\
((\mathbf{rec}\,x.\ \alpha.x+\beta.x) \mid (\mathbf{rec}\,x.\ \alpha.x+\gamma.x)) \mid \ \mathbf{rec}\,x.\ \bar{\alpha}.x \xrightarrow{\mu} q', \mu \neq \alpha, \overline{\alpha} & \nwarrow_{\mathit{Com\ 3rd\ rule},\ \mu=\tau, q'=q_1 \mid q_2} \\
(\mathbf{rec}\,x.\ \alpha.x+\beta.x) \mid \ \mathbf{rec}\,x.\ \alpha.x+\gamma.x \xrightarrow{\lambda} q_1, \quad \mathbf{rec}\,x.\ \bar{\alpha}.x \xrightarrow{\bar{\lambda}} q_2 & \nwarrow_{\mathit{Com\ 2nd\ rule},\ q_1=(\mathbf{rec}\,x.\ \alpha.x+\beta.x) \mid q_3} \\
\mathbf{rec}\,x.\ \alpha.x+\gamma.x \xrightarrow{\lambda} q_3, \quad \mathbf{rec}\,x.\ \bar{\alpha}.x \xrightarrow{\bar{\lambda}} q_2 & \nwarrow_{\mathit{Rec}} \\
\alpha.(\mathbf{rec}\,x.\ \alpha.x+\gamma.x) + \gamma.(\mathbf{rec}\,x.\ \alpha.x+\gamma.x) \xrightarrow{\lambda} q_3, \quad \mathbf{rec}\,x.\ \bar{\alpha}.x \xrightarrow{\bar{\lambda}} q_2 & \nwarrow_{\mathit{Sum}} \\
\alpha.(\mathbf{rec}\,x.\ \alpha.x+\gamma.x) \xrightarrow{\lambda} q_3, \quad \mathbf{rec}\,x.\ \bar{\alpha}.x \xrightarrow{\bar{\lambda}} q_2 & \nwarrow_{\mathit{Act},\ q_3=\mathbf{rec}\,x.\ \alpha.x+\gamma.x,\ \lambda=\alpha} \\
\mathbf{rec}\,x.\ \bar{\alpha}.x \xrightarrow{\bar{\alpha}} q_2 & \nwarrow_{\mathit{Rec}} \\
\bar{\alpha}.(\mathbf{rec}\,x.\ \bar{\alpha}.x) \xrightarrow{\bar{\alpha}} q_2 & \nwarrow_{\mathit{Act},\ q_2=\mathbf{rec}\,x.\ \bar{\alpha}.x} \\
\square &
\end{array}
$$

So we have:

$$
\begin{aligned}
q_2 &= \mathbf{rec}\,x.\ \bar{\alpha}.x \\
q_3 &= \mathbf{rec}\,x.\ \alpha.x+\gamma.x \\
q_1 &= (\mathbf{rec}\,x.\ \alpha.x+\beta.x) \mid q_3 = (\mathbf{rec}\,x.\ \alpha.x+\beta.x) \mid \ \mathbf{rec}\,x.\ \alpha.x+\gamma.x \\
q' &= q_1 \mid q_2 = ((\mathbf{rec}\,x.\ \alpha.x+\beta.x) \mid (\mathbf{rec}\,x.\ \alpha.x+\gamma.x)) \mid \ \mathbf{rec}\,x.\ \bar{\alpha}.x \\
q &= q'\setminus\alpha = (((\mathbf{rec}\,x.\ \alpha.x+\beta.x) \mid (\mathbf{rec}\,x.\ \alpha.x+\gamma.x)) \mid \ \mathbf{rec}\,x.\ \bar{\alpha}.x)\setminus\alpha \\
\mu &= \tau
\end{aligned}
$$

and thus:

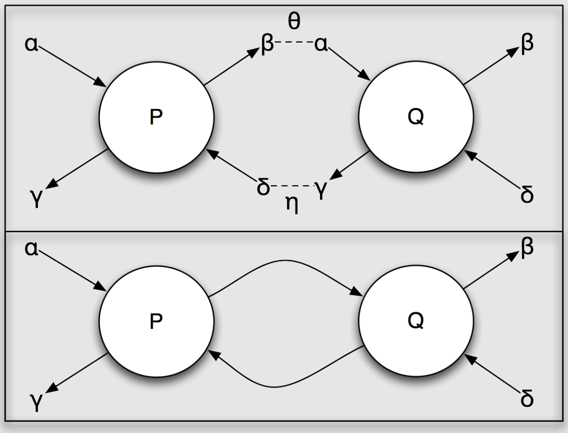
$$
(((\mathbf{rec}\,x.\ \alpha.x+\beta.x) \mid (\mathbf{rec}\,x.\ \alpha.x+\gamma.x)) \mid \ \mathbf{rec}\,x.\ \bar{\alpha}.x)\setminus\alpha \xrightarrow{\tau} (((\mathbf{rec}\,x.\ \alpha.x+\beta.x) \mid (\mathbf{rec}\,x.\ \alpha.x+\gamma.x)) \mid \ \mathbf{rec}\,x.\ \bar{\alpha}.x)\setminus\alpha
$$

Note that during the derivation we had to choose several times between different rules which could be applied; while in general it may happen that wrong choices can lead to dead ends, our choices have been made so to complete the derivation satisfactorily, avoiding any backtracking.

Example 10.3 (Dynamic stack continuation)
Let us conclude the dynamic stack example by formalizing in CCS the concatenation operator:

$$
P \frown Q = (P[^{\vartheta}/_{\beta}, ^{\eta}/_{\delta}] \mid Q[^{\vartheta}/_{\alpha}, ^{\eta}/_{\gamma}])\setminus\vartheta\setminus\eta
$$

where ϑ and η are two new hidden channels shared by the processes. Note that the locality of these names allows to avoid conflicts between channel's names.



10.3. Abstract Semantics of CCS

As we saw each CCS agent can be represented by a LTS, i.e., by a labelled graph. It is easy to see that such operational semantics is much more concrete and detailed than the semantics studied for IMP and HOFL. For example, since the states of the LTS are named by agents it is evident that two syntactically different processes like $p|q$ and $q|p$ are associated with different graphs, even if intuitively one would expect that both exhibit the same behaviour. Thus it is important to find a good notion of equivalence, able to provide a more abstract semantics for CCS. As it happens for the denotational semantics of IMP and HOFL, an abstract semantics defined up to equivalence should abstract from the way agents execute, focusing on their external visible behaviours.

In this section we first show that neither graph isomorphism nor trace semantics are fully satisfactorily abstract semantics to capture the features of communicating systems represented in CCS. Next, we introduce a more satisfactory semantics of CCS by defining a relation, called *bisimilarity*, that captures the ability of processes to simulate each other. Finally, we discuss some positive and negative aspects of bisimilarity and present some possible alternatives.

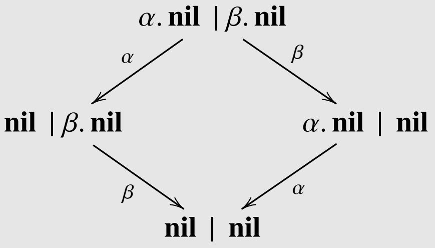
10.3.1. Graph Isomorphism

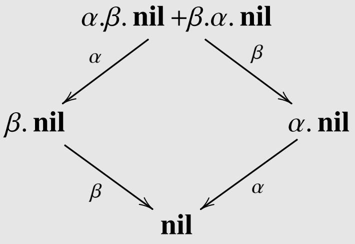
It is quite obvious to think that two agents must be considered as equivalent if their (LTS) graphs are isomorphic. Recall that two graphs are said to be isomorphic if there exists a bijection f which preserves the structure of the graphs.

Example 10.4 (Isomorphic agents)
Let us consider the following agents:

$$\alpha.\,\mathbf{nil}\ \mid \beta.\,\mathbf{nil} \qquad \alpha.\beta.\,\mathbf{nil} + \beta.\alpha.\,\mathbf{nil}$$

Their reachable parts of the LTS are as follows:


$\alpha.\,\mathbf{nil}\ \mid \beta.\,\mathbf{nil}$
α
β
$\mathbf{nil}\ \mid \beta.\,\mathbf{nil}$
$\alpha.\,\mathbf{nil}\ \mid\ \mathbf{nil}$
β
α
$\mathbf{nil}\ \mid\ \mathbf{nil}$


$\alpha.\beta.\,\mathbf{nil} + \beta.\alpha.\,\mathbf{nil}$
α
β
$\beta.\,\mathbf{nil}$
$\alpha.\,\mathbf{nil}$
β
α
$\mathbf{nil}$

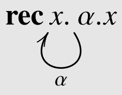
The two graphs are isomorphic, thus the two agents should be considered as equivalent. This result is surprising, since they have a rather different structure. In fact, the example shows that concurrency can be reduced to non-determinism by graph isomorphism. This is due to the interleaving of the actions performed by processes that are composed in parallel, which is a characteristic of the semantics which we have presented.

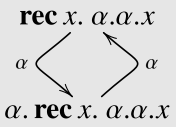
This approach is very simple and natural but still leads to semantics that is too concrete, i.e., graph isomorphism still distinguishes too much. We show this fact in the following example.

Example 10.5
Let us consider the agents:

$$\mathbf{rec}\,x.\ \alpha.x \qquad \mathbf{rec}\,x.\ \alpha.\alpha.x$$

Their reachable parts of the LTS are as follows:


$\mathbf{rec}\,x.\ \alpha.x$
α


$\mathbf{rec}\,x.\ \alpha.\alpha.x$
α
α
$\alpha.\,\mathbf{rec}\,x.\ \alpha.\alpha.x$

The two graphs are not isomorphic, but it is hardly possible to distinguish between the two agents *according to their behaviour: they both are able only to execute any sequence of α.*

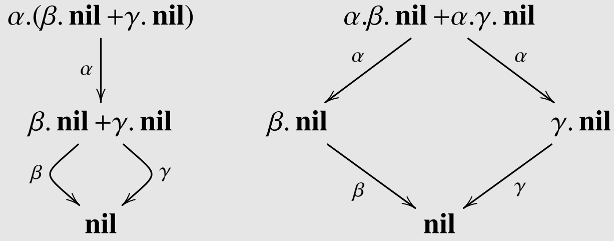
10.3.2. Trace Equivalence

A second approach, trace equivalence, observes the set of traces of an agent, namely the set of sequences of actions labelling any path in its graph. Trace equivalence is strictly coarser than equivalence based on graph isomorphism, since isomorphic graphs have the same traces. Conversely, Example 10.5 shows two agents which are trace equivalent but whose graphs are not isomorphic. In fact, trace equivalence is too coarse: the following example shows that trace equivalence is not able to capture the choice points within agent behaviour.

Example 10.6
Let us consider the following agents:

$$p = \alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}) \qquad q = \alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil}$$

Their reachable parts of the LTS are as follows:



These two graphs are trace equivalent: the trace sets are both $\{\alpha\beta, \alpha\gamma\}$. However the agents do not behave in the same way if we regard the choices they make. In the second agent the choice between β and γ is made during the first step, by selecting one of the two possible α. In the first agent, on the contrary, the same choice is made in a second time, after the execution of the unique α action.

The difference is evident if we consider, e.g., that an agent $\bar{\alpha}.\bar{\beta}.\,\mathbf{nil}$ may be running in parallel, with actions α, β and γ restricted on top: the agent p is always able to carry out the complete interaction with $\bar{\alpha}.\bar{\beta}.\,\mathbf{nil}$, because after the synchronization on α is ready to synchronize on β; vice versa, the agent q is only able to carry out the complete interaction with $\bar{\alpha}.\bar{\beta}.\,\mathbf{nil}$ if the left choice is performed at the time of the first interaction on α, as otherwise $\gamma.\,\mathbf{nil}$ and $\bar{\beta}.\,\mathbf{nil}$ cannot interact. Formally, if we consider the context $C(\_) = (\_ \mid \overline{\alpha}.\overline{\beta}.nil) \setminus \alpha \setminus \beta \setminus \gamma$ we have that $C(q)$ can deadlock, while $C(p)$ cannot. Figure out how embarrassing could be the difference if α would mean for a computer to ask if a file should be deleted, and β, γ were the user yes/no answer: p would behave as expected, while q could decide to delete the file in the first place, and then deadlock if the the user decides otherwise. See also Example 10.20.

Given all the above, we can argue that neither graph isomorphism nor trace equivalence are good candidates for our behavioural equivalence relation. Still, it is obvious that: 1) isomorphic agents must be retained as equivalent; 2) equivalent agents must be trace equivalent. Thus, our candidate equivalence relation must be situated in between graph isomorphism and trace equivalence.

10.3.3. Bisimilarity

In this section we introduce a class of relations between agents called *bisimulations* and we construct a behavioural equivalence relation between agents called *bisimilarity* as the largest bisimulation. This equivalence relation is shown to be the one we were looking for, namely the one that identifies only those agents which intuitively have the same behaviour.

Let us start with an example which illustrates how bisimilarity works.

Example 10.7 (Game Theory)
In this example we use game theory in order to show that the agents of the example 10.6 are not behaviourally

equivalent. In our example game theory is used in order to prove that a system verifies or not a property. We can imagine two player called Alice and Bob, the goal of Alice is to prove that the system has not the property. Bob, on the contrary, wants to show that the system satisfies the property. The game starts and at each turn each player can make a move in order to reach his/her goal. At the end of the game if Alice wins this means that the system does not satisfy the property. If the winner is Bob, instead, the system satisfies the property.

We apply this pattern to the states of LTS which describe CCS agents. Let us take two states p and q of a LTS. Alice would like to show that p is not behavioural equivalent to q, Bob on the other hand would like to show that p and q have the same behaviour.

Alice starts the game. At each turn of the game Alice executes (if possible) a transition of the transition system of either p or q and Bob must execute a transition with the same label but of the other agent. If Alice cannot move on both p and q, then Alice has lost, since this means that p and q are both deadlocked, and thus obviously equivalent. Alice wins if she can make a move that Bob cannot imitate; or if she has a move which, no matter which is the answer by Bob, will lead to a situation where she can make a move that Bob cannot imitate; or . . . and so on for any number of moves. Bob wins if Alice has no such a (finite) strategy. Note that the game does not necessarily terminate: also in this case Bob wins, that is p and q are equivalent.

In the example 10.6, let us take $p = \alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil})$ *and* $q = \alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil}$*. Alice starts by choosing p and by executing the only transition labelled α. Then, Bob can choose one of the two transitions labelled α leaving from q. Suppose that Bob chooses the left α transition (but the case where Bob chooses the right transition leads to the same result of the game). So the reached states are* $\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}$ *and* $\beta.\,\mathbf{nil}$*. In the second turn Alice chooses the transition labelled γ from* $\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}$*, and Bob can not simulate this execution. Since Alice has a winning, two-moves strategy, the two agents are not equivalent.*

Now we define the same relation in a more formal way, as originally introduced by Robin Milner. It is important to notice that the definition applies to a generic labelled transition systems, namely a set of states P equipped with a ternary relation $\longrightarrow \subseteq P \times L \times P$, where L is a generic set of actions. As we have seen, a unique LTS is associated to CCS, where CCS agents are states and a triple (p, α, q) belongs to $\longrightarrow$ iff $p \xrightarrow{\alpha} q$ is a theorem of the operational semantics. Notice that agents with isomorphic graphs are automatically equivalent.

Definition 10.8 (Strong Bisimulation)
Let R be a binary relation on the set of states of an LTS then it is a strong bisimulation *if*

$$\forall s_1 \; R \; s_2 \Rightarrow \quad \begin{array}{lll} \text{if } s_1 \xrightarrow{\alpha} s_1' & \text{then there exists a transition} & s_2 \xrightarrow{\alpha} s_2' \text{ such that } s_1' \; R \; s_2' \\ \text{if } s_2 \xrightarrow{\alpha} s_2' & \text{then there exists a transition} & s_1 \xrightarrow{\alpha} s_1' \text{ such that } s_1' \; R \; s_2' \end{array}$$

For example it is easy to check that the identity relation $\{(p, p) \mid p \text{ is a CCS process }\}$ is a strong bisimulation, that graph isomorphism is a strong bisimulation and that the union $R_1 \cup R_2$ of two strong bisimulation relations R_1 and R_2 is also a strong bisimulation relation. Moreover, given the composition of relations defined by

$$R_1 \circ R_2 \overset{\text{def}}{=} \{(p, p') \mid \exists p''.p \; R_1 \; p'' \wedge p'' \; R_2 \; p'\}$$

it can be shown that $R_1 \circ R_2$ is a strong bisimulation when R_1 and R_2 are such.

Definition 10.9 (Strong bisimilarity ≃)
Let s and s′ be two states of a LTS, then they are said to be bisimilar *and write* $s \simeq s'$ *if and only if there exists a strong bisimulation R such that* $s \; R \; s'$.

The relation ≃ is called strong bisimilarity *and is defined as follows:*

$$\simeq \overset{\text{def}}{=} \bigcup_{R \text{ is a strong bisimulation}} R$$

Strong bisimilarity $\simeq$ is an equivalence relation on CCS processes. Below we recall the definition of equivalence relation.

Definition 10.10 (Equivalence Relation)
Let $\equiv$ *be a binary relation on a set* X*, then we say that it is an* equivalence relation *if it has the following properties:*

- $\forall x, y \in X.\ x \equiv x$ *(Reflexivity)*
- $\forall x, y, z \in X.\ x \equiv y \wedge y \equiv z \Rightarrow x \equiv z$ *(Transitivity)*
- $\forall x, y \in X.\ x \equiv y \Rightarrow y \equiv x$ *(Symmetry)*

Definition 10.11 (Equivalence Class and Quotient Set)
Given an equivalence relation $\equiv$ *on* X *and an element* x *of* X *we call* equivalence class *of* x *the subset* $[x]$ *of* X *defined as follows:*

$$[x] = \{y \in X \mid x \equiv y\}$$

The set $X_{/\equiv}$ *containing all the equivalence classes generated by a relation* $\equiv$ *on the set* X *is called* quotient set.

We omit the proof of the following theorem that is based on the above mentioned properties of strong bisimulations and on the fact that bisimilarity is a strong bisimulation.

Theorem 10.12
The bisimilarity relation $\simeq$ *is an equivalence relation between CCS agents.*

Now we will use the fixpoint theory, which we have introduced in the previous chapters, in order to define bisimilarity in a more effective way. Using fixpoint theory we will construct, by successive approximations, the coarsest (maximal) bisimulation between the states of a LTS, which is actually an equivalence relation.

As usual, we define the $CPO_\perp$ on which the approximation function works. The $CPO_\perp$ is defined on the set $\mathscr{P}(P \times P)$, namely the power set of the pairs of states of the LTS. As we saw in the previous chapters the pair $(\mathscr{P}(P \times P), \subseteq)$ (all the subsets of a given set, ordered by inclusion) is a $CPO_\perp$, however it is not the one which we will use. As we said we would like to start from the roughest relation, which considers all the states equivalent and, by using the fixpoint operator, to reach the relation which will identify only bisimilar agents. So we need a $CPO_\perp$ in which a set with more elements is considered smaller than one with few elements. Thus we define the order relation $R \sqsubseteq R' \Leftrightarrow R' \subseteq R$ between subsets of $\mathscr{P}(P \times P)$. The resulting $CPO_\perp$ $(\mathscr{P}(P \times P), \sqsubseteq)$ is represented in figure 10.1 .

Now we define the function $\Phi : \mathscr{P}(P \times P) \to \mathscr{P}(P \times P)$ on relations on P.

$$p\ \Phi(R)\ q \overset{\text{def}}{=} \begin{cases} p \xrightarrow{\mu} p' & \text{implies} \quad \exists q'.\ q \xrightarrow{\mu} q' \quad \text{and} \quad p'Rq' \\ q \xrightarrow{\mu} q' & \text{implies} \quad \exists p'.\ p \xrightarrow{\mu} p' \quad \text{and} \quad p'Rq' \end{cases}$$

Definition 10.13 (Bisimulation as fixpoint)
Let R *be a relation in* $\mathscr{P}(P \times P)$ *then it is said to be a* bisimulation *iff:*

$$\Phi(R) \sqsubseteq R \quad (i.e.,\ \ R \subseteq \Phi(R))$$

Theorem 10.14 (Bisimilarity as fixpoint)
The function Φ *is monotone and continue, i.e.:*

$$\begin{aligned} R_1 \sqsubseteq R_2 \quad &\Rightarrow \quad \Phi(R_1) \sqsubseteq \Phi(R_2) \\ \Phi(\bigsqcup_{n \in \omega} R_n) \quad &= \quad \bigsqcup_{n \in \omega} \Phi(R_n) \end{aligned}$$

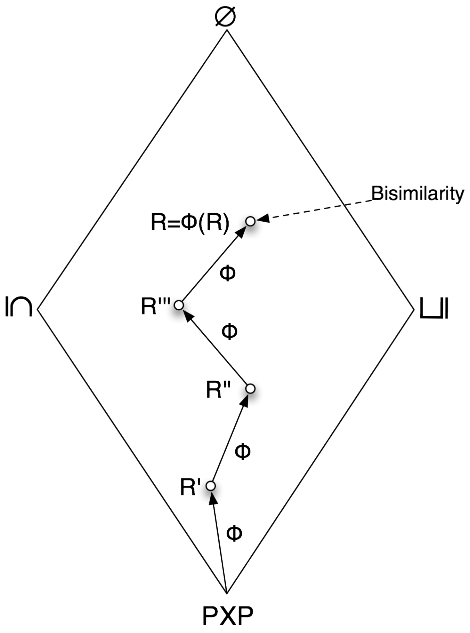


Figure 10.1.: $CPO_{\bot}$ $(\mathscr{P}(P\times P),\sqsubseteq)$

Moreover, the least fixed point of Φ is the bisimilarity, namely it holds:

$$\simeq \overset{\text{def}}{=} \bigsqcap_{R=\Phi(R)} R = \bigsqcup_{n\in\omega} \Phi^n(P\times P)$$

We do not prove the above theorem. Note that monotonicity is obvious, since a larger R will make the conditions on $\Phi(R)$ weaker. Continuity of Φ is granted only if the LTS is finitely branching, namely if every state has a finite number of outgoing transitions. If Φ is not continuous, we still have the existence of a minimal fixpoint, but, it will not be always reachable by the ω chain of approximations.

Example 10.15 (Infinitely branching process)
Let us consider the agent $p = \mathbf{rec}\,x.\ (x \mid \alpha.\,\mathbf{nil})$. The agent p is not guarded, because the occurrence of x in the body of the recursive process is not prefixed by an action. By using the rules of the operational semantics of CCS we have, e.g.:

$$\begin{array}{rl}
\mathbf{rec}\,x.\ (x \mid \alpha.\,\mathbf{nil}) \xrightarrow{\mu} q & \nwarrow_{Rec} \\
(\mathbf{rec}\,x.\ (x \mid \alpha.\,\mathbf{nil})) \mid \alpha.\,\mathbf{nil} \xrightarrow{\mu} q & \nwarrow_{Com\ 1st\ rule,\ q=q_1\ \mid\ \alpha.\,\mathbf{nil}} \\
\mathbf{rec}\,x.\ (x \mid \alpha.\,\mathbf{nil}) \xrightarrow{\mu} q_1 & \nwarrow_{Rec} \\
(\mathbf{rec}\,x.\ (x \mid \alpha.\,\mathbf{nil})) \mid \alpha.\,\mathbf{nil} \xrightarrow{\mu} q_1 & \nwarrow_{Com\ 1st\ rule,\ q_1=q_2\ \mid\ \alpha.\,\mathbf{nil}} \\
\mathbf{rec}\,x.\ (x \mid \alpha.\,\mathbf{nil}) \xrightarrow{\mu} q_2 & \nwarrow_{Rec} \\
\dots & \\
\mathbf{rec}\,x.\ (x \mid \alpha.\,\mathbf{nil}) \xrightarrow{\mu} q_n & \nwarrow_{Rec} \\
(\mathbf{rec}\,x.\ (x \mid \alpha.\,\mathbf{nil})) \mid \alpha.\,\mathbf{nil} \xrightarrow{\mu} q_n & \nwarrow_{Com\ 2nd\ rule,\ q_n=(\mathbf{rec}\,x.\ (x\ \mid\ \alpha.\,\mathbf{nil}))\ \mid\ q'} \\
\alpha.\,\mathbf{nil} \xrightarrow{\mu} q' & \nwarrow_{Act,\ \mu=\alpha, q'=\mathbf{nil}} \\
\square &
\end{array}$$

It is then evident that for any $n \in \omega$ we have:

$$\mathbf{rec}\,x.\ (x \mid \alpha.\,\mathbf{nil}) \xrightarrow{\alpha} (\mathbf{rec}\,x.\ (x \mid \alpha.\,\mathbf{nil})) \mid \mathbf{nil} \mid \underbrace{\alpha.\,\mathbf{nil} \mid \cdots \mid \alpha.\,\mathbf{nil}}_{n}$$

The following lemma ensures that if we consider only guarded terms then the LTS is finitely branching.

Lemma 10.16
Let p be a guarded CCS term then $\{q \mid p \xrightarrow{\mu} q\}$ *is a finite set.*

In order to apply the fixpoint theorem to calculate the bisimilarity, we consider only states which are reachable from the states we want to compare for bisimilarity. If the number of reachable states is finite, i.e. if the system is a finite state automata, the calculation is effective, but possibly quite complex if the number of states is large.

