Roberto Bruni, Ugo Montanari

Models of Computation

– Monograph –

April 29, 2016

Springer

Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.

Alan Turing[1]

[1] The purpose of ordinal logics (from Systems of Logic Based on Ordinals), Proceedings of the London Mathematical Society, series 2, vol. 45, 1939.

Preface

The origins of this book lie their roots on more than 15 years of teaching a course on formal semantics to graduate Computer Science to students in Pisa, originally called *Fondamenti dell'Informatica: Semantica* (*Foundations of Computer Science: Semantics*) and covering models for imperative, functional and concurrent programming. It later evolved to *Tecniche di Specifica e Dimostrazione* (*Techniques for Specifications and Proofs*) and finally to the currently running *Models of Computation*, where additional material on probabilistic models is included.

The objective of this book, as well as of the above courses, is to present different *models of computation* and their basic *programming paradigms*, together with their mathematical descriptions, both *concrete* and *abstract*. Each model is accompanied by some relevant formal techniques for reasoning on it and for proving some properties.

To this aim, we follow a rigorous approach to the definition of the *syntax*, the *typing* discipline and the *semantics* of the paradigms we present, i.e., the way in which well-formed programs are written, ill-typed programs are discarded and the way in which the meaning of well-typed programs is unambiguously defined, respectively. In doing so, we focus on basic proof techniques and do not address more advanced topics in detail, for which classical references to the literature are given instead.

After the introductory material (Part I), where we fix some notation and present some basic concepts such as term signatures, proof systems with axioms and inference rules, Horn clauses, unification and goal-driven derivations, the book is divided in four main parts (Parts II-V), according to the different styles of the models we consider:

IMP: imperative models, where we apply various incarnations of well-founded induction and introduce λ-notation and concepts like structural recursion, program equivalence, compositionality, completeness and correctness, and also complete partial orders, continuous functions, fixpoint theory;

HOFL: higher-order functional models, where we study the role of type systems, the main concepts from domain theory and the distinction between lazy and eager evaluation;

CCS, π: concurrent, non-deterministic and interactive models, where, starting from operational semantics based on labelled transition systems, we introduce the notions of bisimulation equivalences and observational congruences, and overview some approaches to name mobility, and temporal and modal logics system specifications;

PEPA: probabilistic/stochastic models, where we exploit the theory of Markov chains and of probabilistic reactive and generative systems to address quantitative analysis of, possibly concurrent, systems.

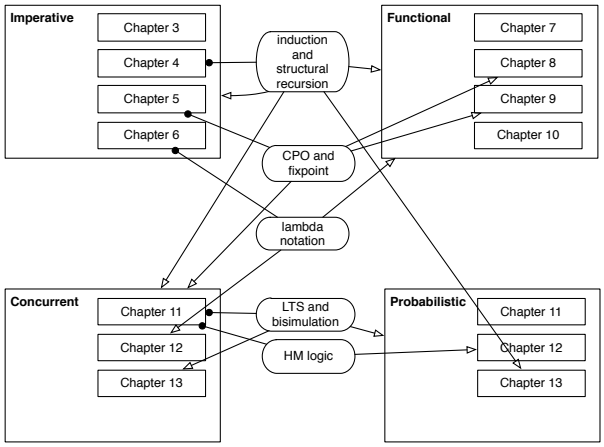
Each of the above models can be studied in separation from the others, but previous parts introduce a body of notions and techniques that are also applied and extended in later parts.

Parts I and II cover the essential, classic topics of a course on formal semantics.

Part III introduces some basic material on process algebraic models and temporal and modal logic for the specification and verification of concurrent and mobile systems. CCS is presented in good detail, while the theory of temporal and modal logic, as well as π-calculus, are just overviewed. The material in Part III can be used in conjunction with other textbooks, e.g., on model checking or π-calculus, in the context of a more advanced course on the formal modelling of distributed systems.

Part IV outlines the modelling of probabilistic and stochastic systems and their quantitative analysis with tools like PEPA. It poses the basis for a more advanced course on quantitative analysis of sequential and interleaving systems.

The diagram that highlights the main dependencies is represented below:



The diagram contains a squared box for each chapter / part and a rounded-corner box for each subject: a line with a filled-circle end joins a subject to the chapter where it is introduced, while a line with an arrow end links a subject to a chapter or part where it is used. In short:

Induction and recursion: various principles of induction and the concept of structural recursion are introduced in Chapter 4 and used extensively in all subsequent chapters.

CPO and fixpoint: the notion of complete partial order and fixpoint computation are first presented in Chapter 5. They provide the basis for defining the denotational semantics of IMP and HOFL. In the case of HOFL, a general theory of product and functional domains is also introduced (Chapter 8). The notion of fixpoint is also used to define a particular form of equivalence for concurrent and probabilistic systems, called bisimilarity, and to define the semantics of modal logic formulas.

Lambda-notation: λ-notation is a useful syntax for managing anonymous functions. It is introduced in Chapter 6 and used extensively in Part III.

LTS and bisimulation: Labelled transition systems are introduced in Chapter 11 to define the operational semantics of CCS in terms of the interactions performed. They are then extended to deal with name mobility in Chapter 13 and with probabilities in Part V. A bisimulation is a relation over the states of an LTS that is closed under the execution of transitions. The before mentioned bisimilarity is the coarsest bisimulation relation. Various forms of bisimulation are studied in Part IV and V.