Example 10.17 (Bisimilarity as fixpoint)
Let us consider the example 10.6 which we have already solved with game theory techniques. Now we show the fixpoint approach to the same system. Let us restrict the attention to the set of reachable states and represent the relations by showing the equivalence classes which they induce (over reachable processes) . We start with the coarsest relation (just one equivalence class):

$$R_0 = \{\ \{\alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil})\ ,\ \alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil}\ ,\ \beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}\ ,\ \beta.\,\mathbf{nil}\ ,\ \gamma.\,\mathbf{nil}\ ,\ \mathbf{nil}\}\ \}$$

By applying Φ*:*

$$R_1 = \Phi(R_0) = \{\ \{\alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}), \alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil}\}\ ,\ \{\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}\}\ ,\ \{\beta.\,\mathbf{nil}\}\ ,\ \{\gamma.\,\mathbf{nil}\}\ ,\ \{\mathbf{nil}\}\ \}$$

$$R_2 = \Phi(R_1) = \{\ \{\alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil})\}\ ,\ \{\alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil}\}\ ,\ \{\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}\}\ ,\ \{\beta.\,\mathbf{nil}\}\ ,\ \{\gamma.\,\mathbf{nil}\}\ ,\ \{\mathbf{nil}\}\ \}$$

Note that R_2 *is a fixpoint, hence it is the coarsest bisimulation.*

10.4. Compositionality

In this section we focus our attention on the *compositionality* aspect of the abstract semantics which we have just introduced. For an abstract semantics to be practically relevant it is important that any process used in a system can be replaced with an equivalent process without changing the semantics of the system. Since we have not used structural induction in defining the abstract semantics of CCS, no one ensures any kind of compositionality w.r.t. the possible way of constructing larger systems, i.e., w.r.t. the operators of CCS.

Definition 10.18 (Context)
A context *is a term with a gap which can be filled by inserting any other term of our language. We write* $C[\,]$ *to indicate a* context.

Definition 10.19 (Congruence)
A relation $\sim_{\mathscr{C}}$ *is said to be a* congruence *(with respect to a class of contexts) if:*

$$\forall C[\,].\ p \sim_{\mathscr{C}} q \quad \Rightarrow \quad C[p] \sim_{\mathscr{C}} C[q]$$

In order to guarantee the compositionality of CCS we must show that the bisimilarity is a congruence relation.

Let us now see an example of a relation which is not a congruence.

Example 10.20 (Trace equivalence)
Let us consider the trace equivalence relation, which we have defined in Section 10.3.2. Take the following context:

$$C[\_] = (\ \_ \mid \bar{\alpha}.\bar{\beta}.\,\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma$$

Now we can fill the gaps with the following terms:

$$C[p] = (\alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}) \mid \bar{\alpha}.\bar{\beta}.\,\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma$$

$$C[q] = ((\alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil}) \mid \bar{\alpha}.\bar{\beta}.\,\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma$$

Obviously $C[p]$ and $C[q]$ generate the same set of traces, however one of the processes can "deadlock" before the interaction on β takes place, but not the other. The difference can be formalized if we consider the so-called completed trace semantics.

A completed trace *of a process p is a sequence of actions $\mu_1 \cdots \mu_k$ (for $k \geq 0$) such that there exists a sequence of transitions*

$$p = p_0 \xrightarrow{\mu_1} p_1 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_{k-1}} p_{k-1} \xrightarrow{\mu_k} p_k \nrightarrow$$

for some $p_1, ..., p_k$. The completed traces of a process characterize the sequences of actions that can lead the system to a deadlocked configuration, where no further action is possible.

The completed trace semantics of p is the same as that of q, namely $\{\ \alpha\beta\ ,\ \alpha\gamma\ \}$. However, the completed traces of $C[p]$ and $C[q]$ are $\{\ \tau\tau\ \}$ and $\{\ \tau\tau\ ,\ \tau\ \}$, respectively. We can thus conclude that the completed trace semantics is not a congruence.

10.4.1. Bisimilarity is Preserved by Parallel Composition

In order to show that bisimilarity is a congruence we should prove that the property holds for all the operators of CCS, since this implies that the property holds for all contexts. However we show the proof only for parallel composition, which is a quite interesting case to consider. The other cases follow by similar arguments.

Formally, we need to prove that:

$$p_1 \simeq p_2 \wedge q_1 \simeq q_2 \quad \overset{?}{\Longrightarrow} \quad p_1 \mid q_1 \simeq p_2 \mid q_2$$

As usual we assume the premises and we would like to prove:

$$\exists R.\ (p_1 \mid q_1)\ R\ (p_2 \mid q_2)\ \wedge\ R \subseteq \Phi(R)$$

Since $p_1 \simeq p_2$ and $q_1 \simeq q_2$ we have:

$$\begin{array}{ll} p_1\ R_1\ p_2 & \text{for some bisimulation } R_1 \\ q_1\ R_2\ q_2 & \text{for some bisimulation } R_2 \end{array}$$

Now we define a bisimulation that satisfies the requested property:

$$R \overset{\text{def}}{=} \{(\hat{p}_1 \mid \hat{q}_1\ ,\ \hat{p}_2 \mid \hat{q}_2) \mid \hat{p}_1\ R_1\ \hat{p}_2\ \wedge\ \hat{q}_1\ R_2\ \hat{q}_2\}$$

By definition it holds $p_1 \mid q_1\ R\ p_2 \mid q_2$.
Now we show that R is a bisimulation ($R \subseteq \Phi(R)$):

$$P(p_1|q_1 \xrightarrow{\mu} p_1'|q_1') \overset{\text{def}}{=} \forall p_2, q_2.\ (p_1|q_1\ R\ p_2|q_2 \Longrightarrow \exists p_2', q_2'.\ p_2|q_2 \xrightarrow{\mu} p_2'|q_2'\ \wedge\ p_1'|q_1'\ R\ p_2'|q_2')$$

We proceed by rule induction. There are three possible rules for parallel composition (Com). We start by considering the rule:

$$\frac{p \xrightarrow{\mu} p'}{p|q \xrightarrow{\mu} p'|q}$$

The property for this rule is the following:

$$P(p|q \xrightarrow{\mu} p'|q) \overset{\text{def}}{=} \forall p_2, q_2.\ (p|q\ R\ p_2|q_2 \Longrightarrow \exists p_2', q_2'.\ p_2|q_2 \xrightarrow{\mu} p_2'|q_2'\ \wedge\ p'|q\ R\ p_2'|q_2')$$

We assume that $p \xrightarrow{\mu} p'$ and that, by definition of R, $p\ R_1\ p_2$ and $q\ R_2\ q_2$. Then we have:

$$\exists p_2'.p_2 \xrightarrow{\mu} p_2'\ \wedge\ p'\ R_1\ p_2'$$

By applying the first (Com) rule:

$$p_2|q_2 \xrightarrow{\mu} p_2'|q_2$$

By definition of R we conclude:

$$p'|q \; R \; p_2'|q_2$$

The proof for the second rule

$$\frac{q \xrightarrow{\mu} q'}{p|q \xrightarrow{\mu} p|q'}$$

is analogous.

Finally, we consider the third (Com) rule:

$$\frac{p \xrightarrow{\lambda} p' \quad q \xrightarrow{\bar{\lambda}} q'}{p|q \xrightarrow{\tau} p'|q'}$$

The property for this rule is the following:

$$P(p|q \xrightarrow{\tau} p'|q') \overset{\text{def}}{=} \forall p_2, q_2.\ (p|q \; R \; p_2|q_2 \implies \exists p_2', q_2'.\ p_2|q_2 \xrightarrow{\tau} p_2'|q_2' \wedge p'|q' \; R \; p_2'|q_2')$$

Assuming the premise and by definition of R we have:

$$p \xrightarrow{\lambda} p' \qquad q \xrightarrow{\bar{\lambda}} q' \qquad p \; R_1 \; p_2 \qquad q \; R_2 \; q_2$$

Therefore:

$$\begin{array}{lllllll} p \xrightarrow{\lambda} p' & \wedge & p \; R_1 \; p_2 & \Rightarrow & \exists p_2'.\ p_2 \xrightarrow{\lambda} p_2' & \wedge & p' \; R_1 \; p_2' \\ q \xrightarrow{\bar{\lambda}} q' & \wedge & q \; R_2 \; q_2 & \Rightarrow & \exists q_2'.\ q_2 \xrightarrow{\bar{\lambda}} q_2' & \wedge & q' \; R_2 \; q_2' \end{array}$$

By applying the third (Com) rule we obtain:

$$p_2|q_2 \xrightarrow{\tau} p_2'|q_2'$$

We conclude by definition of R:

$$p'|q' \; R \; p_2'|q_2'$$

10.5. Hennessy - Milner Logic

In this section we present a *modal logic* introduced by Matthew Hennessy and Robin Milner. Modal logic allows to express concepts as “there exists a next state such that”, or “for all next states”, some property holds. Typically, model checkable properties are stated as formulas in some modal logic. In particular, Hennessy-Milner modal logic is relevant for its simplicity and for its close connection with bisimilarity. As we will see, in fact, two bisimilar agents verify the same set of modal logic formulas. This fact shows that bisimilarity is at the right level of abstraction.

First of all we introduce the syntax of the *Hennessy-Milner logic* (HM-logic):

$$F ::= true \quad | \quad false \quad | \quad \wedge_{i \in I} F_i \quad | \quad \vee_{i \in I} F_i \quad | \quad \Diamond_\mu F \quad | \quad \Box_\mu F$$

We write $\mathscr{L}$ for the set of the HM-logic formulas.

The formulas of HM-logic express properties of LTS states, namely in our case of CCS agents. The meanings of the logic operators are the following:

- *true*: is the formula satisfied by every agent. Notice that *true* can be considered a shorthand for an indexed conjunction $\wedge_{i \in I}$ where the set I of indexes is empty.
- $false$: is the formula never satisfied by any agent. Notice that $false$ can be considered a shorthand for an indexed disjunction $\vee i \in I$ where the set I of indexes is empty.

- $\wedge_{\forall i\in I}F_i$: is equivalent to the classic "and" operator applied to the set of formulas $\{F_i\}_{i\in I}$.

- $\vee_{\forall i\in I}F_i$: is equivalent to the classic "or" operator applied to the set of formulas $\{F_i\}_{i\in I}$.

- $\Diamond_\mu F$: it is a *modal operator*, an agent p satisfies this formula if there exists a transition from p to q labelled with μ and the formula F holds in q.

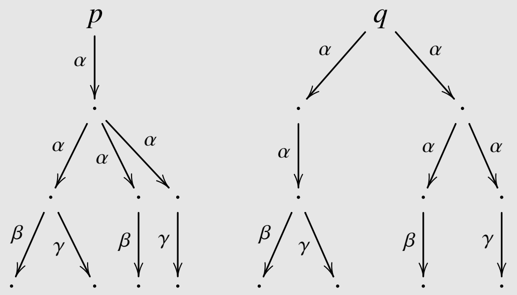
- $\Box_\mu F$: it is also a *modal operator*, an agent p satisfies this formula if for any q such that there is a transition from p to q labelled with μ the formula F holds in q.

As usual in logic satisfaction is defined as a relation $\models$ between formulas and their models, which in our case are states of a LTS.

Definition 10.21 (Satisfaction relation)
The satisfaction relation $\models\ \subseteq P \times \mathscr{L}$ *is defined as follows:*

$$\begin{array}{lll}
p \models true & & \\
p \models \wedge_{i\in I}F_i & \textit{iff} & \forall i \in I.\ p \models F_i \\
p \models \vee_{i\in I}F_i & \textit{iff} & \exists i \in I.\ p \models F_i \qquad \forall i \in I \\
p \models \Diamond_\mu F & \textit{iff} & \exists p'.\ p \xrightarrow{\mu} p' \wedge p' \models F \\
p \models \Box_\mu F & \textit{iff} & \forall p'.\ p \xrightarrow{\mu} p' \Rightarrow p' \models F
\end{array}$$

Example 10.22 (non-equivalent agents)
Let us consider two CCS agents p and q associated with the following graphs:



We would like to show a formula F which is satisfied by one of the two agents and not by the other. For example we can take:

$$F = \Diamond_\alpha \Box_\alpha (\Diamond_\beta true \wedge \Diamond_\gamma true)$$

we have:

$$q \models F \qquad p \not\models F$$

In fact in q we can choose the left α-transition and we reach a state that satisfies $\Box_\alpha(\Diamond_\beta true \wedge \Diamond_\gamma true)$ (i.e., the (only) state reachable by an α-transition can perform both γ and β). On the contrary, the agent p does not satisfy the formula F because after the unique α-transition it is possible to take α-transitions that lead to states where either β or γ is enabled, but not both.

The HM-logic induces an obvious equivalence on CCS processes: two agents are logically equivalent if they satisfy the same set of formulas. Now we present two theorems which allow us to connect bisimilarity and modal logic. As we said this connection is very important both from theoretical and practical point of view. We start by introducing a measure over formulas to estimate the maximal number of consecutive steps that must be taken into account to check the validity of the formulas.

Definition 10.23 (Depth of a formula)
We define the depth *of a formula as follows:*

$$\begin{array}{rcl} D(true) & = & 0 \\ D(false) & = & 0 \\ D(\wedge_{i\in I} F_i) & = & \max(D(F_i) \mid i \in I) \\ D(\vee_{i\in I} F_i) & = & \max(D(F_i) \mid i \in I) \\ D(\Diamond_\mu F) & = & D(F)+1 \\ D(\Box_\mu F) & = & D(F)+1 \end{array}$$

We will denote the set of logic formulas of depth k with $\mathscr{L}_k = \{F \mid D(F) = k\}$.

The first theorem ensures that if two agents are not distinguished by the k^{th} iteration of the fixpoint calculation of bisimilarity, then no formula of depth k can distinguish between the two agents, and viceversa.

Theorem 10.24
Let $\sim_k$ *be defined as follows:*

$$p \sim_k q \Leftrightarrow p\ \Phi^k(P \times P)\ q$$

and let p and q be two CCS agents. Then, we have:

$$p \sim_k q \quad \textit{iff} \quad \forall F \in \mathscr{L}_k.\ (p \models F) \Leftrightarrow (q \models F)$$

The second theorem generalizes the above correspondence by setting up a connection between formulas of any depth and bisimilarity. The proof is by induction on the depth of formulas.

Theorem 10.25
Let p and q two CCS agents, then we have:

$$p \simeq q \quad \textit{iff} \quad \forall F.(p \models F) \Leftrightarrow (q \models F)$$

It is worth reading this result both in the positive sense, namely bisimilar agents satisfy the same set of HM formulas; and in the negative sense, namely if two agents are not bisimilar, then there exists a formula which distinguishes between them. From a theoretical point of view these theorems show that bisimilarity distinguishes all and only those agents which are really different because they enjoy different properties. These results witness that the relation $\simeq$ is a good choice from the logical point of view.

10.6. Axioms for Strong Bisimilarity

Finally, we show that strong bisimilarity can be finitely axiomatized. First we present a theorem which allows to derive for every non recursive CCS agent a suitable normal form.

Theorem 10.26
Let p be a (non-recursive) CCS agent, then there exists a CCS agent strongly bisimilar to p built only with prefix, sum and nil.

Proof. We proceed by structural recursion. First define two binary operators $\rfloor$ and $\|$, where $\rfloor q$ means that q does not perform any action, and $p_1 \| p_2$ means that p_1 and p_2 must perform a synchronization. This corresponds to say that the operational semantics rules for $p \rfloor q$ and $p \| q$ are:

$$\frac{p \xrightarrow{\mu} p'}{p \rfloor q \xrightarrow{\mu} p' | q} \qquad \frac{p \xrightarrow{\lambda} p' \quad q \xrightarrow{\bar{\lambda}} q'}{p \| q \xrightarrow{\tau} p' | q'}$$

We show how to decompose the parallel operator, then we show the other cases:

$$p_1 \mid p_2 \simeq p_1 \rfloor p_2 \ + \ p_2 \rfloor p_1 \ + \ p_1 \| p_2$$

Moreover we have the following equalities:

$$\begin{array}{rcl}
\mu.p \rfloor q & \simeq & \mu.(p \mid q) \\
(p_1 + p_2) \rfloor q & \simeq & p_1 \rfloor q \ + \ p_2 \rfloor q \\
\\
\mu_1.p_1 \| \mu_2.p_2 & \simeq & \mathbf{nil} \ \text{ if } \mu_1 \neq \bar{\mu}_2 \\
\lambda.p_1 \| \overline{\lambda}.p_2 & \simeq & \tau.(p_1 \mid p_2) \\
(p_1 + p_2) \| q & \simeq & p_1 \| q \ + \ p_2 \| q \\
p \| (q_1 + q_2) & \simeq & p \| q_1 \ + \ p \| q_2 \\
\\
(\mu.p) \setminus \alpha & \simeq & \mathbf{nil} \ \text{ if } \mu \in \{\alpha, \overline{\alpha}\} \\
(\mu.p) \setminus \alpha & \simeq & \mu.(p \setminus \alpha) \ \text{ if } \mu \neq \alpha, \overline{\alpha} \\
(p_1 + p_2) \setminus \alpha & \simeq & p_1 \setminus \alpha \ + \ p_2 \setminus \alpha \\
\\
(\mu.p)[\phi] & \simeq & \phi(\mu).p[\phi] \\
(p_1 + p_2)[\phi] & \simeq & p_1[\phi] \ + \ p_2[\phi] \\
\\
\mathbf{nil} \setminus \alpha & \simeq & \mathbf{nil}[\phi] \simeq \mathbf{nil} \mid \mathbf{nil} \simeq \mathbf{nil} \rfloor p \simeq \mathbf{nil} \| p \simeq p \| \mathbf{nil} \simeq \mathbf{nil}
\end{array}$$

□

From the previous theorem, it follows that every finite CCS agent can be equivalently written using action prefix, sum and **nil**. Then, the axioms that characterize the strong bisimilarity relation are the following:

$$\begin{array}{rcl}
p + \mathbf{nil} & = & p \\
p_1 + p_2 & = & p_2 + p_1 \\
p_1 + (p_2 + p_3) & = & (p_1 + p_2) + p_3 \\
p + p & = & p
\end{array}$$

Note that the axioms simply assert that processes with sum define an idempotent, commutative monoid whose neutral element is **nil**.

10.7. Weak Semantics of CCS

Let us now see an example that illustrates the limits of strong bisimilarity as a behavioural equivalence between agents.

Example 10.27
Let p and q be the following CCS agents:

$$p = \tau.\,\mathbf{nil} \qquad q = \mathbf{nil}$$

Obviously the two agents are distinguished by the (invisible) action τ. So they are not bisimilar, but, since we consider τ as an internal action, not visible from outside of the system, we have that, according to the observable behaviours, they should not be distinguished.

The above example shows that strong bisimilarity is not abstract enough. So we could think to abstract away from the invisible transition by defining a new relation. This relation is called *weak bisimilarity*. We start by defining a new, more abstract, LTS.

10.7.1. Weak Bisimilarity

Definition 10.28 (Weak transitions)
We let $\Longrightarrow$ *be the* weak transition relation *on the set of states of an LTS defined as follows:*

$$\begin{array}{lll} p \overset{\tau}{\Rightarrow} q & \textit{iff} & p \overset{\tau}{\rightarrow} \ldots \overset{\tau}{\rightarrow} q \vee p = q \\ p \overset{\lambda}{\Rightarrow} q & \textit{iff} & p \overset{\tau}{\Rightarrow} p' \overset{\lambda}{\rightarrow} q' \overset{\tau}{\Rightarrow} q \end{array}$$

Note that $p \overset{\tau}{\Rightarrow} q$ means that q can be reached from p via a possibly empty sequence of τ-transitions, i.e., $\overset{\tau}{\Rightarrow}$ coincides with the reflexive and transitive closure $(\overset{\tau}{\rightarrow})^*$ of invisible transition $\overset{\tau}{\rightarrow}$, while $p \overset{\lambda}{\Rightarrow} q$ requires the execution of one visible transition (the one labelled with λ).

Now, as done for the strong bisimilarity, we define a function $\Psi : \mathscr{P}(P \times P) \rightarrow \mathscr{P}(P \times P)$ which takes a relation on P and returns a another relation (if it exists) by exploiting weak transitions:

$$p \; \Psi(R) \; q \overset{\text{def}}{=} \begin{cases} p \overset{\mu}{\rightarrow} p' & \text{implies} \quad \exists q'.\, q \overset{\mu}{\Rightarrow} q' \quad \text{and} \quad p' \; R \; q' \\ q \overset{\mu}{\rightarrow} q' & \text{implies} \quad \exists p'.\, p \overset{\mu}{\Rightarrow} p' \quad \text{and} \quad p' \; R \; q' \end{cases}$$

And we define the *weak bisimilarity* as follows:

$$p \approx q \text{ iff } \exists R.\, p \; R \; q \wedge \Psi(R) \sqsubseteq R$$

This relation seems to improve the notion of equivalence w.r.t. $\simeq$, because $\approx$ abstracts away from the invisible transitions as we required. Unfortunately, there are two problems with this relation. First, the $\Rightarrow$ LTS is infinite branching also for guarded terms (consider e.g. $rec\ x.(\tau.x|\alpha.nil)$, analogous to the agent discussed in example 10.15). Thus function Ψ is not continuous, and the minimal fixpoint, which exists anyway, cannot be reached in general with an ω-chain of approximations. Second, and much worse, weak bisimilarity is not a congruence with respect to the + operator, as the following example shows. As a (minor) consequence, weak bisimilarity, differently than strong bisimilarity, cannot be axiomatized.

Example 10.29
Let p and q be the following CCS agents:

$$p = \alpha.\,\mathbf{nil} \qquad q = \tau.\alpha.\,\mathbf{nil}$$

Obviously for the weak equivalence we have $p \approx q$*, since their behaviours differ only by the ability to perform an invisible action* τ*. Now we define the following context:*

$$C[\_] = \_ + \beta.\,\mathbf{nil}$$

Then by embedding p and q within the context $C[\_]$ *we obtain:*

$$\alpha.\,\mathbf{nil} + \beta.\,\mathbf{nil} \not\approx \tau.\alpha.\,\mathbf{nil} + \beta.\,\mathbf{nil}$$

In fact $C[q]$ *can perform a* τ*-transition and become* $\alpha.\,\mathbf{nil}$*, while* $C[p]$ *has only one invisible weak transition that can be used to match such a step, but such weak transition is the idle step* $C[p] \overset{\tau}{\Rightarrow} C[p]$ *and* $C[p]$ *is clearly not equivalent to* $\alpha.\,\mathbf{nil}$ *(because the former can perform a* β*-transition that the latter cannot simulate). This phenomenon is due to the fact that* τ*-transitions are not observable but can be used to discard some non-deterministic choices. While quite unpleasant, the above fact is not in any way due to a CCS weakness, or misrepresentation of reality, but rather enlightens a general property of nondeterministic choice in systems represented as black boxes.*

10.7.2. Weak Observational Congruence

As shown by the Example 10.29, weak bisimilarity is not a congruence relation. In this section we will show a possible (partial) solution. Since weak bisimilarity equivalence is a congruence for all operators except sum, to fix our problem it is enough to impose closure for all sum contexts.

Let us consider the Example 10.29, where the execution of a τ-transition forces the system to make a choice which is invisible to an external observer. In order to make this kind of choices observable we can define the relation $\cong$ as follows:

$$p \cong q \text{ iff } p \approx q \ \wedge\ \forall r.\ p + r \approx q + r$$

This relation, called *weak observational congruence*, can be defined directly as:

$$p \cong q \ \stackrel{\text{def}}{=}\ \begin{cases} p \xrightarrow{\tau} p' & \text{implies} \quad \exists q'.\ q \xrightarrow{\tau}\overset{\tau}{\Longrightarrow} q' \quad \text{and} \quad p' \approx q' \\ p \xrightarrow{\lambda} p' & \text{implies} \quad \exists q'.\ q \overset{\lambda}{\Longrightarrow} q' \quad \text{and} \quad p' \approx q' \\ \text{(and vice versa)} & \end{cases}$$

As we can see we avoided the possibility to stop after the execution of an internal action. Notice however that this is not a fixpoint definition, since $\cong$ is simply defined in terms of $\approx$. This relation is a congruence but as we can see in the following example it is not a bisimulation according to Ψ, namely $\Psi(\cong) \neq \cong$.

Example 10.30
Let p and q defined as follows:

$$p = \alpha.\tau.\beta.\,\mathbf{nil} \quad \textit{and} \quad q = \alpha.\beta.\,\mathbf{nil}$$

we have:

$$p \cong q$$

but if Alice plays α on p, Bob has no chance of playing α and of reaching a state in relation $\cong$:

$$p' = \tau.\beta.\,\mathbf{nil} \quad \textit{and} \quad q' = \beta.\,\mathbf{nil}$$

since $p' \ncong q'$. Thus $\cong$ is not a fixpoint of Ψ.

It is possible to prove that the equivalence relation $\cong$ can be axiomatized by adding to the axioms for strong bisimilarity the following three Milner's τ laws:

$$\begin{aligned} & p + \tau.p = \tau.p \\ & \mu.(p + \tau.q) = \mu.(p + \tau.q) + \mu.q \\ & \mu.\tau.p = \mu.p \end{aligned}$$

10.7.3. Dynamic Bisimilarity

As shown by the Example 10.30 the observational congruence is not a bisimulation. In this section we present the largest relation which is at the same time a congruence and a Ψ-bisimulation. It is called *dynamic bisimilarity* and was introduced by Vladimiro Sassone.

We define the dynamic bisimilarity $\approx_d$ as the largest relation that satisfies:

$$p \approx_d q \quad \text{implies} \quad \forall C.\ C[p]\ \Psi(\approx_d)\ C[q]$$

In this case, at every step we close the relation by comparing the behaviour w.r.t. any possible embedding context. In terms of game theory this definition can be viewed as "at each turn Alice is also allowed to insert both agents into the same context in order to win."

As for the observational congruence, we can define the dynamic bisimilarity as follows:

$$p\ \Theta(R)\ q \ \stackrel{\text{def}}{=}\ \begin{cases} p \xrightarrow{\tau} p' & \text{implies} \quad \exists q'.\ q \xrightarrow{\tau}\overset{\tau}{\Longrightarrow} q' \quad \text{and} \quad p'\, R\, q' \\ p \xrightarrow{\lambda} p' & \text{implies} \quad \exists q'.\ q \overset{\lambda}{\Longrightarrow} q' \quad \text{and} \quad p'\, R\, q' \\ \text{(and vice versa)} & \end{cases}$$

Then, R is a *dynamic bisimulation* if $\Theta(R) \sqsubseteq R$, and the dynamic bisimilarity is obtained by letting:

$$\approx_d = \bigsqcap_{R \subseteq \Theta(R)} R$$

Example 10.31
Let p and q be defined as in the Example 10.30.

$$p = \alpha.\tau.\beta.\,\mathbf{nil} \quad \textit{and} \quad q = \alpha.\beta.\,\mathbf{nil}$$

$$p' = \tau.\beta.\,\mathbf{nil} \quad \textit{and} \quad q' = \beta.\,\mathbf{nil}$$

we have:

$$p \not\approx_d q \quad \textit{and} \quad p' \not\approx_d q'$$

As for the observational congruence we can finitely axiomatize the dynamic bisimilarity. The axiomatization of $\approx_d$ is obtained by omitting the third Milner's τ law as follows:

$$\begin{aligned} &p + \tau p = \tau p \\ &\mu(p + \tau q) = \mu(p + \tau q) + \mu q \end{aligned}$$

11. Temporal Logic and μ-Calculus

As we have discussed in the previous chapter (see Section 10.5) modal logic is a powerful tool that allows to check some behavioral properties of systems. In Section 10.5 the focus was on Hennessy-Milner logic, whose main limitation is due to its finitary structure: only local properties can be investigated. In this chapter we show some extensions of Hennessy-Milner logic that increase the expressiveness of the formulas. The most powerful language that we will present is the *μ-Calculus*. It allows to express complex constraints about the infinite behaviour of our systems.

Classically, we can divide the properties to be investigated in three categories:

- *safety*: if the property expresses the fact that something bad will not happen.
- *liveness*: if the property expresses the fact that something good will happen.
- *fairness*: if the property expresses the fact that something good will happen infinitely many times.

11.1. Temporal Logic

The first step in extending modal logic is to introduce the concept of time in our models. This will extend the expressiveness of modal logic, making it able to talk about concepts like "ever","never" or "sometimes". In order to represent the concept of time in our logics we have to represent it in a mathematical fashion. In our discussion we assume that the time is discrete and infinite.

While temporal logic shares similarities with HM-logic, note that:

- temporal logic is based on a set of *atomic propositions* whose validity is associated with a set of states, i.e., the observations are taken on states and not on (actions labeling the) arcs;
- temporal operators allows to look further than the "next" operator of HML;
- as we will see, the choice of representing the time as linear (linear temporal logic) or as tree (computation tree logic) will lead to different types of logic, that roughly correspond to the trace semantic view vs the bisimulation semantics view.

11.1.1. Linear Temporal Logic

In the case of *Linear Temporal Logic* (LTL) the time is represented as a line. This means that the evolutions of the system are linear, they proceed from a state to another without making any choice. The formulas of LTL are based on a set of *atomic propositions*, which can be composed using the classical logic operators together with the following temporal operators:

- O: is called *next* operator. The formula $O\phi$ means that ϕ is true in the next state (i.e., in the next instant of time). Some literature uses X or N in place of O.
- F: is called *finally* operator. The formula $F\phi$ means that ϕ is true sometime in the future.
- G: The formula $G\phi$ means that ϕ is always (*globally*) valid in the future.
- U: is called *until* operator. The formula $\phi_1 U \phi_2$ means that ϕ_1 is true until the first time that ϕ_2 is true.

The syntax of LTL is as follows:

$$\phi ::= true \quad | \quad false \quad | \quad p \quad | \quad \neg\phi \quad | \quad \phi_1 \wedge \phi_2 \quad | \quad \phi_1 \vee \phi_2 \quad | \quad O\phi \quad | \quad F\phi \quad | \quad G\phi \quad | \quad \phi_1 U \phi_2$$

In order to represent the state of the system while the time elapses we introduce the following mathematical structure.

Definition 11.1 (Linear structure)
Let P be a set of atomic propositions and $S : P \to 2^\omega$ be a function from the atomic propositions to subsets of natural numbers defined as follows:

$$\forall p \in P.\ S(p) = \{x \in \omega \mid x \text{ satisfies } p\}$$

Then we call the pair (S, P) a linear structure.

In a linear structure, the natural numbers $0, 1, 2 \ldots$ represent the time instants, and the states in them, and S represents for every predicate the states where it holds, or, alternatively, it represents for every state the predicates which it satisfies.The operators of LTL allows to quantify (existentially and universally) w.r.t. the traversed states. To define the satisfaction relation, we need to check properties on future states, like some sort of "time travel". To this aim we define the following *shifting* operation on S:

$$\forall i \in \omega\ \forall p \in P.\ S^i(p) = \{x - i \mid x \geq i\ \wedge\ x \in S(p)\}$$

As done for the HM-logic, we define the satisfaction operator $\models$ as follows:

$$\begin{array}{lll}
S & \models & true \\
S & \models & p \quad \text{if } 0 \in S(p) \\
S & \models & \neg\phi \quad \text{if it is not true that } S \models \phi \\
S & \models & \phi_1 \wedge \phi_2 \quad \text{if } S \models \phi_1 \text{ and } S \models \phi_2 \\
S & \models & \phi_1 \vee \phi_2 \quad \text{if } S \models \phi_1 \text{ or } S \models \phi_2 \\
S & \models & O\phi \quad \text{if } S^1 \models \phi \\
S & \models & F\phi \quad \text{if } \exists i \in \mathbb{N} \text{ such that } S^i \models \phi \\
S & \models & G\phi \quad \text{if } \forall i \in \mathbb{N} \text{ it holds } S^i \models \phi \\
S & \models & \phi_1 U \phi_2 \quad \text{if } \exists i \in \mathbb{N} \text{ such that } S^i \models \phi_2 \text{ and } \forall j < i.\ S^j \models \phi_1
\end{array}$$

We say that an LTL formula ϕ is satisfiable if there is some computation S such that $S \models \phi$.

From the satisfaction relation it is easy to check that the operators F and G can be expressed in terms of the until operator as follows:

$$\begin{array}{lllll}
F\phi & \equiv & true\ U\ \phi & & \\
G\phi & \equiv & \neg F \neg\phi & \equiv & \neg(true\ U\ \neg\phi)
\end{array}$$

We now show some examples that illustrate how powerful the LTL is.

Example 11.2

- $G\neg p$: *expresses the fact that p will never happen, so it is a safety property.*
- $p \to Fq \equiv \neg p \vee Fq$: *expresses the fact that if p happens then also q will happen sometime in the future.*
- GFp: *expresses the fact that p happens infinitely many times in the future, so it is a fairness property.*
- FGp: *expresses the fact that p will always hold some time in the future.*
- $G(request \to (request\ U\ grant))$: *expresses the fact that whenever a request is made it holds continuously until it is eventually granted.*

11.1.2. Computation Tree Logic

In this section we introduce CTL and CTL^* two logics which use trees as models of the time. CTL and CTL^* extend LTL with two operators which allows to express properties on paths over trees. The difference between CTL and CTL^* is that the former is a restricted version of the latter. So we start by introducing CTL^*.

We introduce two new operators on paths:

- E: the formula $E\phi$ (read "possibly ϕ") means that there *exists* some path that satisfies ϕ;
- A: the formula $A\phi$ (read "inevitably ϕ") means that each path of the tree satisfies ϕ, i.e., that ϕ is satisfied along *all* paths.

The syntax of CTL is as follows:

$$\phi ::= true \;\mid\; false \;\mid\; p \;\mid\; \neg\phi \;\mid\; \phi_1 \wedge \phi_2 \;\mid\; \phi_1 \vee \phi_2 \;\mid\; O\phi \;\mid\; F\phi \;\mid\; G\phi \;\mid\; \phi_1 U \phi_2 \;\mid\; E\phi \;\mid\; A\phi$$

This time the state of the system is represented by using infinite trees as follows.

Definition 11.3 (Infinite tree)
Let $T = (V, \to)$ be a tree, with V the set of nodes, v_0 the root and $\to \subseteq V \times V$ the parent-child relation. We say that T is an infinite tree *if the following holds:*

$$\to \; \textit{is total on } V, \textit{ namely } \forall v \in V \; \exists w \in V. \; v \to w$$

Definition 11.4 (Branching structure)
Let P be a set of atomic propositions, $T = (V, \to)$ be an infinite tree and $S : P \to 2^V$ be a function from the atomic propositions to subsets of nodes of V defined as follows:

$$\forall p \in P. \; S(p) = \{x \in V \mid x \textit{ satisfies } p\}$$

Then we call (T, S, P) a branching structure.

We are interested in infinite paths on trees.

Definition 11.5 (Infinite paths)
Let $(V, \to)$ be an infinite tree and $\pi = v_0, v_1, \ldots, v_n, \ldots$ be an infinite sequence of nodes in V. We say that π is an infinite path *over $(V, \to)$ iff*

$$\forall i \in \omega. \; v_i \to v_{i+1}$$

As for the linear case, we need a shifting operators on path. So for $\pi = v_0, v_1, \ldots, v_n, \ldots$ we let π^i be defined as follows:

$$\forall i \in \omega. \; \pi^i = v_i, v_{i+1}, \ldots$$

Let (T, S, P) be a branching structure and $\pi = v_0, v_1, \ldots, v_n, \ldots$ be an infinite path. We define the $\models$ relation as follows:

state operators:

$$\begin{array}{lll} S,\pi & \models & true \\ S,\pi & \models & p \quad \text{if } v_0 \in S(p) \\ S,\pi & \models & \neg\phi \quad \text{if it is not true that } S,\pi \models \phi \\ S,\pi & \models & \phi_1 \wedge \phi_2 \quad \text{if } S,\pi \models \phi_1 \text{ and } S,\pi \models \phi_2 \\ S,\pi & \models & \phi_1 \vee \phi_2 \quad \text{if } S,\pi \models \phi_1 \text{ or } S,\pi \models \phi_2 \\ S,\pi & \models & O\phi \quad \text{if } S,\pi^1 \models \phi \\ S,\pi & \models & F\phi \quad \text{if } \exists i \in \omega \text{ such that } S,\pi^i \models \phi \\ S,\pi & \models & G\phi \quad \text{if } \forall i \in \omega \text{ it holds } S,\pi^i \models \phi \\ S,\pi & \models & \phi_1 U \phi_2 \quad \text{if } \exists i \in \omega \text{ such that } S,\pi^i \models \phi_2 \text{ and for all } j < i.\ S,\pi^j \models \phi_1 \end{array}$$

path operators

$$\begin{array}{lll} S,\pi & \models & E\phi \quad \text{if there exists } \pi_1 = v_0, v'_1, \ldots, v'_n, \ldots \text{ such that } S,\pi_1 \models \phi \\ S,\pi & \models & A\phi \quad \text{if for all paths } \pi_1 = v_0, v'_1, \ldots, v'_n, \ldots \text{ we have } S,\pi_1 \models \phi \end{array}$$

Let us see some examples.

Example 11.6

- *EOp: it is the same of the next operator $\Diamond$ in modal logic.*
- *AGp: expresses the fact that p happens in all reachable states.*
- *EFp: expresses the fact that p happens in some reachable state.*
- *AFp: expresses the fact that on every path there exists a state where p holds.*
- *$E(pUq)$: expresses the fact that there exists a path where p holds until q.*
- *$AGEFp$: in every future exists a successive future where p holds.*

The formulas of CTL are obtained by restricting CTL^*: a CTL^* formula is a CTL formula if the followings hold:

- A and E appear only immediately before a linear operator (i.e., F,G,U and O).
- each linear operator appears immediately after a quantifier (i.e., A and E).

It is evident that CTL and LTL are both subsets of CTL^*, but they are not equivalent to each other. Without going into the detail, we mention that:

- no CTL formula is equivalent to the LTL formula $F(Gp)$;
- no LTL formula is equivalent to the CTL formula $AG(p \to (EOq \wedge EO\neg q))$

Finally, we note that all CTL formulas can be written in terms of the minimal set of operators $true$, $\neg$, $\vee$, EG, EU, EO. In fact, for the remaining operators we have the following logical equivalences:

$$\begin{array}{rcl} EF\phi & \equiv & E(true\ U\ \phi) \\ AO\phi & \equiv & \neg(EO\neg\phi) \\ AG\phi & \equiv & \neg(EF\neg\phi) \quad \equiv \quad \neg E(true\ U\ \neg\phi) \\ AF\phi & \equiv & A(true\ U\ \phi) \quad \equiv \quad \neg(EG\neg\phi) \\ A(\phi\ U\ \varphi) & \equiv & \neg(E(\neg\varphi\ U\ \neg(\phi \vee \varphi)) \vee EG\neg\varphi) \end{array}$$

11.2. μ-Calculus

Now we introduce the μ-calculus. The idea is to add the least and greatest fixpoint operators to modal logic. This fits nicely with the fact that many interesting properties can be conveniently expressed as fixpoints. The two operators that we introduce are the following:

- $\mu x.\phi$ is the least fixpoint of ϕ.
- $\nu x.\phi$ is the greatest fixpoint of ϕ.

The syntax of μ-calculus is as follows:

$$\phi ::= true \quad | \quad false \quad | \quad x \quad | \quad p \quad | \quad \neg\phi \quad | \quad \phi_1 \wedge \phi_2 \quad | \quad \phi_1 \vee \phi_2 \quad | \quad \Diamond\phi \quad | \quad \Box\phi \quad | \quad \mu x.\phi \quad | \quad \nu x.\phi$$

Note that, in order to apply the fixpoint operators we require that ϕ is monotone, this means that any occurrence of x in ϕ must be preceded by an even number of negations. The μ-calculus is interpreted on LTSs.

Let $(V, \rightarrow)$ be an LTS, X be the set of predicate variables and P be a set of predicates, we introduce a function $\rho : P \cup X \rightarrow 2^V$ which associates to each predicate and each free variable a subset of vertices. Then we define the denotational semantics of μ-calculus which maps each predicate to the subset of states in which it holds as follows:

$$\begin{array}{rcl}
[\![true]\!]\rho & = & V \\
[\![false]\!]\rho & = & \varnothing \\
[\![x]\!]\rho & = & \rho x \\
[\![p]\!]\rho & = & \rho p \\
[\![\neg\phi]\!]\rho & = & V \setminus [\![\phi]\!]\rho \\
[\![\phi_1 \wedge \phi_2]\!]\rho & = & [\![\phi_1]\!]\rho \cap [\![\phi_2]\!]\rho \\
[\![\phi_1 \vee \phi_2]\!]\rho & = & [\![\phi_1]\!]\rho \cup [\![\phi_2]\!]\rho \\
[\![\Diamond\phi]\!]\rho & = & \{\, v \mid \exists v'.v \rightarrow v' \ \wedge \ v' \in [\![\phi]\!]\rho \,\} \\
[\![\Box\phi]\!]\rho & = & \{\, v \mid \forall v'.v \rightarrow v' \Rightarrow v' \in [\![\phi]\!]\rho \,\} \\
[\![\mu x.\phi]\!]\rho & = & \mathrm{fix}\, \lambda S.\, [\![\phi]\!]\rho[^S/_x] \\
[\![\nu x.\phi]\!]\rho & = & \mathrm{Fix}\, \lambda S.\, [\![\phi]\!]\rho[^S/_x]
\end{array}$$

Example 11.7

- $[\![\mu x.x]\!]\rho = \varnothing$
- $[\![\nu x.x]\!]\rho = V$
- $[\![\mu x.\Diamond x]\!]\rho = \mathrm{fix}\, \lambda S.\{v \mid \exists v'.\ v \rightarrow v' \wedge v' \in S\}$
  we have:
  $$S_0 = \varnothing \qquad S_1 = \{v \mid \exists v'.\ v \rightarrow v' \wedge v' \in \varnothing\} = \varnothing \qquad \textit{(fixpoint reached)}$$
- $[\![\mu x.\Box x]\!]\rho = \mathrm{fix}\, \lambda S.\{v \mid \forall v'.\ v \rightarrow v' \Rightarrow v' \in S\}$
  we have:
  $$S_0 = \varnothing \qquad S_1 = \{v \mid \forall v'.\ v \rightarrow v' \Rightarrow v' \in \varnothing\} = \{v \mid v \not\rightarrow\} \quad \textit{the set of vertices with no outgoing arcs}$$
  $$S_2 = \{v \mid \forall v'.\ v \rightarrow v' \Rightarrow v' \in S_1\} = \textit{ the set of vertices with outgoing paths of length at most } 1$$
  $$S_n = \{v \mid \forall v'.\ v \rightarrow v' \Rightarrow v' \in S_{n-1}\} = \textit{ the set of vertices with outgoing paths of length at most } n-1$$
  $$\bigcup_{i \in \omega} S_i = \textit{ vertices with only finite outgoing paths}$$

- $[\![\nu x.\Box x]\!]\rho = \mathrm{Fix}\,\lambda S.\{v \mid \forall v', v \to v', v' \in S\}$
  we have:
  $$S_0 = V \qquad S_1 = \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in V\} = V \qquad \textit{(fixpoint reached)}$$

- $[\![\mu x.p \vee \Diamond x]\!]\rho = \mathrm{fix}\,\lambda S.\rho p \cup \{v \mid \exists v'.v \to v' \wedge v' \in S\}$ *(similar to EFp, meaning some node in ρp is reachable)*
  we have:
  $$S_0 = \emptyset \qquad S_1 = \rho p \qquad S_2 = \rho p \cup \{v \mid \exists v'.\ v \to v' \wedge v' \in \rho p\} = \rho p \textit{ is reachable in at most one step}$$
  $$S_n = \rho p \textit{ is reachable in at most } n-1 \textit{ steps} \qquad \bigcup_{i \in \omega} S_i = \rho p \textit{ is reachable (in any number of steps)}$$

- $[\![\nu x.\mu y.(p \wedge \Diamond x) \vee \Diamond y]\!]\rho$ *(corresponds to $EGFp$)*
  start a path, $\mu y.(p \wedge \Diamond x) \vee \Diamond y$ means that after a finite number of steps you find a vertex where both (1) p holds and (2) you can reach a vertex where the property recursively holds.

- $[\![\mu x.(p \wedge \Box x \wedge \Diamond x) \vee q]\!]\rho = \mathrm{fix}\,\lambda S.(\rho p \cap \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in S\} \cap \{v \mid \exists v'.\ v \to v' \wedge v' \in S\}) \cup \rho q$ *(corresponds to $ApUq$)*

  Note that in this case the $\Diamond x$ is necessary in order to ensure that the state is not a deadlock one.

- $[\![\mu x.(p \wedge \Diamond x) \vee q]\!]\rho = \mathrm{fix}\,\lambda S.(\rho p \cap \{v \mid \exists v'.\ v \to v' \wedge v' \in S\}) \cup \rho q$ *(corresponds to $EpUq$)*

11.3. Model Checking

The problem of model checking consists in the, possibly automatic, verification of whether a given model of a system meets or not a given logic specification of the properties the system should satisfy, like absence of deadlocks.

The main ingredients of model checking are:

- an LTS (the model) and a vertex (the initial state);
- a formula (in temporal or modal logic) you want to check (for that state in the model)

The result of model checking should be either a positive answer (the given state in the model satisfies the formula) or some counterexample explaining one possible reason why the formula is not satisfied.

In the case of concurrent systems, the LTS is often given implicitly, as the one associated with a term of some process algebra, because in this way the structure of the system is handled more conveniently. However the size of the actual translation can explode even if the system is finite state. For example, let $p_i = \alpha_i.\,\mathbf{nil}$ for $i = 1, .., n$ and take the CCS process $s = p_1 \mid p_2 \mid \cdots \mid p_n$: the number of reachable states of the resulting model is 2^n.

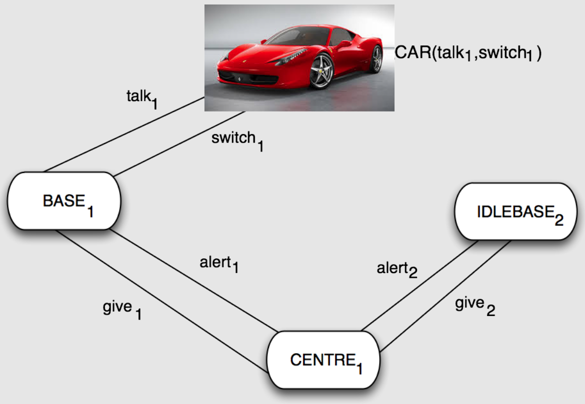
One possibility to tackle the state explosion problem is to minimize the system according to some suitable equivalence. Note that minimization can take place also while combining subprocesses and not just at the end. Of course, this technique is viable only if the minimization is related to an equivalence relation that respects the properties to be checked. For example, the μ-calculus is invariant w.r.t. bisimulation, thus we can minimize CCS processes up to bisimilarity before model checking them.

In model checking algorithms, it is often convenient to proceed by evaluating formulas with the aid of dynamic programming. The idea is to work in a bottom-up fashion: starting from the atomic predicates that appear in the formula, we mark all the states with the sub formulas they satisfy. When a variable is encountered, a separate activation of the procedure is allocated for computing the fixpoint of the corresponding recursive definition. The complexity becomes very large in the case of formulas that involve many least and greatest fix points in alternation.

12. π-Calculus

The structures of today's communication systems are not statically defined, but they change continuously according to the needs of the users. The CCS calculus we saw in chapter 10 is unsuitable for modeling such systems, since its communication structure (the channels) cannot evolve dynamically. In this chapter we present the π-calculus, an extension of CCS introduced by Robin Milner, Joachim Parrow and David Walker in 1989 which allows to model mobile systems. The main feature of the π-calculus is its ability of creating new channels and of sending them in messages allowing agents to change their connections. Let us start with an example which illustrates how the π-calculus can formalize a mobile telephone system.

Example 12.1 (Mobile Phones)
The following figure which represents a mobile phone network: while the car travels, the phone can communicate with different bases in the city, but just one at a time, typically the closest to its position. The communication centre decides when the base must be changed and then the channel for accessing the new base is sent to the car through the switch channel.



As for CCS, also in this case we describe agent behaviour by defining the reachable states:

$$CAR(talk, switch) \stackrel{\text{def}}{=} \overline{talk}.CAR(talk, switch) + switch(talk', switch').CAR(talk', switch')$$

A car can talk on the channel assigned by the communication centre (action $\overline{talk}$). Alternatively the car can receive (action $switch(talk', switch')$) a new pair of channels ($talk'$ and $switch'$) and change the base to which it is connected.

$$BASE_i \stackrel{\text{def}}{=} BASE(talk_i, switch_i, give_i, alert_i) \stackrel{\text{def}}{=}$$
$$talk_i.BASE_i + give_i(talk', switch').\overline{switch_i}(talk', switch').IDLEBASE_i$$

$$IDLEBASE_i \stackrel{\text{def}}{=} IDLEBASE(talk_i, switch_i, give_i, alert_i) \stackrel{\text{def}}{=} alert_i.BASE_i$$

A generic base can be in two possible states: $BASE$ or $IDLEBASE$. In the first case the base is connected to the car, so either the phone can talk or the base can receive two channels from the centre and send them to the car for allowing it to change base. In the second case the base is idle, so it can only be awakened by the communication centre.

$$CENTRE_1 \stackrel{\text{def}}{=} CENTRE_1(give_1, alert_1, give_2, alert_2) = \overline{give_1}(talk_2, switch_2).\overline{alert_2}.CENTRE_2$$
$$CENTRE_2 \stackrel{\text{def}}{=} CENTRE_2(give_1, alert_1, give_2, alert_2) = \overline{give_2}(talk_1, switch_1).\overline{alert_1}.CENTRE_1$$

The communication centre can be in different states according to which base is active. In the example there are only two possible states for the communication centre ($CENTRE_1$ and $CENTRE_2$), because only two bases are considered.

$$SYSTEM_1 \stackrel{\text{def}}{=} (CAR(talk_1, switch_1)|BASE_1|IDLEBASE_2|CENTRE_1)$$

Finally we have the process which represents the entire system in the state where the first car is talking.