HM-logic: Hennessy-Milner logic is the logic counterpart of bisimilarity: two state are bisimilar if and only if they satisfy the same set of HM-logic formulas. In the context of probabilistic system, the approach is extended to Larsen-Skou logic in Chapter 15.

Each chapter of the book is concluded by a list of exercises that span over the main techniques introduced in that chapter. Solutions to selected exercises are collected at the end of the book.

Pisa, *Roberto Bruni*
February 2016 *Ugo Montanari*

Acknowledgements

We want to thank our friend and colleague Pierpaolo Degano for encouraging us to prepare this book and submit it to the EATCS monograph series. We thank Ronan Nugent and all the people at Springer for their editorial work. We acknowledge all the students of the course on *Models of Computation (MOD)* in Pisa for helping us to refine the presentation of the material in the book and to eliminate many typos and shortcomings from preliminary versions of this text. Last but not least, we thank Lorenzo Galeotti, Andrea Cimino, Lorenzo Muti, Gianmarco Saba, Marco Stronati, former students of the course on *Models of Computation*, who helped us with the LaTeX preparation of preliminary versions of this book, in the form of lecture notes.

Contents

Acronyms

$\sim$	operational equivalence in IMP (see Definition 3.3)
$\equiv_{den}$	denotational equivalence in HOFL (see Definition 10.4)
$\equiv_{op}$	operational equivalence in HOFL (see Definition 10.3)
$\simeq$	CCS strong bisimilarity (see Definition 11.5)
$\approx$	CCS weak bisimilarity (see Definition 11.16)
$\cong$	CCS weak observational congruence (see Section 11.8.2)
$\overset{\bullet}{\cong}$	CCS dynamic bisimilarity (see Definition 11.17)
$\overset{\circ}{\sim}_E$	π-calculus early bisimilarity (see Definition 13.3)
$\overset{\circ}{\sim}_L$	π-calculus late bisimilarity (see Definition 13.4)
$\sim_E$	π-calculus strong early full bisimilarity (see Section 13.5.3)
$\sim_L$	π-calculus strong late full bisimilarity (see Section 13.5.3)
$\overset{\bullet}{\approx}_E$	π-calculus weak early bisimilarity (see Section 13.5.4)
$\overset{\bullet}{\approx}_L$	π-calculus weak late bisimilarity (see Section 13.5.4)
$\mathscr{A}$	interpretation function for the denotational semantics of IMP arithmetic expressions (see Section 6.2.1)
ack	Ackermann function (see Example 4.18)
Aexp	set of IMP arithmetic expressions (see Chapter 3)
$\mathscr{B}$	interpretation function for the denotational semantics of IMP boolean expressions (see Section 6.2.2)
Bexp	set of IMP boolean expressions (see Chapter 3)
$\mathbb{B}$	set of booleans
$\mathscr{C}$	interpretation function for the denotational semantics of IMP commands (see Section 6.2.3)
CCS	Calculus of Communicating Systems (see Chapter 11)
Com	set of IMP commands (see Chapter 3)
CPO	Complete Partial Order (see Definition 5.11)
$\text{CPO}_\perp$	Complete Partial Order with bottom (see Definition 5.12)
CSP	Communicating Sequential Processes (see Section 16.2)
CTL	Computation Tree Logic (see Section 12.1.2)
CTMC	Continuous Time Markov Chain (see Definition 14.15)

DTMC	Discrete Time Markov Chain (see Definition 14.14)
Env	set of HOFL environments (see Chapter 9)
fix	(least) fixpoint (see Definition 5.2.2)
FIX	(greatest) fixpoint
gcd	greatest common divisor
HML	Hennessy-Milner modal Logic (see Section 11.6)
HM-Logic	Hennessy-Milner modal Logic (see Section 11.6)
HOFL	A Higher-Order Functional Language (see Chapter 7)
IMP	A simple IMPerative language (see Chapter 3)
int	integer type in HOFL (see Definition 7.2)
Loc	set of locations (see Chapter 3)
LTL	Linear Temporal Logic (see Section 12.1.1)
LTS	Labelled Transition System (see Definition 11.2)
lub	least upper bound (see Definition 5.7)
$\mathbb{N}$	set of natural numbers
$\mathscr{P}$	set of closed CCS processes (see Definition 11.1)
PEPA	Performance Evaluation Process Algebra (see Chapter 16)
Pf	set of partial functions on natural numbers (see Example 5.13)
PI	set of partial injective functions on natural numbers (see Problem 5.12)
PO	Partial Order (see Definition 5.1)
PTS	Probabilistic Transition System (see Section 14.3.2)
$\mathbb{R}$	set of real numbers
$\mathscr{T}$	set of HOFL types (see Definition 7.2)
Tf	set of total functions from $\mathbb{N}$ to $\mathbb{N}_\perp$ (see Example 5.14)
Var	set of HOFL variables (see Chapter 7)
$\mathbb{Z}$	set of integers

Part I
Preliminaries

This part introduces the basic terminology, notation and mathematical tools used in the rest of the book.

Chapter 1
Introduction

It is not necessary for the semantics to determine an implementation, but it should provide criteria for showing that an implementation is correct. (Dana Scott)

Abstract This chapter overviews the motivation, guiding principles and main concepts used in the book. It starts by explaining the role of formal semantics and different approaches to its definition, then briefly exposes some important subjects covered in the book, like domain theory and induction principles and it is concluded by an explanation of the content of each chapter, together with a list of references to the literature for studying some topics in more depth or for using some companion textbooks in conjunction with the current text.