Example 12.2 (Secret Channel via Trusted Server)

As another example, consider two processes Alice (A) and Bob (B) that want to establish a secret channel using a trusted server (S) with which they already have trustworthy communication link c_{AS} (for Alice to send private messages to the server) and c_{SB} (for the server to send private messages to Bob). The system can be represented by the expression:

$$Sys \stackrel{\text{def}}{=} (c_{AS})(c_{BS})(A|S|B)$$

where the restrictions (c_{AS}) *and* (c_{BS}) *guarantees that the link* c_{AS} *and* c_{SB} *are not visible from the environment and where the processes* A, S *and* B *are specified as follows:*

$$\begin{array}{rcl} A & \stackrel{\text{def}}{=} & (c_{AB})\bar{c}_{AS}c_{AB}.\bar{c}_{AB}m.p_A \\ S & \stackrel{\text{def}}{=} & !c_{AS}(x).\bar{c}_{SB}x.\,\mathbf{nil} \\ B & \stackrel{\text{def}}{=} & c_{SB}(y).y(m).q_B \end{array}$$

Restriction (x) is similar to the CCS operator $\backslash x$, with the important difference that in π-calculus the scope of the restriction can change as the process evolves. Alice defines a private name c_{AB} that wants to use for communicating with B, then Alice sends the name c_{AB} to the trusted server over their private shared link c_{AS} and finally sends the message m on the channel c_{AB} and continues as p_A. The server continuously wait fro messages from Alice on channel s_{AS} and forwards the content to Bob. Here the replication operator $!$ allows to serve multiple requests from Alice. Bob waits to receive the name y from the server over the channel c_{SB} and then uses y to input the message from Alice and continue as q_B (which can now use both y and m).

12.1. Syntax of π-calculus

The π-calculus has been introduced to model communicating systems where channel names, representing addresses and links, can be created and forwarded. To this aim we rely on a set of channel names $x, y, z, ...$ and extend the CCS actions with the ability to send and receive channel names. In these notes we present the monadic version of the calculus, namely the version where names can be sent only one at a time. We introduce the syntax, with productions for processes and for actions.

$$\begin{array}{rcl} p & ::= & \mathbf{nil} \mid \alpha.p \mid [x = y]p \mid p + p \mid p|p \mid (y)p \mid !p \\ \alpha & ::= & \tau \mid x(y) \mid \overline{x}y \end{array}$$

The meaning of the operators for building π-calculus processes is the following:

- **nil** is the inactive agent.
- $\alpha.p$ is an agent which can perform an action α and then act like p.
- $[x = y]p$ is the conditional process, which acts like p if $x = y$, while otherwise the process remains blocked.
- $p + q$ is the non-deterministic choice between two processes.

- $p|q$ is the parallel composition of two processes.
- $(y)p$ denotes the restriction of the channel y, which makes the name y private in p.
- $!p$ is a replicated process: it behaves as if an unbounded number of concurrent occurrences of p were available in parallel.

The meaning of the actions is the following:

- τ as usual is the invisible action.
- $x(y)$ is the input on channel x, the received value would be stored in y.
- $\overline{x}y$ is the output on channel x of the name y.

In the above case, we call x the *subject* of the communication (i.e., the channel name where the communication takes place) and y the *object* of the communication (i.e., the channel name that is transmitted or received). As in the λ-calculus, in the π-calculus we have *bound* and *free* occurrence of names. The bounding operators of π-calculus are input and restriction:

- $x(y).p$ (name y is bound in p).
- $(y)p$ (name y is bound in p).

On the contrary, the output prefix is not binding, i.e., if we take the process $\overline{x}y.p$ then the name y is said to be *free* in p. Note that for both $x(y).p$ and $\overline{x}y.p$ the name x is free in p. Moreover we define the *name set* of α as follows:

$$n(\alpha) = fn(\alpha) \cup bn(\alpha)$$

Unlike for CCS, the restriction operator $(y)p$ does not bind statically the scope of y to coincide with p. In fact in the π-calculus channel names are values, so the process p can send the name y to another process which thus becomes part of the scope of y. The possibility to enlarge the scope of a restricted name is a very useful feature of the π-calculus, called *extrusion*, which allows to modify the structure of private communications between agents.

12.2. Operational Semantics of π-calculus

Likewise CCS, we define the operational semantics by using a rule system, where well formed formulas are triples $p \xrightarrow{\alpha} q$ as for CCS. The actions α that can label the transitions are: (i) the silent action τ; (ii) the input $x(y)$ of name y on channel x; (iii) the free output $\bar{x}y$ of name y on channel x; (iv) the bound output $\bar{x}(y)$ of a previously restricted name y on channel x. The definition of free names $fn(\cdot)$, bound names $bn(\cdot)$ and names $n(\cdot)$ are extended to labels by letting:

- $fn(\tau) = \varnothing$, $fn(x(y)) = fn(\bar{x}(y)) = \{x\}$, and $fn(\bar{x}y) = \{x, y\}$;
- $bn(\tau) = bn(\bar{x}y) = \varnothing$ and $bn(x(y)) = fn(\bar{x}(y)) = \{y\}$;
- $n(\alpha) = fn(\alpha) \cup bn(\alpha)$.

$$\text{(Tau)}\ \frac{}{\tau.p \xrightarrow{\tau} p}$$

The rule (Tau) allows to perform invisible actions.

$$\text{(Out)}\ \frac{}{\overline{x}\,y.p \xrightarrow{\bar{x}\,y} p}$$

As we said the π-calculus processes can exchange messages which can contain information (i.e., channel names). The rule (Out) allows p to send the name y on the channel x.

$$(\text{In})\ \frac{}{x(y).p \xrightarrow{x(w)} p\{^{w}/_{y}\}}\ w \notin fn((y)p)$$

The rule (In) allows to receive in input over x some channel name. The received name w is bound to the name y in the process p. In order to avoid name conflicts, we assume w does not appear as a free name in $(y)p$, i.e., the transition is defined only when w is *fresh*.

$$(\text{SumL})\ \frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} \qquad (\text{SumR})\ \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'}$$

The rules (SumL) and (SumR) allow the system $p + q$ to behave as p or q.

$$(\text{Match})\ \frac{p \xrightarrow{\alpha} p'}{[x = x]p \xrightarrow{\alpha} p'}$$

The rule (Match) allows to check the condition between square bracket and unblock the process p. If the matching condition is not satisfied we can not continue the execution.

$$(\text{ParL})\ \frac{p \xrightarrow{\alpha} p'}{p|q \xrightarrow{\alpha} p'|q}\ bn(\alpha) \cap fn(q) = \emptyset \qquad (\text{ParR})\ \frac{q \xrightarrow{\alpha} q'}{p|q \xrightarrow{\alpha} p|q'}\ bn(\alpha) \cap fn(p) = \emptyset$$

As for CCS the two rules (ParL) and (ParR) allow the interleaved execution of two π-calculus agents. The side conditions guarantee that the bound names in α (if any) are fresh w.r.t. the idle process. Notice that if we assume that the bound names of α are fresh wrt. the premise of the rule, thanks to the side condition we can conclude that they are fresh also wrt. the consequence.

$$(\text{ComL})\ \frac{p \xrightarrow{\bar{x}\,z} p' \quad q \xrightarrow{x(y)} q'}{p|q \xrightarrow{\tau} p'|(q'\{^{z}/_{y}\})} \qquad (\text{ComR})\ \frac{p \xrightarrow{x(y)} p' \quad q \xrightarrow{\bar{x}\,z} q'}{p|q \xrightarrow{\tau} p'\{^{z}/_{y}\}|q'}$$

The rules (ComL) and (ComR) allow the synchronization of two parallel process. The formal name y is replaced with the actual name z in the continuation of the receiver.

$$(\text{Res})\ \frac{p \xrightarrow{\alpha} p'}{(y)p \xrightarrow{\alpha} (y)p'}\ y \notin n(\alpha)$$

The rule (Res) expresses the fact that if a name y is restricted on top of the process p, then any action which does not involve y can be performed by p.

Now we present the most important rules of π-calculus Open and Close, dealing with *scope extrusion* of channel names. Rule Open *publishes*, i.e. makes free, a private channel name, while rule Close restricts again the name, but with a broader scope.

$$(\text{Open})\ \frac{p \xrightarrow{\bar{x}\,y} p'}{(y)p \xrightarrow{\bar{x}(w)} p'\{^{w}/_{y}\}}\ y \neq x \quad w \notin fn((y)p)$$

The rule (Open) publishes the private name w, which is guaranteed to be fresh.

$$(\text{CloseL})\ \frac{p \xrightarrow{\bar{x}\,(w)} p' \quad q \xrightarrow{x(w)} q'}{p|q \xrightarrow{\tau} (w)(p'|q')} \qquad (\text{CloseR})\ \frac{p \xrightarrow{x(w)} p' \quad q \xrightarrow{\bar{x}\,(w)} q'}{p|q \xrightarrow{\tau} (w)(p'|q')}$$

The rules (CloseL) and (CloseR) transform the object of the communication over x in a private channel between p and q. Name extrusion is a convenient primitive for formalizing secure data transmission, as implemented e.g. via cryptographic protocols.

$$(\text{Rep})\ \frac{p|!p \xrightarrow{\alpha} p'}{!p \xrightarrow{\alpha} p'}$$

The last rule deals with replication. It allows to replicate a process as many times as needed, in a reentrant fashion, without consuming it. Notice that $!p$ is able also to perform the synchronizations of $p|p$, if any. We conclude this section by showing an example of the use of the rule system.

Example 12.3 (A derivation)
Let us consider the following agent:

$$(((y)\overline{x}\,y.p) \mid q) \mid x(z).r$$

The process $(y)\overline{x}\,y.p$ would like to set up a private channel with $x(z).r$, which however should remain hidden to q.

By using the rule system:

$$\begin{array}{r}
(((y)\overline{x}\,y.p) \mid q) \mid x(z).r \xrightarrow{\alpha} q_1 \quad \nwarrow_{\textit{(Close)},\ q_1=(w)(q_2|q_3),\ \alpha=\tau} \\
((y)\overline{x}\,y.p) \mid q \xrightarrow{\overline{x}(w)} q_2 \qquad x(z).r \xrightarrow{x(w)} q_3 \quad \nwarrow_{\textit{(ParL)},\ q_2=q_4|q} \\
(y)\overline{x}\,y.p \xrightarrow{\overline{x}(w)} q_4 \qquad w \notin fn(q) \qquad x(z).r \xrightarrow{x(w)} q_3 \quad \nwarrow_{\textit{(Open)},\ q_4=q_5\{{}^{w}/_{y}\}} \\
\overline{x}\,y.p \xrightarrow{\overline{x}\,y} q_5 \qquad w \notin fn(q)\ \ w \notin fn((y).p) \qquad x(z).r \xrightarrow{x(w)} q_3 \quad \nwarrow_{\textit{(Out)+(In)},\ q_3=r\{{}^{w}/_{z}\},\ q_5=p} \\
w \notin fn(q)\ \ w \notin fn((y).p)\ \ w \notin fn((z).r) \qquad\qquad\qquad
\end{array}$$

so we have:

$$\begin{array}{rcl}
q_5 & = & p \\
q_4 & = & q_5\left\{{}^{w}/_{y}\right\} = p\left\{{}^{w}/_{y}\right\} \\
q_3 & = & r\{{}^{w}/_{z}\} \\
q_2 & = & q_4 \mid q = p\left\{{}^{w}/_{y}\right\} \mid q \\
q_1 & = & (w)(q_2 \mid q_3) = (w)\left(p\left\{{}^{w}/_{y}\right\} \mid q\right) \mid (r\{{}^{w}/_{z}\})
\end{array}$$

In conclusion:

$$(((y)\overline{x}\,y.p) \mid q) \mid x(z).r \xrightarrow{\tau} (w)\left(p\left\{{}^{w}/_{y}\right\} \mid q\right) \mid (r\{{}^{w}/_{z}\})$$

under the conditions:

$$w \notin fn(q) \quad w \notin fn((y).p) \quad w \notin fn((z).r)$$

12.3. Structural Equivalence of π-calculus

As we have already noticed for CCS, there are different terms representing essentially the same process. As the complexity of the calculus increases, it is more and more convenient to manipulate terms up to some intuitive structural axioms. In the following we denote by $\equiv$ the least congruence over π-calculus processes that includes α-conversion of bound names and that is induced by the following set of axioms. The relation $\equiv$ is called *structural equivalence*.

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
p + \mathbf{nil} & \equiv & p & p + q & \equiv & q + p & (p + q) + r & \equiv & p + (q + r) \\
p \mid \mathbf{nil} & \equiv & p & p \mid q & \equiv & q \mid p & (p \mid q) \mid r & \equiv & p \mid (q \mid r) \\
(x)\,\mathbf{nil} & \equiv & \mathbf{nil} & (y)(x)p & \equiv & (x)(y)p & (x)(p \mid q) & \equiv & p \mid (x)q \ \text{ if } \ x \notin fn(p) \\
[x = y]\,\mathbf{nil} & \equiv & \mathbf{nil} & [x = x]\,p & \equiv & p & p \mid !p & \equiv & !p
\end{array}$$

12.3.1. Reduction semantics

The operational semantics of π-calculus is much more complicated than that of CCS because it needs to handle name passing and scope extrusion. By exploiting structural equivalence we can define a so-called *reduction semantics* that is simpler to understand. The idea is to define an LTS with silent labels only that models all the interactions that can take place in a process, without considering interaction with the environment. This is accomplished by first rewriting the process to a structurally equivalent normal form and then by applying basic reduction rules. In fact it can be proved that for each π-calculus process p there exists:

- a finite number of names $x_1, x_2, ..., x_k$;
- a finite number of guarded sums $s_1, s_2, ..., s_n$;
- and a finite number of processes $p_1, p_2, ..., p_m$

such that

$$P \equiv (x_1)...(x_k)(s_1|...|s_n|!p_1|...!p_m)$$

Then, a reduction is either a silent action performed by some s_i or a communication from an input prefix of say s_i with an output prefix of say s_j. We write the reduction relation as a binary relation on processes using the notation $p \mapsto q$ for indicating that p reduces to q in one step. The rules defining the relation $\mapsto$ are the following:

$$\frac{}{\tau.p + s \mapsto p} \qquad \frac{}{(x(y).p_1 + s_1)|(\bar{x}z.p_2 + s_2) \mapsto p_1 \left\{{}^{z}/{}_{y}\right\} |p_2}$$

$$\frac{p \mapsto p'}{p|q \mapsto p'|q} \qquad \frac{p \mapsto p'}{(x)p \mapsto (x)p'} \qquad \frac{p \equiv q \quad q \mapsto q' \quad q' \equiv p'}{p \mapsto p'}$$

The reduction semantics can be put in correspondence with the (silent transitions of the) labelled operational semantics by the following theorem.

Theorem 12.4 (Harmony Lemma)
For any π-calculus processes p, p' and any action α we have that:

1. $\exists q.\ p \equiv q \ \wedge\ q \xrightarrow{\alpha} p'$ *implies that* $\exists q'.\ p \xrightarrow{\alpha} q' \ \wedge\ q' \equiv p'$
2. $p \mapsto p'$ *iff* $\exists q.\ p \xrightarrow{\tau} q \ \wedge\ q \equiv p'$.

12.4. Abstract Semantics of π-calculus

Now we present an abstract semantics of π-calculus, namely we do not consider the internal structure of terms but focus on their behaviours. As we saw in CCS one of the main goals of abstract semantics is to find the correct degree of abstraction. Thus also in this case there are many kinds of bisimulations that lead to different bisimilarities, which are useful in different circumstances depending on the properties that we want to study.

We start from *strong bisimulation* of π-calculus which is an extended version of the strong bisimulation of CCS. Then we will present the *weak bisimulation* for π-calculus. An important new feature of π-calculus is the choice of the time the names used as objects of input transitions are assigned their actual values. If they are assigned *before* the choice of the (bi)simulating transition, namely if the choice of the transition may depend on the assigned value, we get the *early* bisimulation. Instead, if the choice must hold for all possible names we have the *late* bisimulation case. As we will see in short, the latter option leads to a finer semantics.

12.4.1. Strong Early Ground Bisimulations

In *early* bisimulation we require that for each name w that an agent can receive on a channel x there exists a state q' in which the bisimilar agent will be after receiving w on x. This means that the bisimilar agent can choose a different transition (and thus a different state q') depending on the observed name w. Formally, a binary relation S on π-calculus agents is a *strong early ground bisimulation* if:

$$p\ S\ q \Rightarrow \begin{cases} \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y)\ \wedge\ bn(\alpha) \notin fn(q), \text{ then } \exists q'.\ q \xrightarrow{\alpha} q' \text{ and } p'\ S\ q' \\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q), \text{ then } \forall w.\ \exists q'.\ q \xrightarrow{x(y)} q' \text{ and } p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

Two agents p and q are said to be *early bisimilar*, written $p \overset{\circ}{\sim}_E q$, iff:

$$p\ S\ q \text{ for some strong early ground bisimulation } S.$$

Notice that the conditions $bn(\alpha) \notin fn(q)$ and $y \notin fn(q)$ are required, since otherwise a bound name in the action which is fresh in p could be not fresh in q.

12.4.2. Strong Late Ground Bisimulations

In this case of late bisimulation, we require that, if an agent p can perform an input operation on a channel x, then there exists a state q' in which the bisimilar agent will be after receiving any possible value on x. Formally, a binary relation S on π-calculus agents is a *strong late ground bisimulation* if:

$$p\ S\ q \Rightarrow \begin{cases} \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y)\ \wedge\ bn(\alpha) \notin fn(q), \text{ then } \exists q'.\ q \xrightarrow{\alpha} q' \text{ and } p'\ S\ q' \\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q), \text{ then } \exists q'.\ q \xrightarrow{x(y)} q' \text{ and } \forall w.\ p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

As usual we have that two agents p and q are said to be *late bisimilar*, written $p \overset{\circ}{\sim}_L q$ iff:

$$p\ S\ q \text{ for some strong late ground bisimulation } S.$$

Let us show an example which illustrates the difference between late and early bisimilarities.

Example 12.5 (Early vs late bisimulation)
Let us consider the processes:

$$\begin{aligned} p &= x(y).\tau.\,\mathbf{nil} + x(y).\,\mathbf{nil} \\ q &= p + x(y).[y = z].\tau.\,\mathbf{nil} \end{aligned}$$

The two processes p and q are early bisimilar. In fact, let q perform an input operation on x by choosing the right branch of the + operation. Then, if the received name y is equal to z, then p can choose to perform the left input operation and reach the state $\tau.$ **nil** *which is equal to the state reached by q. Otherwise, if $y \neq z$, then the guard $[y = z]$ is not satisfied and q is blocked and p can choose to perform the right input and reach the state nil.*

On the contrary, if late bisimilarity is considered, then the two agents are not equivalent. In fact p should find a state which can handle all the possible value sent on x. If we choose to move on the left, the choice can work well when $y = z$ but not in the other cases. On the other hand, if we choose to move on the right the choice does not work well with $y = z$.

The above example shows that late bisimulation is not coarser than early. In fact, it is possible to prove that late bisimulation is strictly finer than early.

12.4.3. Strong Full Bisimilarity

Unfortunately both early and late ground bisimilarities are not congruences, even in the strong case, as shown by the following counterexample.

Example 12.6 (Ground bisimilarities are not congruences)
Let us consider the following agents:

$$p = \overline{x}\,x.\,\mathbf{nil} \mid x'(y).\,\mathbf{nil} \qquad q = \overline{x}\,x.x'(y).\,\mathbf{nil} + x'(y).\overline{x}\,x.\,\mathbf{nil}$$

The agents p and q are bisimilar (according to both weak and strong bisimulation relations), as they generate isomorphic transition systems. Now, in order to show that ground bisimulations are not congruences, we define the following context:

$$C[\;\_\;] = z(x')(\;\_\;)$$

by filling the hole of $C[\;\_\;]$ *once with p and once with q we obtain:*

$$p' = C[p] = z(x')(\overline{x}\,x.\,\mathbf{nil} \mid x'(y).\,\mathbf{nil}) \qquad q' = C[q] = z(x')(\overline{x}\,x.x'(y).\,\mathbf{nil} + x'(y).\overline{x}\,x.\,\mathbf{nil})$$

p' *and* q' *are not bisimilar. In fact, the agent* p' *can execute the input action* $z(x)$ *becoming the agent* $\overline{x}\,x.\,\mathbf{nil} \mid x(y).\,\mathbf{nil}$ *that can perform an internal synchronization* τ*;* q' *on the other hand cannot perform the same hidden action* τ *after executing the input action* $z(x)$*.*

The problem illustrated by the previous example is due to aliasing, and it appears often in programming languages with both global variables and parameter passing to procedures. It can be solved by defining a finer relation between agents called *strong early full bisimilarity* and defined as follows:

$$p \sim_E q \Leftrightarrow p\sigma \overset{\circ}{\sim}_E q\sigma \;\text{ for every substitution } \sigma$$

where a substitution σ is a function from names to names that is equal to the identity function almost everywhere (i.e. it differs from the identity function only on a finite number of elements of the domain). It is possible to define *strong late full bisimilarity* in a similar way.

12.4.4. Weak Early and Late Ground Bisimulations

As for CCS, we can define the weak versions of bisimulation relations. The definition of weak early ground bisimulation is the following:

$$p\;S\;q \Rightarrow \begin{cases} \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y) \;\wedge\; bn(\alpha) \notin fn(q), \;\; \text{then } \exists q'.\; q \overset{\alpha}{\Longrightarrow} q' \text{ and } p'\;S\;q' \\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q), \;\; \text{then } \forall w.\; \exists q'.\; q \overset{x(y)}{\Longrightarrow} q' \text{ and } p'\{^w/_y\}\;S\;q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

where here α could be τ. So we define the corresponding bisimilarity as follows:

$$p \overset{\bullet}{\approx}_E q \text{ iff } p\;S\;q \text{ for some weak early ground bisimulation } S.$$

The late version of the weak ground bisimulation is the following:

$$p\;S\;q \Rightarrow \begin{cases} \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y) \;\wedge\; bn(\alpha) \notin fn(q), \;\; \text{then } \exists q'.\; q \overset{\alpha}{\Longrightarrow} q' \text{ and } p'\;S\;q' \\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q), \;\; \text{then } \exists q'.\; q \overset{x(y)}{\Longrightarrow} q' \text{ and } \forall w.\; p'\{^w/_y\}\;S\;q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

So we define the corresponding bisimilarity as follow:

$$p \overset{\bullet}{\approx}_L q \text{ iff } p\;S\;q \text{ for some weak late ground bisimulation } S.$$

As in the strong case, weak ground bisimilarities are not congruences due to aliasing. In addition, weak bisimilarities are not congruences for a + context, as it was already the case for CCS. Both problems can be fixed by combining the solutions we have shown for weak CCS and for π-calculus strong ground bisimilarities.

Part IV.

Probabilistic Models and PEPA

13. Measure Theory and Markov Chains

In this chapter we present models of concurrent systems which make use of probability theory. In these models probability and non-determinism are used and combined in order to model concurrency, parallelism and choices.

We have seen in the previous chapters how non-determinism allows us to represent choices and parallelism. Probability can be viewed as a refinement of non-determinism. We distinguish two main cases: *stochastic* and *probabilistic* models.

In *stochastic* models each event has a duration, which is defined in terms of a probability measure. The model binds a random variable to each operation, this variable represents the time necessary to execute the operation. So in this case probabilities allow one to order the operations over the time. Most models use exponentially distributed variables, associating a rate to each event. Often in stochastic systems also the non-deterministic choice is discarded, in such systems when a race between events happens the fastest operation is executed first.

Probabilistic models use probability to represent choices. These models associate a probability to each operation. If many operations are enabled at the same time, then the system uses the probability distribution to choose the operation which will be executed. As we will see many different combinations of probability and non-determinism have been studied.

We start this chapter by introducing some basics measure theory on which we will rely in order to construct probabilistic and stochastic models. Then we will present one of the most used stochastic models, called *Markov chains*. A Markov chain, named after the Russian mathematician Andrey Markov (1856–1922), is characterized by the the fact that the probability to evolve from one state to another depends only on the current state and not on the sequence of events that preceded it (e.g., it does not depend on the states traversed before reaching the current one). This specific kind of "memorylessness" is called the *Markov property*. A Markov chain allows to predict important statistical properties about system's future. We will discuss both the discrete time and the continuous time variants of Markov chains and we will see some interesting properties which can be studied relying on probability theory.

13.1. Measure Theory

13.1.1. σ-field

We start by introducing measure theory from the core concept of σ-field. A σ-field will be the starting point to define measurable spaces and hence probability spaces.