1.1 Structure and Meaning

Any programming language is fully defined in terms of three essential features:

Syntax: refers to the appearance of the programs of the language;
Types: restrict the syntax to enforce suitable properties on programs;
Semantics: refers to the meanings of (well-typed) programs.

Example 1.1. The alphabet of roman numerals, the numeric system used in ancient Rome, consists of seven letters drawn from the Latin alphabet. A value is assigned to each letter (see Table 1.1) and a number n is expressed by juxtaposing some letters whose values sum to n. Not all sequences are valid though. Symbols are usually placed from left to right, starting with the largest value and ending with the smallest value. However, to avoid four repetitions in a row of the same letter, like IIII, subtractive notation is used: when a symbol with smaller value u is placed to the left of a symbol with higher value v they represent the number $v - u$. So the symbol I can be placed before V or X; the symbol X before L or C and the symbol C before D or M, and 4 is written IV instead of IIII. While IX and XI are both correct sequences, with values 9 and 11, respectively, the sequence IXI is not correct and has no corresponding value. The rules that prescribe the correct sequences of symbols define the (well-typed) syntax of roman numerals. The rules that define how to evaluate roman numerals to positive natural numbers give their semantics.

Table 1.1: Alphabet of roman numerals

Symbol	I	V	X	L	C	D	M
Value	1	5	10	50	100	500	1000

1.1.1 Syntax, Types and Pragmatics

The syntax of a *formal* language tells us which sequences of symbols are valid statements and which ones make no sense and should be discarded.

Mathematical tools such as regular expressions, context-free grammars, and Backus-Naur Form (BNF) are now widely applied tools for defining the syntax of formal languages. They are studied in every computer science degree and are exploited in technical appendices of many programming language manuals to define the grammatical structure of programs without ambiguities.

Types can be used to limit the occurrence of errors or to allow compiler optimisations or to reduce the risk of introducing bugs or just to discourage certain programming malpractices. Types are often presented as set of logic rules, called *type systems* that are used to assign a type unambiguously to each program and computed value. Different type systems can be defined over the same language to enforce different properties.

However, grammars and types do not explain what a correctly written program means. Thus, every language manual also contains natural language descriptions of the meaning of the various constructs, how they should be used and styled, and example code fragments. This attitude falls under the *pragmatics* of a language, describing, e.g., how the various features should be used, which auxiliary tools are available (syntax checkers, debuggers, etc.). Unfortunately this leaves space to different interpretations that can ultimately lead to discordant implementations of the same language or to compilers that rely on questionable code optimisation strategies.

If an official formal semantics of a language would be available as well, it could accompany the language manual too and solve any ambiguity for implementors and programmers. This is not yet the case because effective techniques for specifying the run-time behaviour of programs in a rigorous manner have proved much harder to develop than grammars.

1.1.2 Semantics

The origin of the word ‘semantics’ can be traced back to a book by French philologist Michel Bréal (1832–1915), published in 1900, where it referred to *the study of how words change their meanings*. Subsequently, the word ‘semantics’ has also changed its meaning, and it is now generally defined as *the study of the meanings of words and phrases in a language*.

In Computer Science, the semantics is concerned with *the study of the meaning of (well-typed) programs*.

The studies in formal semantics are not always easily accessible to a student of computer science or engineering, without a good background in mathematical logic and, as a consequence, they are often regarded as an exoteric subject by people not familiar enough with the mathematical tools involved.

Following [36] we can ask ourselves: *what do we gain by formalising the semantics of a programming language?*

After all, programmers can write programs that are trusted to "work as expected", once they have been thoroughly tested, so how would the effort spent in a rigorous formalisation of the semantics pay back? An easy answer is that today, in the era of the Internet of Things and cyberphysical systems, our lives, the machines and devices we use, and the entire world run on software. It is not enough to require that medical implants, personal devices, planes and nuclear reactors seem "to work as expected"!

To give a more circumstantiated answer, we can start from the related question: *What was gained when language syntax was formalised?*

It is generally understood that the formalisation of syntax leads, e.g., to the following benefits:

1. it standardises the language; this is crucial
   - to programmers, as a guide to write syntactically correct programs, and
   - to implementors, as a reference to develop a correct parser.
2. it permits a formal analysis of many properties, like finding and resolving parsing ambiguities.
3. it can be used as input to a compiler front-end generating tool, such as Yacc, Bison, Xtext. In this way, from the syntax definition one can automatically derive an implementation of the compiler's front-end.

Providing a formal semantics definition of a programming language can then lead to similar benefits:

1. it standardises a machine-independent specification of the language; this is crucial:
   - to programmers, for improving the programs they write, and
   - to implementors, to design a correct and efficient code generator.
2. it permits a formal analysis of program properties, like type safety, termination, specification compliance and program equivalence.
3. it can be used as input to a compiler back-end generating tool. In this way, the semantics definition gives also the (prototypal and possibly inefficient) implementation of the back end of the language's compiler. Moreover, efficient compilers need to adhere to the semantics and their optimisations need correctness proofs.

What is then the semantics of a programming language?

A crude view is that the semantics of a programming language is defined by (the back-end of) its compiler or interpreter: from the source program to the target code executed by the computer. This view is clearly not acceptable because, e.g., it refers

to specific pieces of commercial hardware and software, the specification is not good for portability, it is not at the right level of abstraction to be understood by a programmer, it is not at the right level of abstraction to state and prove interesting properties of programs (for example, two programs written for the same purpose by different programmers are likely different, even if they should have the same meaning). Finally, if different implementations are given, how do we know that they are correct and compatible?

Example 1.2. We can hardly claim to know that two programs mean the same if we cannot tell what a program means. For example, consider the Java expressions:

```
x + (y + z)                    (x + y) + z
```

Are they equivalent? Can we replace the former with the latter (and viceversa) in a program, without changing its meaning? Under which circumstances?[1]

To give a semantics for a programming language means to define the behaviour of any program written in this language. As there are infinitely many programs, one would like to have a finitary description of the semantics that can take into account any of them. Only when the semantics is given one can prove such important properties as program equivalence, or program correctness.

1.1.3 Mathematical Models of Computation

In giving a formal semantics to a programming language we are concerned with *building a mathematical model*: Its purpose is to serve as a basis for understanding and reasoning about how programs behave. Not only is a mathematical model useful for various kinds of analysis and verification, but also, at a more fundamental level, because the activity of trying to define precisely the meanings of program constructions can reveal all kinds of subtleties which it is important to be aware of.

Unlike the acceptance of BNF as a standard definition method for syntax, there is little hope that a single definition method will take hold for semantics. This is because semantics

- is harder to formalise than syntax,
- has a wider variety of applications,
- is dependent on the properties we want to tackle, i.e., different models are best suited for tackling different issues.

In fact, different semantic styles and models have been developed for different purposes. The overall aim of the book is to study the main semantic styles, compare their expressiveness, and apply them to study program properties. To this aim it is fundamental to gain acquaintance with the principles and theories on which such semantic models are based.

[1] Remind that the '+' is overloaded in Java: it sums integers, floating points and concatenates strings.

Classically, semantics definition methods fall roughly into three groups: Operational, denotational and axiomatic. In this book we will focus mainly on the first two kinds of semantics, which find wider applicability.

Operational Semantics

In the operational semantics it is of interest *how* the effect of a computation is achieved. Some kind of abstract machine[2] is first defined, then the operational semantics describes the meaning of a program in terms of the steps/actions that this machine executes. The focus of operational semantics is thus on states and state transformations.

An early notable example of operational semantics was concerned with the semantics of LISP (LISt Processor) by John McCarthy (1927–2011) [19]. A later example was the definition of the semantics of Algol 68 over a hypothetical computer [42].

In 1981, Gordon Plotkin (1946–) introduced the structural operational semantics style (SOS-style) in the technical report [28] which is still one of the most cited technical reports in computer science, only recently revised and re-issued in a journal [30, 31].

Gilles Kahn (1946-2006) introduced another form of operational semantics, called natural semantics, or big-step semantics, in 1987, where possibly many steps of execution are incorporated into a single logical derivation [16].

It is relatively easy to write the operational semantics in the form of Horn clauses, a particular form of logical implications. In this way, they can be interpreted by a logic programming system, such as Prolog.[3]

Because of the strong connection with the syntactic structure and the fact that the mathematics involved is usually less complicated than in other semantic approaches, SOS-style operational semantics can provide the programmers with a concise and accurate description of what the language constructs do. In fact, it is syntax-oriented, inductive and easy to grasp. Operational semantics is also versatile: it applies with minor variations to most different computing paradigms.

Denotational Semantics

In denotational semantics, the meaning of a well-formed program is some mathematical object (e.g., a function from input data to output data). The steps taken to calculate

[2] The term *machine* ordinarily refers to a *physical* device that performs mechanical functions. The term *abstract* distinguishes a physically existent device from one that exists in the imagination of its inventor or user: it is a convenient conceptual abstraction that leaves out many implementation details. The archetypical abstract machine is the Turing machine.

[3] However, we have to leave aside issues about performance and the fact that Prolog is not complete, because it exploits a depth-first exploration strategy for the next step to execute: backtracking out of wrong attempted steps is only possible if they are finitely many.

the output and the abstract machine where they are executed are unimportant: Only the *effect* is of interest, not how it is obtained.

The essence of denotational semantics lies in the principle of *compositionality*: the semantics takes the form of a function that assigns an element of some mathematical domain to each individual construct, in such a way that *the meaning of a composite construct does not depend on the particular form of the constituent constructs, but only on their meanings.*

Denotational semantics originated in the pioneering work of Christopher Strachey (1916–1975) and Dana Scott (1932–) in the late 1960s and in fact it is sometimes called Scott-Strachey semantics [38, 37, 39].

Denotational semantics descriptions can also be used to derive implementations. Still there is a problem with performance: operations that can be efficiently performed on computer hardware, such as reading or changing the contents of storage cells, are first mapped to relatively complicated mathematical notions which must then be mapped back again to a concrete computer architecture.

One limitation is that in the case of concurrent, interactive, non-deterministic systems the body of mathematics involved in the definition of denotational semantics is quite heavy.

Axiomatic Semantics

Instead of directly assigning a meaning to each program, axiomatic semantics gives a description of the constructs in a programming language by providing logical conditions that are satisfied by these constructs. Axiomatic semantics poses the focus on valid *assertions* for proving program correctness: there may be aspects of the computation and of the effect that are deliberately ignored.