Definition 13.1 (σ-field)
Let Ω be a set and $\mathscr{A}$ be a family of subsets of Ω, then $\mathscr{A}$ is a σ-field iff:

1. $\varnothing \in \mathscr{A}$
2. $\forall A \in \mathscr{A} \Rightarrow (\Omega \setminus A) \in \mathscr{A}$
3. $\forall A_0, \ldots, A_n, \ldots \in \mathscr{A}$ *countable sequence of sets* $\Rightarrow \bigcup_{i \in \omega} A_i \in \mathscr{A}$

It is immediate to see that $\Omega \in \mathscr{A}$ (by 1 and 2) and that, due to 2, 3 and the De Morgan property also the intersection of a countable sequence (chain) of elements of $\mathscr{A}$ is in $\mathscr{A}$, i.e., $\bigcap_{i \in \omega} A_i = \Omega \setminus (\bigcup_{i \in \omega}(\Omega \setminus A_i))$.

A set contained in $\mathscr{A}$ is usually called a *measurable set*, and the pair $(\Omega, \mathscr{A})$ is called *measurable space*.

Let us illustrate the notion of σ-field by showing a simple example over a finite set of events.

Example 13.2
Let $\Omega = \{a, b, c, d\}$ *be a set, we define a* σ*-field on* Ω *by setting* $\mathscr{A} \subseteq 2^{\Omega}$*:*

$$\mathscr{A} = \{\emptyset, \{a, b\}, \{c, d\}, \{a, b, c, d\}\}$$

Measurable spaces set the domain on which we define a particular class of functions called *measures*, which assign a real number to each measurable set of the space.

Definition 13.3 (Measure on $(\Omega, \mathscr{A})$)
Let $(\Omega, \mathscr{A})$ *be a measurable space, then a function* $\mu : \mathscr{A} \to [-\infty, +\infty]$ *is a measure iff:*

- $\mu(\emptyset) = 0$
- $\forall A \in \mathscr{A}.\ \mu(A) \geq 0$
- $\forall A_1, \ldots, A_n, \ldots \in \mathscr{A}$ *countable sequence of pairwise disjoint sets* $\mu(\bigcup_{i\in\omega} A_i) = \sum_{i\in\omega} \mu(A_i)$

We are interested to a particular class of measures called *probabilities*. A probability is a *normalized* measure.

Definition 13.4 (Probability)
A measure P *on* $(\Omega, \mathscr{A})$ *is called* sub-probability *iff* $P(\Omega) \leq 1$*. Moreover if* $P(\Omega) = 1$ *then* P *is called a* probability *on* $(\Omega, \mathscr{A})$*.*

Definition 13.5 (Probability space)
Let Ω *be a set,* $\mathscr{A}$ *be a* σ*-field on* Ω *and* P *be a probability measure on* $(\Omega, \mathscr{A})$*, then* $(\Omega, \mathscr{A}, P)$ *is called probability space.*

13.1.2. Constructing a σ-field

Obviously one can think that in order to construct a σ-field that contains some sets equipped with a probability it is enough to construct the closure of these sets (together with top and bottom elements) under complement and countable union. But it came out from set theory that this is not possible if Ω is uncountable. In fact it has been shown that it is not possible to construct (in ZFC set theory) a measure on $2^{\aleph_0}$ (i.e. a function $P : 2^{\mathbb{R}} \longrightarrow [0, 1]$). So we have to find another way to define a σ-field on spaces that are uncountable.

We use an example which shows how this problem can be solved.

Example 13.6 (Coin tosses)
Let us consider the classic coin toss experiment. We have a fair coin and we want to model sequences of coin tosses. We would like to define Ω *as follow:*

$$\Omega = \{head, tail\}^{\infty}$$

Unfortunately this set has cardinality $2^{\aleph_0}$*. As we have just said a measure on uncountable sets does not exist. So we restrict our attention on a countable set: the set* τ *of finite paths of coin tosses. In order to define a* σ*-field which allows to express almost all the events that we could express in words we define the following set for each* $\alpha \in \tau$ *called the* shadow *of* α*:*

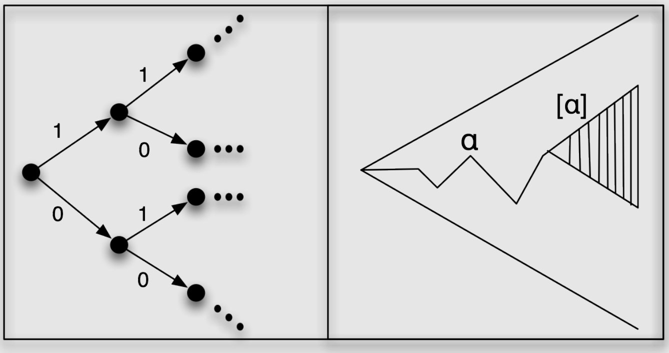
$$[\alpha] = \{\ \omega \mid \alpha\omega' = \omega,\ \textit{where } \omega \textit{ and } \omega' \textit{ are infinite paths}\ \}$$

Now the σ*-field which we were looking for is that generated by the shadows of* τ*. In this way we can start by defining a probability measure* P *on the* σ*-field generated by the shadows of* τ*, then we can define the probability* $p : \{head, tail\}^{\infty} \longrightarrow [0, 1]$ *of arbitrary sequences of coin tosses by setting:*

$$p(\alpha) = \begin{cases} P(\,[\alpha]\,) & \textit{if } \alpha \textit{ is finite} \\ P\Big(\bigcap_{\beta\in\tau,\ \alpha\in[\beta]} [\beta]\Big) & \textit{if } \alpha \textit{ is infinite} \end{cases}$$

For the second case, remind that the definition of σ-field ensures that countable intersection of measurable sets is measurable. Measure theory results show that this measure exists and is unique.

The right hand side of the picture shows graphically the set $[\alpha]$ of infinite paths corresponding to the finite path α.



Very often we have structures that are associated with a topology (e.g. there exists a standard topology associated to each CPO called Scott topology) so it is useful to define a standard method to obtain a σ-field from a topology.

Definition 13.7 (Topology)
Let X be a set and τ be a family of subsets of X. Then τ is said to be a topology *on X iff*

- $X, \emptyset \in \tau$
- $A, B \in \tau \Rightarrow A \cap B \in \tau$
- *let $\{A_i\}$ be any sequence of sets in τ then $\bigcup A_i \in \tau$*

We call A an open *set if it is in τ and* closed *set if $X \setminus A$ is open. The pair (X, τ) is said to be a* topological space.

Definition 13.8 (Borel σ-field)
Let T be a topology we will call Borel *σ-field of T the smallest σ-field that contains T.*

It turns out that the σ-field generated by the shadows which we have seen in the previous example is the Borel σ-field generated by the topology associated with the CPO of sets of infinite paths ordered by inclusion.

Definition 13.9 (Euclidean topology)
The euclidean topology *is a topology on real numbers whose open sets are open intervals of real numbers:*

$$]a, b[\quad = \quad \{x \in \mathbb{R} \mid a < x < b\}$$

We can extend the topology to the correspondent Borel σ-field, then associating to each open interval its length we obtain the Lebesgue *measure.*

13.1.3. Continuous Random Variables

As we said stochastic processes associate a *random variable* exponentially distributed to each event in order to represent its timing. So the concept of random variable and distribution will be central in next sections.

Definition 13.10 (Random variable)
Let $(\Omega, \mathscr{A}, P)$ be a probability space, a function $X : \Omega \to \mathbb{R}$ is said to be a random variable *iff*

$$\forall t \in \mathbb{R}.\ \{\omega \in \Omega \mid X(\omega) \leq t\} \in \mathscr{A}$$

Notice that if we take as $(\Omega, \mathscr{A})$ the measurable space of the real numbers with the Lebesgue measure, the identity $\mathbb{R} \to \mathbb{R}$ satisfies the above condition. As another example, we can take sequences of coin tosses and see them as binary representations of decimals in $[0, 1)$.

Random variables can be classified by considering the set of their values. We call *discrete* random variable a variable which has a numerable or finite set of possible values. We say that a random variable is *continuous* if the set of its values is continuous. Since we are particularly interested in continuous variables, in the remaining of this chapter we will consider only this type of variables. Note that almost all the definitions and theorem that we will see can be reformulated for the discrete case.

A random variable is completely characterized by its *probability law* which describes the probability that the variable will be found in a value less than or equal to the parameter.

Definition 13.11 (Cumulative distribution function)
Let $S = (\Omega, \mathscr{A}, P)$ be a probability space, $X : \Omega \to \mathbb{R}$ be a continuous random variable over S. We call cumulative distribution function *(*probability law*) of X the image of P through X and denote it by F_X:*

$$F_X(t) = P(\{\omega \in \Omega \mid X(\omega) \leq t\})$$

The other important function which describes the relative probability of a continuous random variable to take a specified value is the *probability density*.

Definition 13.12 (Probability density)
Let $S = (\Omega, \mathscr{A}, P)$ be a probability space, $X : \Omega \to \mathbb{R}$ be a continuous random variable over S, we call f the probability density *of X iff:*

$$\forall a, b \in \mathbb{R}.\ P(\{\omega \in \Omega \mid a \leq X(\omega) \leq b\}) = \int_a^b f(x)dx$$

So we can define the law F_X of a variable X with density f as follows:

$$F_X(t) = \int_{-\infty}^{t} f(x)dx$$

From now on we will write $P(X = a)$ by meaning $P(\{\omega \mid X(\omega) = a\})$ and we will write $P(a \leq X \leq b)$ by meaning $P(\{\omega \in \Omega \mid a \leq X(\omega) \leq b\})$.

As we said we are particularly interested in exponentially distributed random variables.

Definition 13.13 (Exponential distribution)
A continuous random variable X is said to be exponentially distributed *with parameter λ if its probability law and density function are defined as follows:*

$$F_X(x) = \begin{cases} 1 - e^{-\lambda x} & \text{if } x \geqslant 0 \\ 0 & x < 0 \end{cases} \qquad f(x) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x \geqslant 0 \\ 0 & x < 0 \end{cases}$$

The parameter λ is called the rate *of X.*

Random variables can be used to represent the timing of events: from now on we will interpret a variable as a function which given an event (i.e., an element of Ω) returns the time at which it is executed.

One of the most important features of exponentially distributed random variables is that they are memoryless, meaning that the current value of the random variable does not depend on the previous values. Let us show this concept with a simple example.

Example 13.14 (Radioactive Atom)
Let us consider a radioactive atom, which due to its instability can easily loose energy. It turns out that the probability that an atom will decay is constant over the time. So this system can be modelled by using an exponentially distributed continuous random variable whose rate is the decay rate of the atom. Since the random variable is memoryless we have that the probability that the atom will decay at time $t_0 + t$ knowing that it is not decaying yet at time t_0 is the same for any choice of t_0.

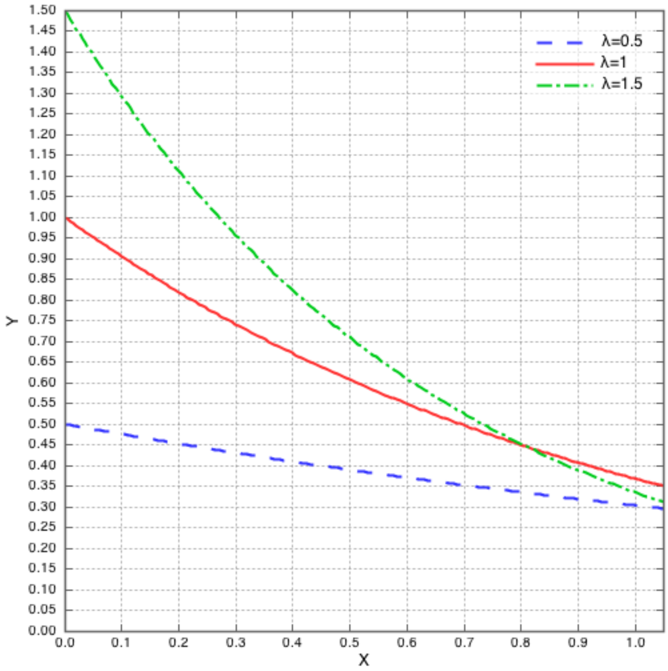


Figure 13.1.: Exponential density distributions

Theorem 13.15 (Memoryless)
Let d be an exponentially distributed continuous random variable with rate λ then we have:

$$P(d \leq t_0 + t \mid d > t_0) = P(d \leq t)$$

Proof. Since d is exponentially distributed we have that its cumulative probability is defined as follows:

$$F_d(t) = \int_0^t \lambda e^{-\lambda x} dx$$

so we want to show:

$$\frac{\int_{t_0}^{t_0+t} \lambda e^{-\lambda x} dx}{\int_{t_0}^{\infty} \lambda e^{-\lambda x} dx} \overset{?}{=} \int_0^t \lambda e^{-\lambda x} dx$$

Since $\int_a^b \lambda e^{-\lambda x} dx = \left[-e^{-\lambda x}\right]_a^b = \left[e^{-\lambda x}\right]_b^a$ thus:

$$\frac{\int_{t_0}^{t_0+t} \lambda e^{-\lambda x} dx}{\int_{t_0}^{\infty} \lambda e^{-\lambda x} dx} = \frac{\left[e^{-\lambda x}\right]_{t_0+t}^{t_0}}{\left[e^{-\lambda x}\right]_{\infty}^{t_0}} = \frac{e^{-\lambda t_0} - e^{-\lambda t} \cdot e^{-\lambda t_0}}{e^{-\lambda t_0}} = \frac{\cancel{e^{-\lambda t_0}}(1 - e^{-\lambda t})}{\cancel{e^{-\lambda t_0}}} = 1 - e^{-\lambda t}$$

We conclude by:

$$\int_0^t \lambda e^{-\lambda x} dx = \left[e^{-\lambda x}\right]_t^0 = 1 - e^{-\lambda t}$$

□

Another interesting feature of exponentially distributed random variables is the easy way in which we can compose information in order to find the probability of more complex events. For example if we have two random variables d_1 and d_2 which represent the delay of two events e_1 and e_2, we can try to calculate the probability that either of the two events will be executed before a specified time t. As we will see it happens that we can define an exponentially distributed random variable whose cumulative probability is the probability that either e_1 or e_2 executes before a specified time t.

Theorem 13.16
Let d_1 and d_2 be two exponentially distributed continuous random variables with rate respectively λ_1 and λ_2 then:

$$P(\min\{d_1, d_2\} \leq t) = 1 - e^{-(\lambda_1+\lambda_2)t}$$

Proof. Easily we have:

$$\begin{aligned}
P(\min\{d_1, d_2\} \leq t) &= P(d_1 \leq t) + P(d_2 \leq t) - P(d_1 \leq t \wedge d_2 \leq t) \\
&= (1 - e^{-\lambda_1 t}) + (1 - e^{-\lambda_2 t}) - (1 - e^{-\lambda_1 t})(1 - e^{-\lambda_2 t}) \\
&= 1 - e^{-\lambda_1 t} \cdot e^{-\lambda_2 t} \\
&= 1 - e^{-(\lambda_1+\lambda_2)t}
\end{aligned}$$

□

A second important value that we can calculate is the probability that an event will be executed before another. This corresponds in our view to calculate the probability that d_1 will take a value smaller than that of d_2.

Theorem 13.17
Let d_1 and d_2 be two exponentially distributed continuous random variables with rate respectively λ_1 and λ_2 then:

$$P(d_1 < d_2) = \frac{\lambda_1}{\lambda_1 + \lambda_2}$$

Proof. Easily we have:

$$\begin{aligned}
\int_0^\infty \lambda_1 e^{-\lambda_1 t_1} \left(\int_{t_1}^\infty \lambda_2 e^{-\lambda_2 t_2} dt_2 \right) dt_1 &= \int_0^\infty \lambda_1 e^{-\lambda_1 t_1} \left[e^{-\lambda_2 t_2} \right]_\infty^{t_1} dt_1 \\
&= \int_0^\infty \lambda_1 e^{-\lambda_1 t_1} \cdot e^{-\lambda_2 t_1} dt_1 \\
&= \int_0^\infty \lambda_1 e^{-(\lambda_1+\lambda_2)t_1} dt_1 \\
&= \left[\frac{\lambda_1}{\lambda_1 + \lambda_2} e^{-(\lambda_1+\lambda_2)t} \right]_\infty^0 \\
&= \frac{\lambda_1}{\lambda_1 + \lambda_2}
\end{aligned}$$

□

13.2. Stochastic Processes

Stochastic processes are a very powerful mathematical tool that allows us to describe and analyse a wide variety of systems.

Definition 13.18 (Stochastic process)
Let $(\Omega, \mathscr{A}, P)$ be a probability space and T be a set, then a family $\{X_t\}_{t\in T}$ of random variables over Ω is said to be a stochastic process*. A stochastic process can be identified with a function $X : \Omega \times T \longrightarrow \mathbb{R}$ such that:*

$$\forall t \in T.\ X(\_, t) : \Omega \longrightarrow \mathbb{R} \text{ is a random variable}$$

Usually the values in Ω that each random variable can take are called states *and the element of T are interpreted as times.*

Obviously the set T strongly characterizes the process. A process in which T is $\mathbb{N}$ or a subset of $\mathbb{N}$ is said to be a *discrete time* process, on the other hand if $T = \mathbb{R}$ then the process is a *continuous time* process. The same distinction is usually done on the value that each random variable can assume: if this set has a countable or finite cardinality then the process is *discrete*; otherwise it is *continuous*. We will focus only on discrete processes with both discrete and continuous time.

13.3. Markov Chains

Stochastic processes studied by classical probability theory often involve only independent variables, namely the outcomes of the process are totally independent from the past. *Markov chains* extend the classic theory by dealing with processes where each variable is influenced by the previous one. This means that in Markov processes the next outcome of the system is influenced only by the previous state. One could think to extend this theory in order to allow general dependencies between variables, but it turns out that it is very difficult to prove general results on processes with dependent variables. We are interested in Markov chains since they provide a mathematical framework to represent and analyse interleaving and sequential systems.

Definition 13.19 (Markov chain)
Let $(\Omega, \mathscr{A}, P)$ be a probability space, T be a set and $\{X_t\}_{t \in T}$ be a stochastic process. Then, $\{X_t\}_{t \in T}$ is said to be a Markov chain *if for each sequence $t_{n+1} > t_n > \ldots > t_0$ of times in T and for any measurable subset of states A (i.e., $A \in \mathscr{A}$) we have:*

$$P(X_{t_{n+1}} \in A \mid X_{t_n} = x_n, \ldots, X_{t_0} = x_0) = P(X_{t_{n+1}} \in A \mid X_{t_n} = x_n)$$

The previous proposition is usually referred to as Markov property.

An important characteristic of a Markov chain is the way in which it is influenced by the time. We have two type of Markov chains, *inhomogeneous* and *homogeneous*. In the first case the state of the system depends on the time, namely the probability distribution changes over the time. In homogeneous chains on the other hand the time does not influence the distribution, i.e., the transition probability does not change during the time. We will consider only the simpler case of homogeneous Markov chains gaining the possibility to shift the time axis back and forward.

Definition 13.20 (Homogeneous Markov chain)
Let $\{X_t\}_{t \in T}$ be a Markov chain then it is said to be homogeneous if for any measurable set A and for each $t', t \in T$ with $t' > t$ we have:

$$P(X_{t'} \in A | X_t = x) = P(X_{t'-t} \in A | X_0 = x)$$

13.3.1. Discrete and Continuous Time Markov Chain

As we said one of the most important thing about stochastic processes in general, and about Markov chains in particular, is the choice of the set of times. In this section we will introduce two kinds of Markov chain, those in which $T = \mathbb{N}$ called *discrete time Markov chain* (DTMC) and those in which $T = \mathbb{R}$ referred as *continuous time Markov chain*.

Definition 13.21 (Discrete time Markov Chain (DTMC))
Let $\{X_t\}_{t \in \mathbb{N}}$ be a stochastic process then it is a discrete time Markov chain *(DTMC) iff:*

$$P(X_{n+1} = x_{n+1} \mid X_n = x_n, \ldots, X_0 = x_0) = P(X_{n+1} = x_{n+1} \mid X_n = x_n) \text{ for } n \in \mathbb{N}$$

Since we are restricting our attention to homogeneous chains then we can reformulate the Markov property as follows:

$$P(X_{n+1} = x_{n+1} \mid X_n = x_n, \ldots, X_0 = x_0) = P(X_1 = x_{n+1} \mid X_0 = x_n)$$

Definition 13.22 (Continuous time Markov Chain (CTMC))
Let $\{X_t\}_{t\in\mathbb{R}}$ *be a stochastic process then it is a* continuous time Markov chain *(CTMC) if for any* $\Delta_t \in \mathbb{R}$ *and any sequence* $t_n + \Delta_t > t_n > \ldots > t_0$ *we have :*

$$P(X_{t_n+\Delta_t} = x \mid X_{t_n} = x_n, \ldots, X_{t_0} = x_0) = P(X_{t_n+\Delta_t} = x \mid X_{t_n} = x_n)$$

As for the discrete case the homogeneity allows to reformulate the Markov property as follows:

$$P(X_{t_n+\Delta_t} = x \mid X_{t_n} = x_n, \ldots, X_{t_0} = x_0) = P(X_{\Delta_t} = x \mid X_0 = x_n)$$

So from now on we use the term "Markov chain" as a synonym for "homogeneous Markov chain". We remark that the exponential random variable is the only continuous random variable with the memoryless property, i.e., CTMC are exponentially distributed.

13.3.2. DTMC as LTS

A DTMC can be viewed as a particular LTS whose labels are probabilities. Usually such LTS are called *probabilistic transition systems* (PTS). The PTS (S, α) associated with a DTMC has a transition function of the type $\alpha : S \longrightarrow (D(S) + 1)$ where $D(S)$ is the set of discrete probability distributions over S and $1 = \{*\}$ is a singleton representing the deadlock states. So if $\{X_t\}_{t\in\omega}$ is a DTMC whose set of states is S we can define a PTS by taking S as the set of states of the transition system and defining the transition function $\alpha : S \longrightarrow (D(S) + 1)$ as follows:

$$\alpha(s) = \begin{cases} \lambda s'.P(X_1 = s' \mid X_0 = s) & \text{if } s \text{ is not a deadlock state} \\ * & \text{otherwise} \end{cases}$$

Note that for each state s, which is not a deadlock state, it holds:

$$\sum_i \alpha(s)(s_i) = 1$$

A difference between LTS and PTS is that in LTS we can have structures like that shown in Figure 13.2 (a). In PTS we cannot have this kind of situation since two different transitions between the same pair of states have the same meaning of a single transition labeled with the sum of the probabilities, as shown in Figure 13.2 (b).

(a) (b)

Figure 13.2.: Two equivalent DTMCs

Usually the transition function is represented through a matrix P whose indices i, j represent states s_i, s_j and each element $a_{i,j}$ is the probability that knowing that the system is in the state i it would be in the state j in the next time instant, namely $\forall i, j \in\mid S \mid .\ a_{i,j} = \alpha(s_i)(s_j)$, note that in this case each row of P must sum to one. This representation allows us to study the system by relaying on linear algebra. In fact we can represent the present state of the system by using a row vector $\Pi^{(t)} = [\pi_i^{(t)}]_{i\in S}$ where $\pi_i^{(t)}$ represents the probability that the system is in the state i at the time t. If we want to calculate how the system will evolve (i.e., the next state distribution) starting from this state we can simply multiply the vector with the matrix which represents the transition function as follows:

$$\Pi^{(t+1)} = \Pi^t * P = \begin{vmatrix} \pi_1 & \pi_2 & \pi_3 \end{vmatrix} \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{vmatrix} = \begin{vmatrix} a_{1,1} * \pi_1 + a_{2,1} * \pi_2 + a_{3,1} * \pi_3 \\ a_{1,2} * \pi_1 + a_{2,2} * \pi_2 + a_{3,2} * \pi_3 \\ a_{1,3} * \pi_1 + a_{2,3} * \pi_2 + a_{3,3} * \pi_3 \end{vmatrix}^T$$

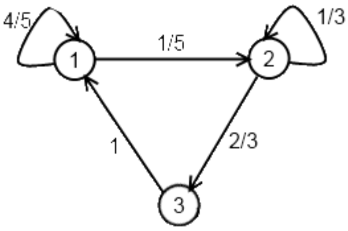


Figure 13.3.: DTMC

As we will see for some special class of DTMCs we can prove the existence of a limit vector for $t \longrightarrow \infty$, that is to say the probability that the system is found in a particular state is stationary in the long run.

Example 13.23 (DTMC)
Let us consider the DTMC in Figure 13.3. We represent the chain algebraically by using the following matrix:

$$P = \begin{vmatrix} 4/5 & 1/5 & 0 \\ 0 & 1/3 & 2/3 \\ 1 & 0 & 0 \end{vmatrix}$$

Now suppose that we do not know the state of the system at time t, thus we assume the system has probability $\frac{1}{3}$ of being in one of the three states. We represent this situation with the following vector:

$$\Pi^{(t)} = \begin{vmatrix} 1/3 & 1/3 & 1/3 \end{vmatrix}$$

Now we can calculate the state distribution at time $t + 1$ as follows:

$$\begin{vmatrix} 1/3 & 1/3 & 1/3 \end{vmatrix} \begin{vmatrix} 4/5 & 1/5 & 0 \\ 0 & 1/3 & 2/3 \\ 1 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 3/5 & 8/45 & 2/9 \end{vmatrix}$$

Notice that the sum of probabilities in the new state is again 1. Obviously we can iterate this process in order to simulate the evolution of the system.

Since we have represented a Markov chain by using a transition system it is quite natural to ask for the probability of a finite path.