The axiomatic semantics has been put forward by the work of Robert W.Floyd (1936–2001) on flowchart languages [8] and of Tony Hoare (1934–) on structured imperative programs [15]. In fact it is sometimes referred to as Floyd-Hoare logic. The basic idea is that program statements are described by two logical assertions: a pre-condition, prescribing the state of the system before executing the program, and a post-condition, satisfied by the state after the execution, when the preconditions are valid. Using such an axiomatic description it is possible, at least in principle, to prove the correctness of a program with respect to a specification. Two main forms of correctness are considered:

Partial: a program is partially correct w.r.t. a pre-condition and a post-condition if whenever the initial state fulfils the pre-condition *and* the program terminates, the final state is guaranteed to fulfil the post-condition. The partial correctness property does not ensure that the program will terminate, e.g., a program which never terminates satisfies every property.

Total: a program is totally correct w.r.t. a pre-condition and a post-condition if whenever the initial state fulfils the pre-condition, *then* the program terminates, and the final state is guaranteed to fulfil the post-condition.

The axiomatic method becomes cumbersome in the presence of modular program constructs, e.g., goto's and objects, but also as simple as blocks and procedures. Another limitation of axiomatic semantics is that it is scarcely applicable to the case of concurrent, interactive systems, whose correct behaviour often involves non-terminating computations (for which post-conditions cannot be used).

1.2 A Taste of Semantics Methods: Numerical Expressions

We can give a first, informal overview of the different flavours of semantics styles we will consider in this book by taking a simple example of numerical expressions.[4] Let us consider two syntactic categories *Nums* and *Exp*, respectively, for numerals $\mathtt{n} \in Nums$ and expressions $E \in Exp$, defined by the grammar:

$$\begin{array}{lcl} \mathtt{n} & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots \\ e & ::= & \mathtt{n} \mid e \oplus e \mid e \otimes e \end{array}$$

The above language of numerical expressions uses the auxiliary set of *numerals*, *Nums*, which are syntactic representations of the more abstract set of natural numbers.

Remark 1.1 (Numbers vs numerals). The natural numbers $0, 1, 2, \ldots$ are mathematical objects which exist in some abstract world of concepts. They find concrete representations in different languages. For example, the number 5 is represented by:

- the string "five" in English,
- the string "101" in binary notation,
- the string "V" in roman numerals.

To differentiate between numerals ($\mathtt{5}$) and numbers (5) we use here different fonts.

From the grammar it is evident that there are three ways to build expressions:

- any numeral $\mathtt{n}$ is also an expression;
- if we are given any two expressions e_0 and e_1, then $e_0 \oplus e_1$ is also an expression;
- if we are given any two expressions e_0 and e_1, then $e_0 \otimes e_1$ is also an expression.

In the book we will always use abstract syntax representations, as if all concrete terms were parsed before we start to work with them.

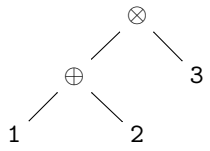
Remark 1.2 (Concrete and abstract syntax). While the *concrete* syntax of a language is concerned with the precise linear sequences of symbols which are valid terms of the language, we are interested in the *abstract* syntax, which describes expressions purely in terms of their structure. We will never be worried about where the brackets are in expressions like

$$\mathtt{1} \oplus \mathtt{2} \otimes \mathtt{3}$$

[4] The example has been inspired from some course notes on the "Semantics of programming languages", by Matthew Hennessy.

because we will never deal with such unparsed terms.

In other words we are considering only (valid) *abstract syntax trees*, like



Since it would be tedious to draw trees every time, we use linear syntax and brackets, like $(\mathtt{1} \oplus \mathtt{2}) \otimes \mathtt{3}$ to save space while avoiding ambiguities.

An Informal Semantics

Since in the expressions we deliberately used some non-standard symbols $\oplus$ and $\otimes$, we must define what is their meaning. Programmers primarily learn the semantics of a language through examples, their intuitions about the underlying computational model, and some natural language description. An informal description of the meaning of the expressions we are considering could be the following:

- a numeral $\mathtt{n}$ is evaluated to the corresponding natural number n;
- to find the value associated with an expression of the form $e_0 \oplus e_1$ we evaluate the expressions e_0 and e_1 and take the sum of the results;
- to find the value associated with an expression of the form $e_0 \otimes e_1$ we evaluate the expressions e_0 and e_1 and take the product of the results.

We hope the reader agrees that the above guidelines are sufficient to determine the value of any well-formed expression, no matter how large.[5]

To accompany the description with examples, we can add that:

- $\mathtt{2}$ is evaluated to 2
- $(\mathtt{1} \oplus \mathtt{2}) \otimes \mathtt{3}$ is evaluated to 9
- $(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4})$ is evaluated to 21

Since natural language is notoriously prone to mis-interpretations and misunderstandings, in the following we try to make the above description more accurate.

We show next how the operational semantics can formalise the steps needed to evaluate an expression over some abstract computational device and how the denotational semantics can assign meaning to numerical expressions (their valuation).

A Small-Step Operational Semantics

There are several versions of operational semantics for the above language of expressions. The first one we present is likely familiar to you: it simplifies expressions until a value is met. This is achieved by defining judgements of the form

[5] Note that we are not telling the order in which e_0 and e_1 must be evaluated: is it important?

$$\frac{}{\mathtt{n}_0 \oplus \mathtt{n}_1 \to \mathtt{n}}\ n = n_0 + n_1\ (sum) \qquad \frac{e_0 \to e_0'}{e_0 \oplus e_1 \to e_0' \oplus e_1}\ (sumL) \qquad \frac{e_1 \to e_1'}{e_0 \oplus e_1 \to e_0 \oplus e_1'}\ (sumR)$$

$$\frac{}{\mathtt{n}_1 \otimes \mathtt{n}_2 \to \mathtt{n}}\ n = n_1 \times n_2\ (prod) \qquad \frac{e_0 \to e_0'}{e_0 \otimes e_1 \to e_0' \otimes e_1}\ (prodL) \qquad \frac{e_1 \to e_1'}{e_0 \otimes e_1 \to e_0 \otimes e_1'}\ (prodR)$$

Fig. 1.1: Small-step semantics rules for numerical expressions

$$e_0 \to e_1$$

to be read as: *after performing one step of evaluation of e_0, the expression e_1 remains to be evaluated.*

Small-step semantics formally describes how individual steps of a computation take place on an abstract device, but it ignores details like the use of registers and storage addresses. This makes the description independent of machine architectures and implementation strategies.

The logic inference rules are written in the general form (see Section 2.2):

$$\frac{premises}{conclusion}\ side\text{-}condition \quad (rule\ name)$$

meaning that if the *premises* and the *side-condition* are met then the *conclusion* can be drawn, where the premises consist of one, none or more judgements and the side-condition is a single boolean predicate. The *rule name* is just a convenient label that can be used to refer the rule. Rules with no premises are called axioms and their conclusion is postulated to be always valid.

The rules for the expressions are given in Figure 1.1. For example, the rule *sum* says that $\oplus$ applied to two numerals evaluates to the numeral representing the sum of the two arguments, while the rule *sumL* (respectively, *sumR*) says that we are allowed to simplify the left (resp., right) argument. Analogously for product.

For example, we can derive both the judgements

$$(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to \mathtt{3} \otimes (\mathtt{3} \oplus \mathtt{4}) \qquad (\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to (\mathtt{1} \oplus \mathtt{2}) \otimes \mathtt{7}$$

as witnessed by the formal derivations

$$\frac{\dfrac{}{\mathtt{1} \oplus \mathtt{2} \to \mathtt{3}}\ 3 = 1 + 2\ (sum)}{(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to \mathtt{3} \otimes (\mathtt{3} \oplus \mathtt{4})}\ (prodL) \qquad \frac{\dfrac{}{\mathtt{3} \oplus \mathtt{4} \to \mathtt{7}}\ 7 = 3 + 4\ (sum)}{(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to (\mathtt{1} \oplus \mathtt{2}) \otimes \mathtt{7}}\ (prodR)$$

A derivation is represented as an (inverted) tree, with the goal to be verified at the root. The tree is generated by applications of the defining rules, with the terminating leaves being generated by axioms. As derivations tend to grow large, we will intro-

duce a convenient alternative notation for them in Chapter 2 (see Example 2.5 and Section 2.3) and will use it extensively in the subsequent chapters.

Note that even for a deterministic program, there can be many different computation sequences leading to the same final result, since the semantics may not specify a totally ordered sequence of evaluation steps.

If we want to enforce a specific evaluation strategy, then we can change the rules so to guarantee, e.g., that the left-most occurrence of an operator $\oplus$/$\otimes$ which has both its operands already evaluated is always executed first, while the evaluation of the second-operand is conducted only after the left-operand has been evaluated. We show only the two rules that need to be changed (changes are highlighted with boxes):

$$\frac{e_1 \to e_1'}{\boxed{\mathtt{n}_0} \oplus e_1 \to \boxed{\mathtt{n}_0} \oplus e_1'} \ (sumR) \qquad \frac{e_1 \to e_1'}{\boxed{\mathtt{n}_0} \otimes e_1 \to \boxed{\mathtt{n}_0} \otimes e_1'} \ (prodR)$$

Now the step judgement

$$(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to (\mathtt{1} \oplus \mathtt{2}) \otimes \mathtt{7}$$

is no longer derivable.

Instead, it is not difficult to derive the judgements:

$$(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to \mathtt{3} \otimes (\mathtt{3} \oplus \mathtt{4}) \qquad \mathtt{3} \otimes (\mathtt{3} \oplus \mathtt{4}) \to \mathtt{3} \otimes \mathtt{7} \qquad \mathtt{3} \otimes \mathtt{7} \to \mathtt{21}$$

The steps can be composed: let us write

$$e_0 \to^k e_k$$

if e_0 can be reduced to e_k in k-steps: that is there exists $e_1, e_2, \ldots, e_{k-1}$ such that we can derive the judgements

$$e_0 \to e_1 \qquad e_1 \to e_2 \qquad \ldots \qquad e_{k-1} \to e_k$$

This includes the case when $k = 0$: then e_k must be the same as e_0, i.e., in 0 steps any expression can reduce to itself.

In our example, by composing the above steps, we have

$$(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to^3 \mathtt{21}$$

We also write

$$e \not\to$$

when no expression e' can be found such that $e \to e'$.

It is immediate to see that for any numeral $\mathtt{n}$, we have $\mathtt{n} \not\to$, as no conclusion of the inference rules has a numeral as source of the transition.