Definition 13.24 (Finite path probability)
Let $\{X_t\}_{t\in\omega}$ a DTMC and $s_1, \ldots, s_n$ a finite path of its PTS (i.e. $\forall 1 \leq i < n.\ \alpha(s_i)(s_{i+1}) > 0$) we define the probability of the path as follows:

$$P(s_1\ \ldots\ s_n) = \Pi_{i=1}^{n-1}\alpha(s_i)(s_{i+1})$$

Example 13.25 (Finite paths)
Let us consider the DTMC of Example 13.23 and take the path $1\ 2\ 3\ 1$. We have:

$$P(1\ 2\ 3\ 1) = \frac{1}{5} \cdot \frac{2}{3} \cdot 1 = \frac{2}{15}$$

Note that if we consider the sequence of states $1\ 1\ 3\ 1$:

$$P(1\ 1\ 3\ 1) = \frac{4}{5} \cdot 0 \cdot 1 = 0$$

In fact there is no transition allowed from state 1 to 3.

Note that it would make little sense to define the probability of infinite paths as the product of each choice, in this case each infinite sequence would have a null probability. Obviously as we said we can avoid this problem by using the Borel σ-field generated by the shadows as we saw in the example 13.6.

Example 13.26 (Random walk with barrier)
Let us consider a DTMC with a countable set of states. Each state bigger than 0 *has two outcome transitions one which lead to the next state and one which lead to the previous one. The state* 0 *has only one arrow to the state* 1. *The structure of each state is described as follows:*

$$m \xrightarrow{1/2} m-1$$
$$m \xrightarrow{1/2} m+1$$

At the beginning we impose $P(0) = 1$, *namely the system starts at* 0. *Now it could be shown that the probability that an infinite path hits* 0 *after some time is* 1.

13.3.3. DTMC Steady State Distribution

In this section we will present a special class of DTMCs which guarantee that the probability that the system is found in a state does not change on the long term. This means that the distribution of each state of the DTMC (i.e. the correspondent value in the vector $\Pi^{(t)} = \left|\pi_1^{(t)} \dots \pi_n^{(t)}\right|$) reach a *steady state distribution* which does not change in the future, namely if π_i is the steady state distribution for the state i if $\pi_i^{(0)} = \pi_i$ then $\pi_i^{(t)} = \pi_i$ for each $t > 0$.

Definition 13.27 (Steady state distribution)
We define the steady state distribution $\Pi = \left|\pi_1 \dots \pi_n\right|$ *of a DTMC as:*

$$\pi_i = \lim_{t \longrightarrow \infty} \pi_i^{(t)}$$

when such limit exists.

In order to guarantee that the limit exists we will restrict our attention to a subclass of Markov chains.

Definition 13.28 (Ergodic Markov chain)
Let $\{X_t\}_{t \in T}$ *be a Markov chain then it is said to be* ergodic *if it has the following properties:*

- i) *irreducibility:each state is reachable from each other.*
- ii) *aperiodicity: the MCD of the lengths of all paths from any state to itself must be 1.*

Theorem 13.29
Let $\{X_t\}_{t \in T}$ *be an ergodic homogeneous Markov chain then the steady state probability* Π *always exists and it is independent from the initial state probability distribution.*

Probability distribution Π can be computed by solving the following system of linear equations: $\Pi = \Pi * P$ where P is the matrix associated to the chain, under the constraint that the sum of all probabilities be 1. It is possible to prove that the solution of such a system always exists and it is unique.

Example 13.30 (Steady state distribution)
Let us consider the DTMC of the example13.23. Let us find the steady state distribution. We have to solve the following linear system:

$$\begin{vmatrix} \pi_1 & \pi_2 & \pi_3 \end{vmatrix} \begin{vmatrix} 4/5 & 1/5 & 0 \\ 0 & 1/3 & 2/3 \\ 1 & 0 & 0 \end{vmatrix} = \begin{vmatrix} \pi_1 & \pi_2 & \pi_3 \end{vmatrix}$$

By solving the system of linear equations we obtain the solution:

$$\begin{vmatrix} 10/3 * \pi_2 & \pi_2 & 2/3 * \pi_2 \end{vmatrix}$$

i.e., $\pi_1 = \frac{10}{3}\pi_2$ *and* $\pi_3 = \frac{2}{3}\pi_2$.

Now by imposing $\pi_1 + \pi_2 + \pi_3 = 1$ *we have* $\pi_2 = 1/5$ *thus:*

$$\Pi = \begin{vmatrix} 2/3 & 1/5 & 2/15 \end{vmatrix}$$

So it is more likely to find the system in the state 1 *than in states* 2 *and* 3 *in the long run, because the steady state probability of this state is larger than the probabilities of the other two states.*

13.3.4. CTMC as LTS

Also in the continuous case Markov chains can be represented as LTSs (S, α). Labels of the LTS which represents a CTMC will be rates. We have two equivalent definitions for the transition function:

$$\alpha : S \to S \to \mathbb{R}$$

$$\alpha : (S \times S) \to \mathbb{R}$$

Where S is the set of states of the chain and $\mathbb{R}$ would represent the rate which will label the transition. Also in this case we have that two different transitions between two states would be represented by only one transition summing the rates as was for the LTS associated to a DTMC. Moreover we have that the self loops can be ignored, this is due to the fact that in continuous time we allow the system to *sojourn* in a state for a period. As for DTMCs we can represent a CTMC by using linear algebra. In this case the matrix Q which represents the system is defined by setting $q_{i,j} = \alpha(i)(j)$, this matrix is usually called *infinitesimal generator*.

The probability that no transition happens from a state i in some time r is 1 minus the probability that some transition happens, which has a rate that is the sum of the rates of all the transitions outgoing from i:

$$\forall r \in (0, \infty).\ P(X(t + r) = i \mid X(t) = i) = e^{-\lambda r} \text{ with } \lambda = \sum_{j \neq i} q_{i,j}$$

where X is the random variable describing the state of the process at a given time ($X(t) = i$ means that the process is in a state i at time t).

13.3.5. Embedded DTMC of a CTMC

Often the study of a CTMC results very hard particularly in term of computational complexity. So it is useful to have a standard way to discretize the CTMC obtaining the *embedded* DTMC in order to simplify the analysis. Let C be a CTMC whose transition function is α_C then we define the embedded DTMC D of C as follows. The set of states of D is the same as C and each element of the one-step transition probability matrix of the Embedded DTMC represents the conditional probability of transitioning from state s_i into state s_j:

$$\alpha_D(s_i)(s_j) = \begin{cases} \frac{\alpha_C(s_i)(s_j)}{\sum_{s \neq s_i} \alpha_C(s_i)(s)} & \text{if } s_i \neq s_j \\ 0 & \text{otherwise} \end{cases}$$

as we can see the previous definition simply normalizes to 1 the rates in order to calculate a probability.

While the Embedded DTMC completely determines the probabilistic behaviour of the embedded discrete-time Markov chain, it does not fully capture the behaviour of the continuous-time process because it does not specify the rates at which transitions occur.

Regarding the steady state analysis, it is easy to notice that, as in the infinitesimal generator matrix Q describing the CTMC we have $q_{i,i} = -\sum_{j \neq i} q_{i,j}$, the steady state distribution can be computed by solving the equation $\pi \cdot Q = 0$.

13.3.6. CTMC Bisimilarity

Obviously, since Markov chains can be seen as a particular type of LTS one could think to modify the notion of bisimilarity in order to study the equivalence between stochastic systems.

Let us start by defining the notion of LTS bisimilarity in a slightly different way from that seen in Chapter 10. We define a function $\gamma : S \times Act \times 2^S \longrightarrow \{true, false\}$ which takes a state p, an action μ and a set of states I and returns true if there exists a state q in I reachable from p with a transition labelled by μ, formally:

$$\gamma(p,\mu,I) = \exists q \in I.\ p \xrightarrow{\mu} q$$

Now we define the bisimilarity on a LTS as follows:

$$p\ \phi(R)\ q \overset{\text{def}}{=} (\ \forall \mu \in Act, I \in R.\ \gamma(p,\mu,I) \Leftrightarrow \gamma(q,\mu,I)\ )$$

This means that two states are said to be bisimilar if they have the same function γ whatever way we choose an equivalence class I of R and an action μ.

Now we extend this construction to a CTMC. We define a function $\gamma_M : S \times 2^S \to \mathbb{R}$ simply by extending the transition function to sets of states as follows:

$$\gamma_M(s,I) = \sum_{s' \in I} \alpha(s)(s')$$

From now on we will use α also for its extension on sets of states $\alpha(s)(I)$. As we have just done for LTS we define the bisimilarity on CTMCs as follows:

$$s_1\ \varphi\ (R) s_2 \overset{\text{def}}{=} \forall I \in R.\ \gamma_M(s_1,I) = \gamma_M(s_2,I)$$

$$\simeq\ = \bigsqcap_{R=\varphi(R)} R \qquad \text{CTMC bisimilarity}$$

meaning that the total rate of reaching from s_1 and s_2 any equivalence class of R is the same

Let us show how this construction works with an example.

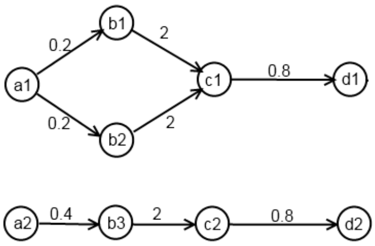


Figure 13.4.: CTMC bisimilarity

Example 13.31

Let us consider the two CTMCs in Figure 13.4. We argue that the following relation identifies the classes of bisimilar states:

$$R = \{\ \{a_1,a_2\}, \{b_1,b_2,b_3\}, \{c_1,c_2\}, \{d_1,d_2\}\ \}$$

Let us show that R is a bisimulation:

$$\gamma_M(a_1,\{b_1,b_2,b_3\}) = \gamma_M(a_2,\{b_1,b_2,b_3\}) = 0.4$$

$$\gamma_M(b_1,\{c_1,c_2\}) = \gamma_M(b_2,\{c_1,c_2\}) = \gamma_M(b_3,\{c_1,c_2\}) = 2$$

$$\gamma_M(c_1,\{d_1,d_2\}) = \gamma_M(c_2,\{d_1,d_2\}) = 0.8$$

note that we have not mentioned all remaining trivial cases, where γ_M returns 0.

13.3.7. DTMC Bisimilarity

One could think that the same argument that we have just used for CTMCs can be also extended to DTMCs. It is easy to show that if a DTMC has no deadlock states, in particular if it is ergodic, then the bisimilarity become trivial. In fact we have $\gamma_M(s, S) = \sum_{s' \in S} \alpha(s)(s') = 1$ for each state s, i.e., all states are bisimilar. This does not mean that the concept of bisimulation on ergodic DTMCs is useless, in fact these relations can be used to factorize the chain (lumping) in order to study particular properties.

On the other hand if we consider DTMCs with some deadlock states the bisimilarity is not the trivial equivalence relation. We define $\gamma_M : S \longrightarrow 2^S \to (\mathbb{R} + 1)$ as follows:

$$\gamma_M(s)(I) = \begin{cases} * & \text{if } \alpha(s) = * \\ \sum_{s' \in I} \alpha(s)(s') & \text{otherwise} \end{cases}$$

In this case we have that the bisimilarity will have an equivalence class containing exactly all deadlock states. In fact, recalling the definition of bisimulation R, we have:

$$s_1 \ \varphi\ (R) s_2 \Longrightarrow \forall I \in R.\ \gamma_M(s_1)(I) = \gamma_M(s_2)(I)$$

Therefore, any two deadlock states s_1, s_2 are bisimilar ($\forall I \in 2^S.\ \gamma_M(s_1)(I) = \gamma_M(s_2)(I) = *$) and separated from any non deadlock state s ($\forall I \in 2^S.\ \gamma_M(s_1)(I) = * \neq \gamma_M(s)(I) \in \mathbb{R}$).

14. Markov Chains with Actions and Non-determinism

In this chapter we introduce some *probabilistic models* which can be defined by modifying the transition function of PTSs. As we have seen for Markov chains, the transition system representation is very useful since it comes with a notion of bisimilarity. In fact, using the advanced, categorical notion of *coalgebra* there is a standard method to define bisimilarity just according to the type of the transition function. Also a corresponding notion of Hennessy-Milner logic can be defined accordingly.

First we will see two different ways to add actions to our probabilistic models, then we will present extensions which combine non-determinism, actions and probabilities.

14.1. Discrete Markov Chains With Actions

In this section we show how it is possible to change the transition function of the LTS in order to extend Markov chains with labels that represent actions performed by the system. There are two main cases to consider, called *reactive models* and *generative models*, respectively. In the first case we add actions that are used by the controller to stimulate the system. When we want the system to change its state we give an input action to it which could determine its future state (its reaction). This is the reason why this type of models is called "reactive". In the second case the actions would represent the outcomes of the system, this means that when the system changes its state it shows an action, whence the terminology "generative". Formally we have:

$\alpha_R : S \to L \to (D(S) + 1)$ reactive probabilistic transition system (also called Markov decision processes)
$\alpha_G : S \to (D(L \times S) + 1)$ generative probabilistic transition system

where we recall that $D(S)$ is the set of discrete probability distributions over S.

We have that in a reactive system for each $s \in S$ and for every $l \in L$:

$$\sum_{s' \in S} \alpha_R\, s\, l\, s' = 1$$

On the other hand in a generative system for each $s \in S$:

$$\sum_{(l,s') \in L \times S} \alpha_G\, s\, (l, s') = 1$$

This means that in reactive systems the probability of every action must sum to 1 (w.r.t. a given state), while in a generative system the distribution of every state must sum to 1.

14.1.1. Reactive DTMC

Let us illustrate how a reactive system works by using a simple example.

Example 14.1 ("Random" coffee maker)
Let us consider a system which we call "random" coffee maker, in which the user can insert a coin (1 or 2 euros) then, the coffee maker, based on the value of the input, chooses to make a coffee or a cappuccino with larger or smaller probabilities. The system is represented in Figure 14.1. Note that, since we want to allow the system to take input from the environment we have chosen a reactive system to represent the coffee maker. The set of labels is $L = \{1€, 2€, \textbf{Coffee}, \textbf{Cappuccino}\}$ *and the corresponding transitions are represented as dashed arrows. There are three states* s_1*,* s_2 *and* s_3*, represented with black-filled circles. If*

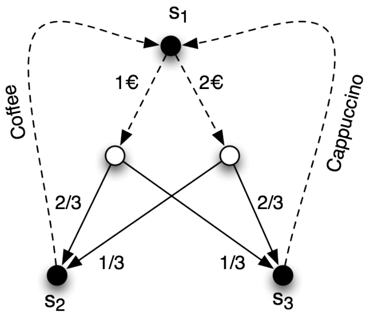


Figure 14.1.: A reactive PTS which represents a coffee maker

the input 1€ is received in state s_1, then we can reach state s_2 with probability $\frac{2}{3}$ or s_3 with probability $\frac{1}{3}$, as illustrated by the solid arrows departing from the white-filled circle associated with the distribution. Vice versa, if the input 2€ is received in state s_1, then we can reach state s_2 with probability $\frac{1}{3}$ or s_3 with probability $\frac{2}{3}$. From state s_2 there is only one transition available, with label **Coffee***, that leads to s_1 with probability 1. Here the white-filled circle is omitted because the probability distribution is trivial. Similarly, from state s_3 there is only one transition available, with label* **Cappucino***, that leads to s_1 with probability 1.*

As we said using LTS we have a standard method to define bisimilarity between probabilistic systems. A relation R between states of reactive systems is a bisimulation if for every equivalence class I of R and for each action l we have:

$$s_1 \; R \; s_2 \implies \alpha(s_1)(l)(I) = \alpha(s_2)(l)(I)$$

Note that two states s_1 and s_2 in order to be related must have for each action the same probability to reach the states in the equivalence class. Two states are said to be bisimilar if there exists a bisimulation in which they are related.

14.1.1.1. Larsen-Skou Logic

Now we will present a probabilistic version of Hennessy-Milner logic. This logic has been introduced by Larsen and Skou, and provides a new version of the modal operator. As usual we start from the syntax of the Larsen-Skou logic:

$$\varphi ::= \mathbf{true} \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \langle l\rangle_q \, \varphi$$

The novelty resides in the new modal operator $\langle l\rangle_q \, \varphi$ that takes three parameters, a formula φ, an action l and a real number $q \leq 1$. Informally, the formula $\langle l\rangle_q \, \varphi$ expresses the possibility to reach a state satisfying the formula φ by performing the action l with probability at least q.

As we have done for Hennessy-Milner logic we present the Larsen-Skou logic by defining a satisfaction relation:

$$\begin{array}{lll} s \models \mathbf{true} & & \\ s \models \varphi_1 \wedge \varphi_2 & \Leftrightarrow & s \models \varphi_1 \text{ and } s \models \varphi_2 \\ s \models \neg\varphi & \Leftrightarrow & \neg s \models \varphi \\ s \models \langle l\rangle_q \, \varphi & \Leftrightarrow & \alpha \, s \, l \, [\![\varphi]\!] \geq q \text{ where } [\![\varphi]\!] = \{s \in S \mid s \models \varphi\} \end{array}$$

A state s verifies the formula $s \models \langle l\rangle_q \, \varphi$ if the probability to pass in a state that verifies φ from s with an action labelled l is bigger than or equal to q. Note that the corresponding modal operator of the Hennessy-Milner logic can be obtained by setting $q = 1$.

As it was the case for Hennessy-Milner logic, also in this case logic formulas characterize the bisimilarity. Moreover we have an additional strong result, in fact it can be shown that it is enough to consider only the version of the logic without negation.

Theorem 14.2 (Larsen-Skou bisimilarity characterization)
Two states of a reactive transition system are bisimilar iff they satisfy the same formulas of Larsen-Skou logic without negation.

Example 14.3 (Larsen-Skou logic)
Let us show how Larsen-Skou logic works by considering the reactive system in Figure 14.1. We would like to prove the following formula:

$$s_1 \models \langle 1€\rangle_{1/2}\, \langle \mathbf{coffee} \rangle\, \mathbf{true}$$

By using the semantics we should have:

$$\alpha\, s_1\, 1€\, I_1 \geq 1/2$$

where:

$$I_1 = \{s \in S \,|\, s \models \langle \mathbf{coffee} \rangle\, \mathbf{true}\}$$

Therefore:

$$I_1 = \{s \in S \,|\alpha s\, \mathbf{coffee}\, I_2 \geq 1\}$$

where:

$$I_2 = \{s \in S \,|\, s \models \mathbf{true}\} = \{s_1, s_2, s_3\}$$

So we have:

$$I_1 = \{s \in S \,|\alpha s\, \mathbf{coffee}\{s_1, s_2, s_3\} \geq 1\} = \{s_2\}$$

Finally :

$$\alpha\, s_1\, 1€\, \{s_2\} \geq 1/2$$

14.1.2. DTMC With Non-determinism

In this section we will add non-determinism to generative and reactive systems. In this case we use non-determinism to allow the system to choose between different probability distributions. We will introduce two classes of models called *Segala automata* and *simple Segala automata*.

14.1.2.1. Segala Automata

Segala automata have been developed by Roberto Segala in 1995. They are generative systems that combine probability and non-determinism. When the system has to pass from a state to another, first of all it has to choose non-deterministically a probability distribution, then it uses this information to perform the transition. Formally the transition function is defined as follows:

$$\alpha_s : S \to \mathscr{P}(D(L \times S))$$

As we can see, the distribution is defined on pairs of labels and states. Note that in this case it is not necessary to have the singleton to represent a deadlock state since we can use the empty set.

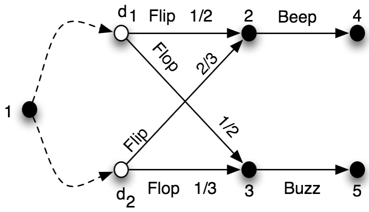


Figure 14.2.: A Segala automata

Example 14.4 (Segala automata)
Let us consider the system in Figure 14.2. We have an automata with five states (black dots). When it is in the state 1 *the system can choose non-deterministically (dotted arrows) between two different distributions* d_1 *and* d_2*:*

$$\alpha_S(1) = \{d_1, d_2\} \quad where \quad \begin{array}{ll} d_1(Flip, 2) = \frac{1}{2} & d_1(Flop, 3) = \frac{1}{2} \\ d_2(Flip, 2) = \frac{2}{3} & d_2(Flop, 3) = \frac{1}{3} \end{array}$$

$$\alpha_S(2) = \{d_3\} \quad where\ d_3(Beep, 4) = 1$$

$$\alpha_S(3) = \{d_4\} \quad where\ d_4(Buzz, 5) = 1$$

$$\alpha_S(4) = \alpha_S(5) = \varnothing$$

14.1.2.2. Simple Segala Automata

Now we present the reactive version of Segala automata. In this case we have that the system can react to an external stimulation by using a probability distribution. Since we can have more than one distribution for each label, the system uses non-determinism to choose between different distributions for the same label. Formally a *simple Segala automata* is defined as follows:

$$\alpha_{simS} : S \to \mathscr{P}(L \times D(S))$$

Example 14.5 (A Simple Segala Automata)
Let us consider the system in Figure14.3 (for some suitable probability value ϵ*). We have six states (black dots), the state* 1 *has two possible inputs, a and c, moreover the label a has two different distributions* d_1 *and* d_3*. Formally the system is defined as follows:*

$$\alpha_{simS}(1) = \{(a, d_1), (c, d_2), (a, d_3)\} \ where$$

$$\begin{array}{l} d_1(2) = d_1(3) = \frac{1}{2} \\ d_2(4) = \frac{1}{3} \quad d_2(5) = \frac{2}{3} \\ d_3(1) = \epsilon \quad d_3(6) = 1 - \epsilon \end{array}$$

$$\alpha_{simS}(2) = \alpha_{simS}(3) = \alpha_{simS}(4) = \alpha_{simS}(5) = \alpha_{simS}(6) = \varnothing$$

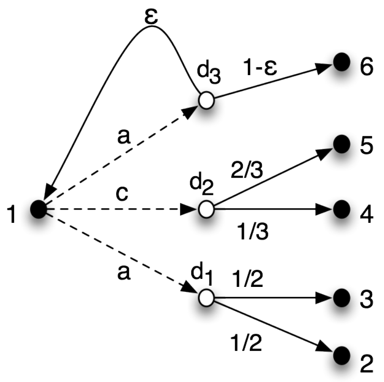


Figure 14.3.: A Simple Segala automata

14.1.2.3. Non-determinism, Probability and Actions

As we saw there are many ways to combine probability, non-determinism and actions. We conclude this chapter by mentioning two other interesting models which can be obtained by redefining the transition function of a PTS.

The first class of systems is that of *alternating* transition systems. In this case we allow the system to perform two types of transition: one using probability distributions and one using non-determinism. An alternating system can be defined formally as follows:

$$\alpha : S \longrightarrow (D(S) + \mathscr{P}(L \times S))$$

So in this kind of systems we can alternate probabilistic and non-deterministic choices.

The second type of systems that we present is that of *bundle* transition systems. In this case the system associates a distribution to subsets of non-deterministic choices. Formally:

$$\alpha : S \longrightarrow D(\mathscr{P}(L \times S))$$

So when a bundle transition system has to perform a transition, first of all it chooses by using a probability distribution a set of possible choices, then non-deterministically it picks one of these.

15. PEPA - Performance Evaluation Process Algebra

In this last chapter we introduce a language for quantitative analysis of systems called PEPA. To understand the differences between qualitative analysis and quantitative analysis, we remark that qualitative questions like:

- Will the system arrive in a particular state?
- Does the system behaviour match its specification?
- Does a given property ϕ hold within the system?

are replaced by quantitative questions like:

- How long will it take for the system to arrive in a particular state?
- With what probability does the system behaviour match its specification?
- Does a given property ϕ hold within the system within time t with probability p?

Jane Hillston defined the PEPA language in her PhD thesis in 1994. This process algebra has been developed as a high-level language for the description of continuous time Markov chains. It was obtained by extending CSP (Calculus for Sequential Processes) with probabilities. Over the years PEPA has been shown to provide an expressive formal language for modelling distributed systems. In the spirit of process algebras, PEPA models are obtained as the structure assembly of components that perform individual activities at certain rates and can cooperate on shared actions. The most important features of PEPA w.r.t. other approaches to performance modelling are:

- compositionality, i.e., the ability to model a system as the interaction of subsystems;
- formality, i.e., rigorous semantics giving a precise meaning to all terms in the language; and
- abstraction, i.e., the ability to build up complex models from detailed components, disregarding the details when it is appropriate to do so;
- separation of concerns, i.e., the ability to model the components and the interaction separately;
- structure, i.e., the ability to impose a clear structure to models, which become easy to understand;
- refinement, i.e., the ability to construct models systematically by refining their specifications;
- reusability, i.e., the ability to maintain a library of model components.

For example, queueing networks offer compositionality but not formality; stochastic extensions of Petri nets offer formality but not compositionality; neither offer abstraction mechanisms.

We will start with a brief introduction to CSP, then we will conclude with the presentation of PEPA.

15.1. CSP

Communicating Sequential Processes (CSP) is a process algebra introduced by C. A. R. Hoare in 1978 and is a very powerful tool for systems specification and verification. Contrary to CCS, CSP actions have no dual counterpart and the synchronization between two or more processes is possible when they all perform the same action α (in which case the result of the synchronization is still α). Since during communication the joint action remains visible to the environment, it can be used to interact with other (more than two) processes, realizing multiway synchronization.

15.1.1. Syntax of CSP

The syntax of CSP is the following:

$$P, Q ::= \mathbf{nil} \mid \alpha.P \mid P + Q \mid P \underset{L}{\bowtie} Q \mid P/L \qquad \text{Where } L \text{ is a set of actions}$$

nil : is the inactive process.

$\alpha.P$: is a process which can perform an action α and then behaves like P.

$P + Q$: is a process which can choose to behave like P or like Q.

$P \underset{L}{\bowtie} Q$: is a synchronization operator, also called *cooperation combinator*. It is more precisely an indexed family of operators, one for each possible set of actions L. The set L is called *cooperation set* and fixes the set of *shared actions* between P and Q. Processes P and Q can use the actions in L to synchronize each other. The actions not included in L are called *individual activities* and can be performed separately by P and Q. As a special case, if $L = \varnothing$ then all the actions of P and Q are just interleaved. Note that this operator is not associative, for example $(\alpha.\beta.\,\mathbf{nil} \underset{\{\alpha\}}{\bowtie} \mathbf{nil}) \underset{\varnothing}{\bowtie} \alpha.\,\mathbf{nil} \neq \alpha.\beta.\,\mathbf{nil} \underset{\{\alpha\}}{\bowtie} (\mathbf{nil} \underset{\varnothing}{\bowtie} \alpha.\,\mathbf{nil})$, in fact the first process can perform only an action α, the second process on the contrary can perform a synchronization on α reaching the state $\beta.\,\mathbf{nil} \underset{\{\alpha\}}{\bowtie} (\mathbf{nil} \underset{\varnothing}{\bowtie} \mathbf{nil})$ then it can perform an action β reaching $\mathbf{nil} \underset{\{\alpha\}}{\bowtie} (\mathbf{nil} \underset{\varnothing}{\bowtie} \mathbf{nil})$.

P/L : is the hiding operator. If an action α is in L then it can be performed only as a silent action τ.

15.1.2. Operational Semantics of CSP

Now we present the semantics of CSP. As we have done for CCS we define the semantics of CSP by using a rule system.

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \notin L}{P/L \xrightarrow{\alpha} P'/L} \qquad \frac{P \xrightarrow{\alpha} P' \quad \alpha \in L}{P/L \xrightarrow{\tau} P'/L}$$

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \notin L}{P \underset{L}{\bowtie} Q \xrightarrow{\alpha} P' \underset{L}{\bowtie} Q} \qquad \frac{Q \xrightarrow{\alpha} Q' \quad \alpha \notin L}{P \underset{L}{\bowtie} Q \xrightarrow{\alpha} P \underset{L}{\bowtie} Q'} \qquad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\alpha} Q' \quad \alpha \in L}{P \underset{L}{\bowtie} Q \xrightarrow{\alpha} P' \underset{L}{\bowtie} Q'}$$

15.2. PEPA

As we said PEPA is obtained by adding probabilities to CSP. As we will see there are not explicit probabilistic operators in PEPA, but the probabilistic behaviour is obtained by associating an exponentially distributed continuous random variable to each action, this random variable will represent the time needed to execute the action. These random variables lead to a clear relationship between the process algebra model and a CTMC. Via this underlying Markov process performance measures can then be extracted from the model.

15.2.1. Syntax of PEPA

In PEPA an action is a pair (α, r) where α is the action type and r is the rate of the continuous random variable associated with the action. The rate r can be any positive real number. We have the following grammar:

$$P, Q ::= \mathbf{nil} \mid (\alpha, r).P \mid P + Q \mid P \underset{L}{\bowtie} Q \mid P/L \mid C$$

$(\alpha, r).P$: is a process which can perform an action α and then behaves like P. In this case the rate r is used to define the exponential variable which describes the duration of the action. A component may have a purely sequential behaviour, repeatedly undertaking one activity after another and eventually returning to its initial state. As a simple example, consider a web server in a distributed system that can serve one request at a time:

$$WS \overset{\text{def}}{=} (request, \top).(serve, \mu).(respond, \top).WS$$

In some cases, as here, the rate of an action falls out of the control of this component: such actions are carried out jointly with another component, with the current component playing some sort of passive role. For example, the web server is passive with respect to the request and respond actions, as it cannot influence the rate at which application execute these actions. This is recorded by using the distinguished rate $\top$ which we can assume to represent an extremely high value that cannot influence the rate of interacting components.

$P + Q$: has the same meaning of the CSP operator for choice. For example, we can consider an application in a distributed system that can either access a locally available method (with probability p_1) or access to a remote web service (with probability $p_2 = 1 - p_1$). The decision is taken by performing a think action which is parametric to the rate λ:

$$\begin{array}{rcl} Appl & \overset{\text{def}}{=} & (think, p_1 \cdot \lambda).(local, m).Appl \\ & + & (think, p_2 \cdot \lambda).(request, rq).(respond, rp).Appl \end{array}$$

$P \underset{L}{\bowtie} Q$: has the same meaning of the CSP operator. In the web service example, we can assume that the application and the web server interact overthe set of shared actions $L = \{request, respond\}$:

$$Sys \overset{\text{def}}{=} (Appl \underset{\emptyset}{\bowtie} Appl) \underset{L}{\bowtie} WS$$

During the interaction, the resulting action will have the same type of the shared action and a rate reflecting the rate of the slowest action.

P/L : is the same of the CSP hiding operator: the duration of the action is unaffected, but its type becomes hidden. In our running example, we may want to hide the local computation of $Appl$ to the environment:

$$Appl' \overset{\text{def}}{=} Appl/\{local\}$$

C : is the name of a recursive process which is defined as $C \overset{\text{def}}{=} P$ in a separate set Δ of declarations. Using recursive definition as the ones given above for $Appl$ and WS we are able to describe components with infinite behaviour without introducing an explicit recursion or replication operator.

Usually we are interested only in those agents which have an ergodic underlying Markov process, this because we want to apply the steady state analysis. It has been shown that it is possible to ensure the ergodicity by using syntactic restrictions on the agents. In particular, the class of PEPA terms which satisfy these syntactic conditions are called *cyclic components* and they can be described by the following grammar:

$$\begin{array}{rcl} P, Q & ::= & S \mid P \underset{L}{\bowtie} Q \mid P/L \\ S, T & ::= & (\alpha, r).S \mid S + T \mid C \end{array}$$

where each recursive process C is sequential, i.e., $C \overset{\text{def}}{=} S$ for some sequential process S.

15.2.2. Operational Semantics of PEPA

We give the PEPA operational semantics by using the following rule system whose well formed formulas have the form $P \xrightarrow{(\alpha,r)} Q$ for suitable PEPA processes P and Q, activity α and rate r. We assume a set Δ of declarations.

$$\frac{}{(\alpha, r).P \xrightarrow{(\alpha,r)} P} \qquad \frac{P \xrightarrow{(\alpha,r)} P'}{P + Q \xrightarrow{(\alpha,r)} P'} \qquad \frac{Q \xrightarrow{(\alpha,r)} Q'}{P + Q \xrightarrow{(\alpha,r)} Q'}$$

$$\frac{P \xrightarrow{(\alpha,r)} P' \quad \alpha \notin L}{P/L \xrightarrow{(\alpha,r)} P'/L} \qquad \frac{P \xrightarrow{(\alpha,r)} P' \quad \alpha \in L}{P/L \xrightarrow{(\tau,r)} P'/L}$$

$$\frac{P \xrightarrow{(\alpha,r)} P' \quad \alpha \notin L}{P \underset{L}{\bowtie} Q \xrightarrow{(\alpha,r)} P' \underset{L}{\bowtie} Q} \qquad \frac{Q \xrightarrow{(\alpha,r)} Q' \quad \alpha \notin L}{P \underset{L}{\bowtie} Q \xrightarrow{(\alpha,r)} P \underset{L}{\bowtie} Q'}$$

$$\frac{P \xrightarrow{(\alpha,r_1)} P' \quad Q \xrightarrow{(\alpha,r_2)} Q' \quad \alpha \in L}{P \underset{L}{\bowtie} Q \xrightarrow{(\alpha,r)} P' \underset{L}{\bowtie} Q'} \qquad \text{where } r = \min(r_\alpha(P), r_\alpha(Q)) * \frac{r_1}{r_\alpha(P)} * \frac{r_2}{r_\alpha(Q)}$$

$$\frac{P \xrightarrow{(\alpha,r)} P' \quad (C \overset{\text{def}}{=} P) \in \Delta}{C \xrightarrow{(\alpha,r)} P'}$$

The only rule that deserves some explanation is the cooperation rule, where we have denoted by $r_\alpha(P)$ the *apparent rate* of action α in P, which is defined by structural recursion as follows:

$$\begin{array}{rcl}
r_\alpha(\mathbf{nil}) & = & 0 \\
r_\alpha((\beta, r).P) & = & \begin{cases} r & \text{if } \alpha = \beta \\ 0 & \text{if } \alpha \neq \beta \end{cases} \\
r_\alpha(P + Q) & = & r_\alpha(P) + r_\alpha(Q) \\
r_\alpha(P/L) & = & \begin{cases} r_\alpha(P) & \text{if } \alpha \notin L \\ 0 & \text{if } \alpha \in L \end{cases} \\
r_\alpha(P \underset{L}{\bowtie} Q) & = & \begin{cases} min(r_\alpha(P), r_\alpha(Q)) & \text{if } \alpha \in L \\ r_\alpha(P) + r_\alpha(Q) & \text{if } \alpha \notin L \end{cases}
\end{array}$$

Roughly, the apparent rate $r_\alpha(S)$ is the sum of the rates of all distinct actions α that can be performed by S, thus $r_\alpha(S)$ expresses the overall rate of α in S (because of the property of rates of exponentially distributed variables). Notably, in the case of shared actions $P \underset{L}{\bowtie} Q$ the apparent rate is the slower of the apparent rates of P and Q.

The selection of the exponential distribution as the governing distribution for action durations in PEPA has profound consequences. In terms of the underlying stochastic process, it is the only choice which gives rise to a Markov process. This is due to the memoryless properties of the exponential distribution: the time until the next event is independent of the time since the last event, because the exponential distribution forgets how long it has already waited. For instance, if we consider the process $(\alpha, r).\,\mathbf{nil} \underset{\emptyset}{\bowtie} (\beta, s).\,\mathbf{nil}$ and the system performs the action α, the time needed to complete β from $\mathbf{nil} \underset{\emptyset}{\bowtie} (\beta, s).\,\mathbf{nil}$ does not need to consider the time already taken to carry out the action α.

Let us now explain the calculation $r = \min(r_\alpha(P), r_\alpha(Q)) * \frac{r_1}{r_\alpha(P)} * \frac{r_2}{r_\alpha(Q)}$ that appears in the cooperation rule. The best way to resolve what should be the rate of the shared action has been a topic of some debate. The definition of cooperation in PEPA is based on the assumption that a component cannot be made to exceed its bounded capacity for carrying out the shared actions, where the bounded capacity consists of the apparent rate of the action. The underlying assumption is that the choice of a specific action (with rate r_i) to carry on the shared activity occurs independently in the two cooperating components P and Q. Now, the probability that a certain action (α, r_1) (respectively, (α, r_2)) is chosen by P is $\frac{r_1}{r_\alpha(P)}$ (respectively, $\frac{r_2}{r_\alpha(Q)}$ in the case of Q). Then, from the independence of the choices in P and Q we obtain the combined probability $\frac{r_1}{r_\alpha(P)} * \frac{r_2}{r_\alpha(Q)}$. Finally, the resulting rate is the product of the apparent rate $\min(r_\alpha(P), r_\alpha(Q))$ and the above probability. Notice that if we sum up the rates of all possible synchronizations on α of $P \underset{L}{\bowtie} Q$ we just get $min(r_\alpha(P), r_\alpha(Q))$ as stated above. See the example below.

Example 15.1

Let us define two PEPA agents as follows:

$$P = (\alpha, r).P_1 + \ldots + (\alpha, r).P_n$$

$$Q = (\alpha, r).Q_1 + \ldots + (\alpha, r).Q_m$$

We assume $n \leq m$:

$$P \xrightarrow{(\alpha,r)} P_i \qquad i = 1, \ldots, n$$

$$Q \xrightarrow{(\alpha,r)} Q_i \qquad i = 1, \ldots, m$$

So we have the the following apparent rates:

$$r_\alpha(P) = n * r$$

$$r_\alpha(Q) = m * r$$

Then considering the synchronization between P and Q, we have:

$$P \underset{\{\alpha\}}{\bowtie} Q \xrightarrow{(\alpha,r')} P_i \underset{\{\alpha\}}{\bowtie} Q_j \qquad i = 0 \ldots n \qquad j = 1 \ldots m$$

Where $r' = \frac{r}{nr} * \frac{r}{mr} * nr = \frac{r}{m}$. The apparent rate of the synchronization is:

$$r_\alpha(P \underset{\{\alpha\}}{\bowtie} Q) = m * n * r' = m * n * \frac{r}{m} = n * r = min(r_\alpha(P), r_\alpha(Q))$$

The underlying CTMC is obtained from the LTS by associating a state with each process, and the transitions between states are derived from the transitions of the LTS. Since all activity durations are exponentially distributed, the total transition rate between two states will be the sum of the activity rates labelling arcs connecting the corresponding nodes in the LTS.

We conclude this section by showing an example of process modelled by using PEPA.

Example 15.2 (Roland the gunman)

We want to model a far west duel. We have two main characters: Roland the gunman and his enemies. Upon its travels Roland will encounter some enemies with whom he will have no choice but to fight back. For simplicity we assume that Roland has two guns with one bullet in each and that each hit is fatal. We also assume that a sense of honour prevents an enemy from attacking Roland if he is already involved in a gun fight. We model the behaviour of Roland as follows. Normally, Roland is in an idle state $Roland_{idle}$, but when he is attacked (attacks) he moves to state $Roland_2$, where he has two bullets available in his gun:

$$Roland_{idle} \stackrel{\text{def}}{=} (attack, \top).Roland_2$$

In front of his enemies, Roland can act in three ways: if he hits his enemies then he reloads his gun and returns idle; if he misses the enemies he tries a second attack (see $Roland_1$); finally if an enemy hits him, *Roland dies.*

$$\begin{aligned} Roland_2 \stackrel{\text{def}}{=} & (hit, r_{hit}).(reload, r_{reload}).Roland_{idle} \\ & +(miss, r_{miss}).Roland_1 \\ & +(e\text{-}hit, \top).Roland_{dead} \end{aligned}$$

The second attempt to shoot by Roland is analogous to the first one, but this time it is the last bullet in Roland's gun and if the enemy is missed no further shot is possible in $Roland_{empty}$ *until the gun is reloaded.*

$$\begin{aligned} Roland_1 & \stackrel{\text{def}}{=} (hit, r_{hit}).(reload, r_{reload}).Roland_{idle} \\ & \quad +(miss, r_{miss}).Roland_{empty} \\ & \quad +(e\text{-}hit, \top).Roland_{dead} \\ Roland_{empty} & \stackrel{\text{def}}{=} (reload, r_{reload}).Roland_2 \\ & \quad +(e\text{-}hit, \top).Roland_{dead} \end{aligned}$$

Finally if Roland is dead he cannot perform any action.

$$Roland_{dead} \stackrel{\text{def}}{=} \mathbf{nil}$$

We describe enemies behaviour as follows. If the enemies are idle they can try to attack Roland:

$$Enemies_{idle} \stackrel{\text{def}}{=} (attack, r_{attack}).Enemies_{attack}$$

Enemies shoot once and either get hit or they hit Roland.

$$\begin{aligned} Enemies_{attack} \stackrel{\text{def}}{=} & (e\text{-}hit, r_{e\text{-}hit}).Enemies_{idle} \\ & +(hit, \top).Enemies_{idle} \end{aligned}$$

The rates involved in the model are measured in seconds, so a rate of 1.0 *would indicate that the action is expected to occur once every second. We define the following rates:*

$$\begin{array}{ccc} \top & = \text{ about } \infty & \\ r_{fire} & = 1 & \text{one shot per second} \\ r_{hit\text{-}success} & = 0.8 & \text{80\% of success} \\ r_{hit} & = 0.8 & r_{fire} * r_{hit\text{-}success} \\ r_{miss} & = 0.2 & r_{fire} * (1 - r_{hit\text{-}success}) \\ r_{reload} & = 0.3 & \text{3 seconds to reload} \\ r_{attack} & = 0.01 & \text{Roland is attacked once every 100 seconds} \\ r_{e\text{-}hit} & = 0.02 & \text{Enemies can hit once every 50 seconds} \end{array}$$

So we model the duel as follows:

$$Duel \stackrel{\text{def}}{=} Roland_{idle} \underset{\{hit,attack,e-hit\}}{\bowtie} Enemies_{idle}$$

We can perform various types of analysis of the system by using standard methods. Using the steady state analysis, that we have seen in the previous chapters, we can prove that Roland will always die and the system will deadlock, because there is an infinite supply of enemies (so the system is not ergodic). Moreover we can answer many other questions by using the following techniques:

- *Transient analysis: we can ask for the probability that Roland is dead after 1 hour, or the probability that Roland will have killed some enemy within 30 minutes.*
- *Passage time analysis: we can ask for the probability of passing at least 10 seconds from the first attack to Roland to the time it has hit 3 enemies, or the probability that 1 minute after he is attacked Roland has killed his attacker (i.e., the probability that the model performs a hit action within 1 minute after having performed an attack action).*

The PEPA language is supported by a range of tools and by a wide community of users. PEPA application areas span the subject areas of informatics and engineering including, e.g., cellular telephone networks, database systems, diagnostic expert systems, multiprocessor access-contention protocols, protocols for fault-tolerant systems, software architectures. Additional information and a PEPA Eclipse Plug-in are freely available at `http://www.dcs.ed.ac.uk/pepa/`.

Part V.

Appendices

A. Summary

A.1. Induction rules 3.1.2

A.1.1. Noether

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

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

A.1.2. Weak Mathematical Induction 3.1.3

Let P be a unary predicate over ω.

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

A.1.3. Strong Mathematical Induction 3.1.4

Let P be a unary predicate over ω.

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

A.1.4. Structural Induction 3.1.5

Let Σ be a signature, T_Σ be the set of terms over Σ and P be a property defined on T_Σ.

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

A.1.5. Derivation Induction 3.1.6

Let R be a set of inference rules and D the set of derivations defined on R. We define:

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

where $d_1, \ldots, d_n$ are derivation for $x_1, \ldots, x_n$.

A.1.6. Rule Induction 3.1.7

Let R be a set of rules and I_R the set of theorems of R.

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

A.1.7. Computational Induction 5.4

Let P be a property, $(D, \sqsubseteq)$ a $CPO_\perp$ and F a monotone, continuous function on it. We define:

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

A.2. IMP 2

A.2.1. IMP Syntax 2.1

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

A.2.2. IMP Operational Semantics 2.2

A.2.2.1. IMP Arithmetic Expressions

$$
\frac{}{\langle x, \sigma \rangle \to \sigma(x)}\ (\text{ide}) \qquad \frac{}{\langle n, \sigma \rangle \to n}\ (\text{num}) \qquad \frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n_0 + n_1}\ (\text{sum})
$$

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

A.2.2.2. IMP Boolean Expressions

$$
\frac{}{\langle v, \sigma \rangle \to v}\ (\text{bool}) \qquad \frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 = a_1, \sigma \rangle \to (n_0 = n_1)}\ (\text{equ}) \qquad \frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 \leq a_1, \sigma \rangle \to (n_0 \leq n_1)}\ (\text{leq})
$$

$$
\frac{\langle b, \sigma \rangle \to v}{\langle \neg b, \sigma \rangle \to \neg v}\ (\text{not}) \qquad \frac{\langle b_0, \sigma \rangle \to v_0 \quad \langle b_1, \sigma \rangle \to v_1}{\langle b_0 \vee b_1, \sigma \rangle \to (v_0 \vee v_1)}\ (\text{or}) \qquad \frac{\langle b_0, \sigma \rangle \to v_0 \quad \langle b_1, \sigma \rangle \to v_1}{\langle b_0 \wedge b_1, \sigma \rangle \to (v_0 \wedge v_1)}\ (\text{and})
$$

A.2.2.3. IMP Commands

$$
\frac{}{\langle \mathbf{skip}, \sigma \rangle \to \sigma}\ (\text{skip}) \qquad \frac{\langle a, \sigma \rangle \to m}{\langle x := a, \sigma \rangle \to \sigma[^m/_x]}\ (\text{assign})
$$

$$
\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}\ (\text{seq}) \qquad \frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma \rangle \to \sigma'}\ (\text{iftt}) \qquad \frac{\langle b, \sigma \rangle \to \mathbf{false} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma \rangle \to \sigma'}\ (\text{ifff})
$$

$$
\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle c, \sigma \rangle \to \sigma'' \quad \langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma'' \rangle \to \sigma'}{\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma \rangle \to \sigma'}\ (\text{whtt}) \qquad \frac{\langle b, \sigma \rangle \to \mathbf{false}}{\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma \rangle \to \sigma}\ (\text{whff})
$$

A.2.3. IMP Denotational Semantics 5

A.2.3.1. IMP Arithmetic Expressions $\mathscr{A} : Aexpr \to (\Sigma \to \mathbb{N})$

$$
\begin{array}{rcl}
\mathscr{A}\, \llbracket n \rrbracket\, \sigma & = & n \\
\mathscr{A}\, \llbracket x \rrbracket\, \sigma & = & \sigma x \\
\mathscr{A}\, \llbracket a_0 + a_1 \rrbracket\, \sigma & = & (\mathscr{A}\, \llbracket a_0 \rrbracket\, \sigma) + (\mathscr{A}\, \llbracket a_1 \rrbracket\, \sigma) \\
\mathscr{A}\, \llbracket a_0 - a_1 \rrbracket\, \sigma & = & (\mathscr{A}\, \llbracket a_0 \rrbracket\, \sigma) - (\mathscr{A}\, \llbracket a_1 \rrbracket\, \sigma) \\
\mathscr{A}\, \llbracket a_0 \times a_1 \rrbracket\, \sigma & = & (\mathscr{A}\, \llbracket a_0 \rrbracket\, \sigma) \times (\mathscr{A}\, \llbracket a_1 \rrbracket\, \sigma)
\end{array}
$$

A.2.3.2. IMP Boolean Expressions $\mathscr{B} : Bexpr \to (\Sigma \to \mathbb{B})$

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

A.2.3.3. IMP Commands $\mathscr{C} : Com \to (\Sigma \rightharpoonup \Sigma)$

$$\begin{aligned}
\mathscr{C}[\![\mathbf{skip}]\!]\sigma &= \sigma \\
\mathscr{C}[\![x := a]\!]\sigma &= \sigma\left[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}\right] \\
\mathscr{C}[\![c_0; c_1]\!]\sigma &= \mathscr{C}[\![c_1]\!]^*(\mathscr{C}[\![c_0]\!]\sigma) \\
\mathscr{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]\sigma &= \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma \\
\mathscr{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] &= fix\,\Gamma = \bigsqcup_{n\in\omega}\Gamma^n(\bot_{\Sigma\to\Sigma_\bot})
\end{aligned}$$

where $\Gamma : (\Sigma \to \Sigma_\bot) \to \Sigma \to \Sigma_\bot$ is defined as follows:

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

A.3. HOFL 6.1

A.3.1. HOFL Syntax 6.1

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

A.3.2. HOFL Types 6.1.1

$$\tau ::= int \mid \tau * \tau \mid \tau \to \tau$$

A.3.3. HOFL Typing Rules 6.1.1

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

$$\frac{t_0 : \tau_0 \quad t_1 : \tau_1}{(t_0, t_1) : \tau_0 * \tau_1} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{fst}(t) : \tau_0} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{snd}(t) : \tau_1}$$

$$\frac{x : \tau_0 \quad t : \tau_1}{\lambda x.t : \tau_0 \to \tau_1} \qquad \frac{t_1 : \tau_0 \to \tau_1 \qquad t_0 : \tau_0}{(t_1\ t_0) : \tau_1}$$

$$\frac{x : \tau \quad t : \tau}{\mathbf{rec}\ x.t : \tau}$$

A.3.4. HOFL Operational Semantics 6.2

A.3.4.1. HOFL Canonical Forms

$$\frac{\varnothing}{n \in C_{int}} \qquad \frac{t_0 : \tau_0 \quad t_1 : \tau_1 \quad t_0, t_1 \text{ closed}}{(t_0, t_1) \in C_{\tau_0 * \tau_1}} \qquad \frac{\lambda x.t : \tau_0 \to \tau_1 \quad \lambda x.t \text{ closed}}{\lambda x.t \in C_{\tau_0 \to \tau_1}}$$

A.3.4.2. HOFL Axiom

$$\frac{}{c \to c}$$

A.3.4.3. HOFL Arithmetic and Conditional Expressions

$$\frac{t_0 \to n_0 \quad t_1 \to n_1}{t_0 \text{ op } t_1 \to n_0 \ \underline{op}\ n_1} \qquad \frac{t_0 \to 0 \quad t_1 \to c_1}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to c_1} \qquad \frac{t_0 \to n \quad n \neq 0 \quad t_2 \to c_2}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to c_2}$$