To fully evaluate an expression, we need to indefinitely compute successive derivations until eventually a final numeral is obtained, that cannot be evaluated

$$\frac{}{\mathtt{n} \twoheadrightarrow \mathtt{n}}\ (num) \qquad \frac{e_0 \twoheadrightarrow \mathtt{n}_1 \quad e_1 \twoheadrightarrow \mathtt{n}_2}{e_0 \oplus e_1 \twoheadrightarrow \mathtt{n}}\ n = n_1 + n_2\ (sum) \qquad \frac{e_0 \twoheadrightarrow \mathtt{n}_1 \quad e_1 \twoheadrightarrow \mathtt{n}_2}{e_0 \otimes e_1 \twoheadrightarrow \mathtt{n}}\ n = n_1 \times n_2\ (prod)$$

Fig. 1.2: Natural semantics for numerical expressions

further. We write

$$e \to^* \mathtt{n}$$

to mean that there is some natural number k such that $e \to^k \mathtt{n}$, i.e., e can be evaluated to $\mathtt{n}$ in k steps. The relation $\to^*$ is called the *reflexive and transitive closure of* $\to$. Note that we have, e.g., $\mathtt{n} \to^* \mathtt{n}$ for any numeral $\mathtt{n}$.

In our example we can derive the judgement

$$(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to^* \mathtt{21}$$

Small-step operational semantics will be especially useful in Parts IV and V to assign different semantics to non-terminating systems.

A Big-Step Operational Semantics (or Natural Semantics)

Like small-step semantics, a natural semantics is a set of inference rules, but a *complete computation is done as a single, large derivation*. For this reason, a natural semantics is sometimes called a *big-step operational semantics*.

Big-step semantics formally describes how the overall results of the executions are obtained. It hides even more details than the small-step operational semantics. Like small-step operational semantics, natural semantics shows the context in which a computation step occurs, and like denotational semantics, natural semantics emphasises that the computation of a phrase is built from the computations of its sub-phrases.

Natural semantics have the advantage of often being simpler (needing fewer inference rules) and of often directly corresponding to an efficient implementation of an interpreter for the language. In our running example, we disregard the individual steps that led to the result and focus on the final outcome, i.e., we formalise the predicate $e \to^* \mathtt{n}$. Typically, the same predicate symbol $\to$ is used also in the case of natural semantics. To avoid ambiguities and to not overload the notation, here, for the sake of the running example, we use a different symbol. We define the predicate

$$e \twoheadrightarrow \mathtt{n}$$

to be read as: *the expression e is (eventually) evaluated to* $\mathtt{n}$.

The rules are reported in Figure 1.2. This time only three rules are needed, which immediately correspond to the informal semantics we gave for numerical expressions.

We can now verify that the judgement

$$(\mathsf{1} \oplus \mathsf{2}) \otimes (\mathsf{3} \oplus \mathsf{4}) \twoheadrightarrow \mathsf{21}$$

can be derived as follows:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{1} \twoheadrightarrow \mathsf{1}}\,(num) \quad \dfrac{}{\mathsf{2} \twoheadrightarrow \mathsf{2}}\,(num)}{\mathsf{1} \oplus \mathsf{2} \twoheadrightarrow \mathsf{3}}\; 3 = 1 + 2\;(sum) \quad \dfrac{\dfrac{}{\mathsf{3} \twoheadrightarrow \mathsf{3}}\,(num) \quad \dfrac{}{\mathsf{4} \twoheadrightarrow \mathsf{4}}\,(num)}{\mathsf{3} \oplus \mathsf{4} \twoheadrightarrow \mathsf{7}}\; 7 = 3 + 4\;(sum)}{(\mathsf{1} \oplus \mathsf{2}) \otimes (\mathsf{3} \oplus \mathsf{4}) \twoheadrightarrow \mathsf{21}}\; 21 = 3 \times 7\;(prod)$$

Small-step operational semantics gives more control of the details and order of evaluation. These properties make small-step semantics more convenient when proving type soundness of a type system against an operational semantics. Natural semantics can lead to simpler proofs, e.g., when proving the preservation of correctness under some program transformation. Natural semantics is also very useful to define reduction to canonical forms.

An interesting drawback of natural semantics is that semantics derivations can be drawn only for terminating programs. The main disadvantage of natural semantics is thus that non-terminating (diverging) computations do not have an inference tree.

We will exploit natural semantics mainly in Parts II and III of the book.

A Denotational Semantics

Differently from operational semantics, denotational semantics is concerned with manipulating mathematical objects and not with executing programs.

In the case of expressions, the intuition is that a term represents a number (expressed in form of a calculation). So we can choose as a *semantic domain* the set of natural numbers $\mathbb{N}$, and the *interpretation function* will then map expressions to natural numbers.

To avoid ambiguities between pieces of syntax and mathematical objects, we usually enclose syntactic terms within a special kind of brackets $[\![\cdot]\!]$ that serve as a separation. It is also common, when different interpretation functions are considered, to use calligraphic letters to distinguish the kind of terms they apply to (one for each syntax category).

In our running example, we define two semantics functions:

$$\mathscr{N}[\![\cdot]\!] : \mathit{Nums} \to \mathbb{N}$$
$$\mathscr{E}[\![\cdot]\!] : \mathit{Exp} \to \mathbb{N}$$

Remark 1.3. When we will study more complex languages, we will find that we need more complex (and less familiar) domains than $\mathbb{N}$. For example, as originally developed by Strachey and Scott, denotational semantics provides the meaning of a computer program as a function that maps input into output. To give denotations to recursively defined programs, Scott proposed working with continuous functions between domains, specifically complete partial orders.

Notice that our choice of semantic domain has certain immediate consequences for the semantics of our language: it implies that every expression will mean exactly one number! Without having defined yet the interpretation functions, and contrary to the operational semantics definitions, anyone looking at the semantics already knows that the language is:

deterministic: each expression has at most one answer;
normalising: every expression has an answer.