A.3.4.4. HOFL Pairing Rules

$$\frac{t \to (t_0, t_1) \quad t_0 \to c_0}{\textbf{fst}(t) \to c_0} \qquad \frac{t \to (t_0, t_1) \quad t_1 \to c_1}{\textbf{snd}(t) \to c_1}$$

A.3.4.5. HOFL Function Application

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[^{t_0}/_x] \to c}{(t_1 \quad t_0) \to c} \quad \text{(lazy)}$$

$$\frac{t_1 \to \lambda x.t_1' \quad t_0 \to c_0 \quad t_1'[^{c_0}/_x] \to c}{(t_1 \quad t_0) \to c} \quad \text{(eager)}$$

A.3.4.6. HOFL Recursion

$$\frac{t[^{\textbf{rec}\, x.t}/_x] \to c}{\textbf{rec}\, x.t \to c}$$

A.3.5. HOFL Denotational Semantics 8 $\llbracket t : \tau \rrbracket : Env \longrightarrow (V_\tau)_\perp$

$$\begin{aligned}
\llbracket n \rrbracket \rho &= \lfloor n \rfloor \\
\llbracket x \rrbracket \rho &= \rho x \\
\llbracket t_0 \textbf{ op } t_1 \rrbracket \rho &= \llbracket t_0 \rrbracket \rho \ \underline{op}_\perp\ \llbracket t_1 \rrbracket \rho \\
\llbracket \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \rrbracket \rho &= Cond\,(\llbracket t_0 \rrbracket \rho, \llbracket t_1 \rrbracket \rho, \llbracket t_2 \rrbracket \rho) \\
\llbracket (t_0, t_1) \rrbracket \rho &= \lfloor (\llbracket t_0 \rrbracket \rho, \llbracket t_1 \rrbracket \rho) \rfloor \\
\llbracket \textbf{fst}(t) \rrbracket \rho &= \textbf{let}\, v \Leftarrow \llbracket t \rrbracket \rho. \quad \pi_1 v \\
\llbracket \textbf{snd}(t) \rrbracket \rho &= \textbf{let}\, v \Leftarrow \llbracket t \rrbracket \rho. \quad \pi_2 v \\
\llbracket \lambda x.t \rrbracket \rho &= \left\lfloor \lambda d.\, \llbracket t \rrbracket \rho[^d/_x] \right\rfloor \\
\llbracket (t_1 \ t_0) \rrbracket &= \textbf{let}\, \varphi \Leftarrow \llbracket t_1 \rrbracket \rho.\varphi(\llbracket t_0 \rrbracket \rho) \\
\llbracket \textbf{rec}\ \ x.t \rrbracket \rho &= \textbf{fix}\, \lambda d.\, \llbracket t \rrbracket \rho[^d/_x]
\end{aligned}$$

A.4. CCS 10

A.4.1. CCS Syntax 10.1

$$p, q ::= x \quad | \quad \mathbf{nil} \quad | \quad \mu.p \quad | \quad p\backslash\alpha \quad | \quad p[\phi] \quad | \quad p+p \quad | \quad p \mid p \quad | \quad \mathbf{rec}\, x.p$$

A.4.2. CCS Operational Semantics 10.2

$$(\text{Act}) \ \frac{}{\mu.p \xrightarrow{\mu} p} \qquad (\text{Res}) \ \frac{p \xrightarrow{\mu} q}{p\backslash\alpha \xrightarrow{\mu} q\backslash\alpha}\ \mu \neq \alpha, \bar{\alpha} \qquad (\text{Rel}) \ \frac{p \xrightarrow{\mu} q}{p[\phi] \xrightarrow{\phi(\mu)} q[\phi]}$$

$$(\text{Sum}) \ \frac{p \xrightarrow{\mu} p'}{p + q \xrightarrow{\mu} p'} \qquad \frac{q \xrightarrow{\mu} q'}{p + q \xrightarrow{\mu} q'}$$

$$(\text{Com}) \ \frac{p \xrightarrow{\mu} p'}{p|q \xrightarrow{\mu} p'|q} \qquad \frac{q \xrightarrow{\mu} q'}{p|q \xrightarrow{\mu} p|q'} \qquad \frac{p_1 \xrightarrow{\lambda} p_2 \quad q_1 \xrightarrow{\bar{\lambda}} q_2}{p_1|q_1 \xrightarrow{\tau} p_2|q_2}$$

$$(\text{Rec}) \ \frac{p[\mathbf{rec}\, x.p/x] \xrightarrow{\mu} q}{\mathbf{rec}\, x.p \xrightarrow{\mu} q}$$

A.4.3. CCS Abstract Semantics 10.3

A.4.3.1. CCS Strong Bisimulation 10.3.3

$$p \ \Phi(R)\ q \overset{\text{def}}{=} \begin{cases} p \xrightarrow{\mu} p' & \text{implies} \quad \exists q'.\ q \xrightarrow{\mu} q' \quad \text{and} \quad p'Rq' \\ q \xrightarrow{\mu} q' & \text{implies} \quad \exists p'.\ p \xrightarrow{\mu} p' \quad \text{and} \quad p'Rq' \end{cases}$$

A.4.3.2. CCS Weak Bisimulation 10.7

The weak transition relation $\Rightarrow$is defined as follows:

$$\begin{aligned} p \overset{\tau}{\Rightarrow} q \quad & \text{iff} \quad p \xrightarrow{\tau} \ldots \xrightarrow{\tau} q \ \vee \ p = q \\ p \overset{\lambda}{\Rightarrow} q \quad & \text{iff} \quad p \overset{\tau}{\Rightarrow} p' \xrightarrow{\lambda} q' \overset{\tau}{\Rightarrow} q \end{aligned}$$

A weak bisimulation is defined as follows:

$$p \ \Psi(R)\ q \overset{\text{def}}{=} \begin{cases} p \xrightarrow{\mu} p' & \text{implies} \quad \exists q'.\ q \overset{\mu}{\Rightarrow} q' \quad \text{and} \quad p'\ R\ q' \\ q \xrightarrow{\mu} q' & \text{implies} \quad \exists p'.\ p \overset{\mu}{\Rightarrow} p' \quad \text{and} \quad p'\ R\ q' \end{cases}$$

A.4.3.3. CCS Observational Congruence 10.7.2

$$p \cong q \overset{\text{def}}{=} \begin{cases} p \xrightarrow{\tau} p' & \text{implies} \quad \exists q'.\ q \xrightarrow{\tau} \overset{\tau}{\Longrightarrow} q' \quad \text{and} \quad p' \approx q' \\ p \xrightarrow{\lambda} p' & \text{implies} \quad \exists q'.\ q \overset{\lambda}{\Longrightarrow} q' \quad \text{and} \quad p' \approx q' \\ \text{(and vice versa)} & \end{cases}$$

Alternatively $\cong$ can be defined as follows:

$$p \cong q \ \text{ iff } \ p \approx q \ \wedge \ \forall r.\ p + r \approx q + r$$

A.4.3.4. CCS Axioms for Observational Congruence (Milner τ Laws) 10.7.2

$$\begin{aligned}
&p + \tau.p = \tau.p\\
&\mu.(p + \tau.q) = \mu.(p + \tau.q) + \mu.q\\
&\mu.\tau.p = \mu.p
\end{aligned}$$

A.4.3.5. CCS Dynamic Bisimulation 10.7.3

$$p\ \Theta(R)\ q \overset{\text{def}}{=} \begin{cases} p \overset{\tau}{\to} p' & \text{implies} \quad \exists q'.\ q \xrightarrow{\tau} \overset{\tau}{\Longrightarrow} q' \quad \text{and} \quad p'\ R\ q' \\ p \overset{\lambda}{\to} p' & \text{implies} \quad \exists q'.\ q \overset{\lambda}{\Longrightarrow} q' \quad \text{and} \quad p'\ R\ q' \\ \text{(and vice versa)} & \end{cases}$$

A.4.3.6. CCS Axioms for Dynamic Bisimilarity 10.7.3

$$\begin{aligned}
&p + \tau p = \tau p\\
&\mu(p + \tau q) = \mu(p + \tau q) + \mu q
\end{aligned}$$

A.5. Temporal and Modal Logic

A.5.1. Hennessy - Milner Logic 10.5

The syntax of Hennessy-Milner logic formulas is:

$$F ::= true \quad | \quad false \quad | \quad \wedge_{i \in I} F_i \quad | \quad \vee_{i \in I} F_i \quad | \quad \Diamond_\mu F \quad | \quad \Box_\mu F$$

The *satisfaction relation* $\models \subseteq P \times \mathscr{L}$ is defined as follows:

$$\begin{array}{lll}
p \models true & & \\
p \models \wedge_{i \in I} F_i & \text{iff} & \forall i \in I.\ p \models F_i \\
p \models \vee_{i \in I} F_i & \text{iff} & \exists i \in I.\ p \models F_i \qquad \forall i \in I \\
p \models \Diamond_\mu F & \text{iff} & \exists p'.\ p \overset{\mu}{\to} p' \wedge p' \models F \\
p \models \Box_\mu F & \text{iff} & \forall p'.\ p \overset{\mu}{\to} p' \Rightarrow p' \models F
\end{array}$$

A.5.2. Linear Temporal Logic 11.1.1

The syntax of LTL is as follows:

$$\phi ::= true \quad | \quad false \quad | \quad p \quad | \quad \neg\phi \quad | \quad \phi_1 \wedge \phi_2 \quad | \quad \phi_1 \vee \phi_2 \quad | \quad O\phi \quad | \quad F\phi \quad | \quad G\phi \quad | \quad \phi_1 U \phi_2$$

Let P be a set of atomic propositions and $S : P \to 2^\omega$. We define the satisfaction operator $\models$ as follows:

$$\begin{array}{lll}
S & \models & true \\
S & \models & p \quad \text{if } 0 \in S(p) \\
S & \models & \neg\phi \quad \text{if it is not true that } S \models \phi \\
S & \models & \phi_1 \wedge \phi_2 \quad \text{if } S \models \phi_1 \text{ and } S \models \phi_2 \\
S & \models & \phi_1 \vee \phi_2 \quad \text{if } S \models \phi_1 \text{ or } S \models \phi_2 \\
S & \models & O\phi \quad \text{if } S^1 \models \phi \\
S & \models & F\phi \quad \text{if } \exists i \in \mathbb{N} \text{ such that } S^i \models \phi \\
S & \models & G\phi \quad \text{if } \forall i \in \mathbb{N} \text{ it holds } S^i \models \phi \\
S & \models & \phi_1 U \phi_2 \quad \text{if } \exists i \in \mathbb{N} \text{ such that } S^i \models \phi_2 \text{ and } \forall j < i.\ S^j \models \phi_1
\end{array}$$

A.5.3. Computation Tree Logic 11.1.2

The syntax of CTL is as follows:

$$\phi ::= true \mid false \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid O\phi \mid F\phi \mid G\phi \mid \phi_1 U \phi_2 \mid E\phi \mid A\phi$$

Let (T, S, P) be a branching structure and $\pi = v_0, v_1, \ldots, v_n, \ldots$ be an infinite path. We define the $\models$ relation for as follows:

state operators:

$$\begin{array}{lcl}
S,\pi & \models & true \\
S,\pi & \models & p \quad \text{if } v_0 \in S(p) \\
S,\pi & \models & \neg\phi \quad \text{if it is not true that } S,\pi \models \phi \\
S,\pi & \models & \phi_1 \wedge \phi_2 \quad \text{if } S,\pi \models \phi_1 \text{ and } S,\pi \models \phi_2 \\
S,\pi & \models & \phi_1 \vee \phi_2 \quad \text{if } S,\pi \models \phi_1 \text{ or } S,\pi \models \phi_2 \\
S,\pi & \models & O\phi \quad \text{if } S,\pi^1 \models \phi \\
S,\pi & \models & F\phi \quad \text{if } \exists i \in \omega \text{ such that } S,\pi^i \models \phi \\
S,\pi & \models & G\phi \quad \text{if } \forall i \in \omega \text{ it holds } S,\pi^i \models \phi \\
S,\pi & \models & \phi_1 U \phi_2 \quad \text{if } \exists i \in \omega \text{ such that } S,\pi^i \models \phi_2 \text{ and for all } j < i.\ S,\pi^j \models \phi_1
\end{array}$$

path operators

$$\begin{array}{lcl}
S,\pi & \models & E\phi \quad \text{if there exists } \pi_1 = v_0, v'_1, \ldots, v'_n, \ldots \text{ such that } S,\pi_1 \models \phi \\
S,\pi & \models & A\phi \quad \text{if for all paths } \pi_1 = v_0, v'_1, \ldots, v'_n, \ldots \text{ we have } S,\pi_1 \models \phi
\end{array}$$

This semantics apply both for CTL and CTL^*. The formulas of CTL are obtained by restricting CTL^*: a CTL^* formula is a CTL formula if the followings hold:

- A and E appear only immediately before a linear operator (i.e., F,G,U and O).
- each linear operator appears immediately after a quantifier (i.e., A and E).

A.6. μ-Calculus 11.2

The syntax of μ-calculus is as follows:

$$\phi ::= true \mid false \mid x \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \Diamond\phi \mid \Box\phi \mid \mu x.\phi \mid \nu x.\phi$$

We define the denotational semantics of μ-calculus which maps each predicate to the subset of states in which it holds as follows:

$$\begin{array}{rcl}
[\![true]\!]\rho & = & V \\
[\![false]\!]\rho & = & \emptyset \\
[\![x]\!]\rho & = & \rho x \\
[\![p]\!]\rho & = & \rho p \\
[\![\neg\phi]\!]\rho & = & V \setminus [\![\phi]\!]\rho \\
[\![\phi_1 \wedge \phi_2]\!]\rho & = & [\![\phi_1]\!]\rho \cap [\![\phi_2]\!]\rho \\
[\![\phi_1 \vee \phi_2]\!]\rho & = & [\![\phi_1]\!]\rho \cup [\![\phi_2]\!]\rho \\
[\![\Diamond\phi]\!]\rho & = & \{\, v \mid \exists v'.v \to v' \ \wedge\ v' \in [\![\phi]\!]\rho \,\} \\
[\![\Box\phi]\!]\rho & = & \{\, v \mid \forall v'.v \to v' \Rightarrow v' \in [\![\phi]\!]\rho \,\} \\
[\![\mu x.\phi]\!]\rho & = & \text{fix}\, \lambda S.\, [\![\phi]\!]\rho[^S/_x] \\
[\![\nu x.\phi]\!]\rho & = & \text{Fix}\, \lambda S.\, [\![\phi]\!]\rho[^S/_x]
\end{array}$$

A.7. π-calculus 12

A.7.1. π-calculys Syntax 12.1

$$\begin{array}{rcl} p & ::= & \mathbf{nil} \mid \alpha.p \mid [x = y]p \mid p + p \mid p|p \mid (y)p \mid !p \\ \alpha & ::= & \tau \mid x(y) \mid \overline{x}y \end{array}$$

A.7.2. π-calculus Operational Semantics 12.2

$$\text{(Tau)}\ \frac{}{\tau.p \xrightarrow{\tau} p} \qquad \text{(Out)}\ \frac{}{\overline{x}\,y.p \xrightarrow{\overline{x}\,y} p} \qquad \text{(In)}\ \frac{}{x(y).p \xrightarrow{x(w)} p\{^w/_y\}}\ w \notin fn((y)p)$$

$$\text{(SumL)}\ \frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} \qquad \text{(SumR)}\ \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'} \qquad \text{(Match)}\ \frac{p \xrightarrow{\alpha} p'}{[x = x]p \xrightarrow{\alpha} p'}$$

$$\text{(ParL)}\ \frac{p \xrightarrow{\alpha} p'}{p|q \xrightarrow{\alpha} p'|q}\ bn(\alpha) \cap fn(q) = \emptyset \qquad \text{(ParR)}\ \frac{q \xrightarrow{\alpha} q'}{p|q \xrightarrow{\alpha} p|q'}\ bn(\alpha) \cap fn(p) = \emptyset$$

$$\text{(ComL)}\ \frac{p \xrightarrow{\overline{x}\,z} p' \quad q \xrightarrow{x(y)} q'}{p|q \xrightarrow{\tau} p'|(q'\{^z/_y\})} \qquad \text{(ComR)}\ \frac{p \xrightarrow{x(y)} p' \quad q \xrightarrow{\overline{x}\,z} q'}{p|q \xrightarrow{\tau} p'\{^z/_y\}|q'}$$

$$\text{(Res)}\ \frac{p \xrightarrow{\alpha} p'}{(y)p \xrightarrow{\alpha} (y)p'}\ y \notin n(\alpha) \qquad \text{(Open)}\ \frac{p \xrightarrow{\overline{x}\,y} p'}{(y)p \xrightarrow{\overline{x}(w)} p'\{^w/_y\}}\ y \neq x \quad w \notin fn((y)p)$$

$$\text{(CloseL)}\ \frac{p \xrightarrow{\overline{x}\,(w)} p' \quad q \xrightarrow{x(w)} q'}{p|q \xrightarrow{\tau} (w)(p'|q')} \qquad \text{(CloseR)}\ \frac{p \xrightarrow{x(w)} p' \quad q \xrightarrow{\overline{x}\,(w)} q'}{p|q \xrightarrow{\tau} (w)(p'|q')} \qquad \text{(Rep)}\ \frac{p|!p \xrightarrow{\alpha} p'}{!p \xrightarrow{\alpha} p'}$$

A.7.3. π-calculus Abstract Semantics 12.4

A.7.3.1. Strong Early Ground Bisimulation 12.4.1

$$p\ S\ q \Rightarrow \begin{cases} \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y)\ \wedge\ bn(\alpha) \notin fn(q), \ \text{ then } \exists q'.\ q \xrightarrow{\alpha} q' \text{ and } p'\ S\ q' \\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q), \ \text{then } \forall w.\ \exists q'.\ q \xrightarrow{x(y)} q' \text{ and } p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

We define the strong early ground bisimilarity as follows:

$$p \overset{\circ}{\sim}_E q \Leftrightarrow p\ S\ q \text{ for some strong early ground bisimulation } S$$

A.7.3.2. Strong Early Full Bisimilarity 12.4.3

$$p \sim_E q \Leftrightarrow p\sigma \overset{\circ}{\sim}_E q\sigma \ \text{ for every substitution } \sigma$$

A.7.3.3. Strong Late Ground Bisimulation 12.4.2

$$p\ S\ q \Rightarrow \begin{cases} \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y)\ \wedge\ bn(\alpha) \notin fn(q), & \text{then } \exists q'.\ q \xrightarrow{\alpha} q' \text{ and } p'\ S\ q' \\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q), & \text{then } \exists q'.\ \forall w.\ q \xrightarrow{x(y)} q' \text{ and } p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} & \end{cases}$$

We define the strong late ground bisimilarity as follows:

$$p \overset{\circ}{\sim}_L q \Leftrightarrow p\ S\ q \text{ for some late early ground bisimulation } S$$

A.7.3.4. Strong Late Full Bisimilarity 12.4.3

$$p \sim_L q \Leftrightarrow p\sigma \overset{\circ}{\sim}_L q\sigma \ \text{ for every substitution } \sigma$$

A.7.3.5. Weak Early Ground Bisimulation 12.4.4

$$p\ S\ q \Rightarrow \begin{cases} \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y)\ \wedge\ bn(\alpha) \notin fn(q), & \text{then } \exists q'.\ q \overset{\alpha}{\Longrightarrow} q' \text{ and } p'\ S\ q' \\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q), & \text{then } \forall w.\ \exists q'.\ q \overset{x(y)}{\Longrightarrow} q' \text{ and } p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} & \end{cases}$$

where here α could be τ. We define the corresponding bisimilarity as follows:

$$p \overset{\bullet}{\approx}_E q \text{ iff } p\ S\ q \text{ for some weak early ground bisimulation } S.$$

A.7.3.6. Weak Late Ground Bisimulation 12.4.4

$$p\ S\ q \Rightarrow \begin{cases} \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y)\ \wedge\ bn(\alpha) \notin fn(q), & \text{then } \exists q'.\ q \overset{\alpha}{\Longrightarrow} q' \text{ and } p'\ S\ q' \\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q), & \text{then } \exists q'.\ q \overset{x(y)}{\Longrightarrow} q' \text{ and } \forall w.\ p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} & \end{cases}$$

We define the corresponding bisimilarity as follow:

$$p \overset{\bullet}{\approx}_L q \text{ iff } p\ S\ q \text{ for some weak late ground bisimulation } S.$$

A.8. LTL for Action, Non-determinism and Probability

Let S be a set of states, T be a set of transitions, L be a set of labels, $D(S)$ and $D(L \times S)$ be respectively the set of discrete probabilistic distributions over S and over $L \times S$.

- CCS: $\alpha : S \longrightarrow \mathscr{P}(L \times S)$
- DTMC: $\alpha : S \longrightarrow (D(S) + 1)$
- CTMC: $\alpha : S \rightarrow S \rightarrow \mathbb{R}$
- Reactive probabilistic transition systems: $\alpha : S \rightarrow L \rightarrow (D(S) + 1)$
- Generative probabilistic transition systems: $\alpha : S \rightarrow (D(L \times S) + 1)$
- Segala Automata: $\alpha : S \rightarrow \mathscr{P}(D(L \times S))$
- Simple Segala Automata: $\alpha : S \rightarrow \mathscr{P}(L \times D(S))$

A.9. Larsen-Skou Logic 14.1.1.1

We define the Larsen-Skou satisfaction relation as follows:

$$\begin{array}{lcl} s \models \mathbf{true} & & \\ s \models \varphi_1 \wedge \varphi_2 & \Leftrightarrow & s \models \varphi_1 \text{ and } s \models \varphi_2 \\ s \models \neg\varphi & \Leftrightarrow & \neg s \models \varphi \\ s \models \langle l \rangle_q \varphi & \Leftrightarrow & \alpha\, s\, l\, [\![\varphi]\!] \geq q \text{ where } [\![\varphi]\!] = \{s \in S | s \models \varphi\} \end{array}$$

A.10. PEPA 15

A.10.1. PEPA Syntax 15.2.1

$$P ::= \mathbf{nil}\ \mid (\alpha, r).P \mid P + P \mid P \underset{L}{\bowtie} P \mid P/L \mid C$$

A.10.2. PEPA Operational Semantics 15.2.2

$$\frac{}{(\alpha, r).P \xrightarrow{(\alpha,r)} P} \qquad \frac{P \xrightarrow{(\alpha,r)} P'}{P + Q \xrightarrow{(\alpha,r)} P'} \qquad \frac{Q \xrightarrow{(\alpha,r)} Q'}{P + Q \xrightarrow{(\alpha,r)} Q'}$$

$$\frac{P \xrightarrow{(\alpha,r)} P' \quad \alpha \notin L}{P/L \xrightarrow{(\alpha,r)} P'/L} \qquad \frac{P \xrightarrow{(\alpha,r)} P' \quad \alpha \in L}{P/L \xrightarrow{(\tau,r)} P'/L}$$

$$\frac{P \xrightarrow{(\alpha,r)} P' \quad \alpha \notin L}{P \underset{L}{\bowtie} Q \xrightarrow{(\alpha,r)} P' \underset{L}{\bowtie} Q} \qquad \frac{Q \xrightarrow{(\alpha,r)} Q' \quad \alpha \notin L}{P \underset{L}{\bowtie} Q \xrightarrow{(\alpha,r)} P \underset{L}{\bowtie} Q'}$$

$$\frac{P \xrightarrow{(\alpha,r_1)} P' \quad Q \xrightarrow{(\alpha,r_2)} Q' \quad \alpha \in L}{P \underset{L}{\bowtie} Q \xrightarrow{(\alpha,r)} P' \underset{L}{\bowtie} Q'} \quad \text{where } r = \min(r_\alpha(P), r_\alpha(Q)) * \frac{r_1}{r_\alpha(P)} * \frac{r_2}{r_\alpha(Q)}$$

$$\frac{P \xrightarrow{(\alpha,r)} P' \quad (C \overset{\text{def}}{=} P) \in \Delta}{C \xrightarrow{(\alpha,r)} P'}$$

where we have denoted by $r_\alpha(P)$ the *apparent rate* of action α in P, which is defined by structural recursion as follows:

$$\begin{array}{rcl} r_\alpha(\mathbf{nil}) & = & 0 \\ r_\alpha((\beta, r).P) & = & \begin{cases} r & \text{if } \alpha = \beta \\ 0 & \text{if } \alpha \neq \beta \end{cases} \\ r_\alpha(P + Q) & = & r_\alpha(P) + r_\alpha(Q) \\ r_\alpha(P/L) & = & \begin{cases} r_\alpha(P) & \text{if } \alpha \notin L \\ 0 & \text{if } \alpha \in L \end{cases} \\ r_\alpha(P \underset{L}{\bowtie} Q) & = & \begin{cases} min(r_\alpha(P), r_\alpha(Q)) & \text{if } \alpha \in L \\ r_\alpha(P) + r_\alpha(Q) & \text{if } \alpha \notin L \end{cases} \end{array}$$