Giving a meaning to numerals is immediate

$$\mathscr{N}[\![\mathtt{n}]\!] = n$$

For composite expressions, the meaning will be determined by composing the meaning of the arguments

$$\begin{aligned}\mathscr{E}[\![\mathtt{n}]\!] &= \mathscr{N}[\![\mathtt{n}]\!]\\ \mathscr{E}[\![e_0 \oplus e_1]\!] &= \mathscr{E}[\![e_0]\!] + \mathscr{E}[\![e_1]\!]\\ \mathscr{E}[\![e_0 \otimes e_1]\!] &= \mathscr{E}[\![e_0]\!] \times \mathscr{E}[\![e_1]\!]\end{aligned}$$

We have thus defined the interpretation function *by induction on the structure of the expressions* and it is

compositional: the meaning of complex expressions is defined in terms of the meaning of the constituents.

As an example, we can interpret our running expression:

$$\begin{aligned}\mathscr{E}[\![(\mathtt{1}\oplus\mathtt{2})\otimes(\mathtt{3}\oplus\mathtt{4})]\!] &= \mathscr{E}[\![\mathtt{1}\oplus\mathtt{2}]\!] \times \mathscr{E}[\![\mathtt{3}\oplus\mathtt{4}]\!]\\ &= (\mathscr{E}[\![\mathtt{1}]\!] + \mathscr{E}[\![\mathtt{2}]\!]) \times (\mathscr{E}[\![\mathtt{3}]\!] + \mathscr{E}[\![\mathtt{4}]\!])\\ &= (\mathscr{N}[\![\mathtt{1}]\!] + \mathscr{N}[\![\mathtt{2}]\!]) \times (\mathscr{N}[\![\mathtt{3}]\!] + \mathscr{N}[\![\mathtt{4}]\!])\\ &= (1+2)\times(3+4) = 21\end{aligned}$$

Denotational semantics is best suited for sequential systems and thus exploited in Parts II and III.

Semantic Equivalence

We have now available three different semantics for numerical expressions:

$$e \to^* \mathtt{n} \qquad e \twoheadrightarrow \mathtt{n} \qquad \mathscr{E}[\![e]\!]$$

and we are faced with several questions:

1. Is it true that for every expression e there exists some numeral $\mathtt{n}$ such that $e \to^* \mathtt{n}$? The same property, often referred to as *normalisation* can be asked also for $e \twoheadrightarrow \mathtt{n}$, while it is trivially satisfied by $\mathscr{E}[\![e]\!]$.

2. Is it true that if $e \to^* \mathtt{n}$ and $e \to^* \mathtt{m}$ we have $\mathtt{n} = \mathtt{m}$?
   The same property, often referred to as *determinacy* can be asked also for $e \twoheadrightarrow \mathtt{n}$, while it is trivially satisfied by $\mathscr{E}[\![e]\!]$.
3. Is it true that $e \to^* \mathtt{n}$ iff $e \twoheadrightarrow \mathtt{n}$?
   This has to do with the *consistency* of the semantics and the question can be posed between any two of the three semantics we have defined.

We can also derive some intuitive relations of *equivalence* between expressions:

- Two expressions e_0 and e_1 are equivalent if for any numeral $\mathtt{n}$, $e_0 \to^* \mathtt{n}$ iff $e_1 \to^* \mathtt{n}$.
- Two expressions e_0 and e_1 are equivalent if for any numeral $\mathtt{n}$, $e_0 \twoheadrightarrow \mathtt{n}$ iff $e_1 \twoheadrightarrow \mathtt{n}$.
- Two expressions e_0 and e_1 are equivalent if $\mathscr{E}[\![e_0]\!] = \mathscr{E}[\![e_1]\!]$.

Of course, if we prove the consistency of the three semantics, then we can conclude that the three notions of equivalence coincide.

Expressions with Variables

Suppose now we want to extend numerical expressions with the possibility to include formal parameters in them, drawn from an infinite set X, ranged over by x.

$$e \quad ::= \quad x \quad | \quad \mathtt{n} \quad | \quad e \oplus e \quad | \quad e \otimes e$$

How can we evaluate an expression like $(x \oplus 4) \otimes y$? We cannot, unless the values assigned to x and y are known: in general, the result will depend on them.

Operationally, we must provide such an information to the machine, e.g., in form of some memory $\sigma : X \to \mathbb{N}$ that is part of the machine state. We use the notation $\langle e, \sigma \rangle$ to denote the state where e is to be evaluated in the memory σ. The corresponding small-/big-step rules for variables would then look like:

$$\frac{}{\langle x, \sigma \rangle \to \mathtt{n}} \; n = \sigma(x) \; (var) \qquad \frac{}{\langle x, \sigma \rangle \twoheadrightarrow \mathtt{n}} \; n = \sigma(x) \; (var)$$

Exercise 1.1. The reader may complete the missing rules as an exercise.

Denotationally, the interpretation function needs to receive a memory as an additional argument:

$$\mathscr{E}[\![\cdot]\!] : Exp \to ((X \to \mathbb{N}) \to \mathbb{N})$$

Note that this is quite different from the operational approach, where the memory is part of the state.

The corresponding defining equations would then look like:

$$\begin{aligned}
\mathscr{E}[\![\mathtt{n}]\!]\sigma &= \mathscr{N}[\![\mathtt{n}]\!] \\
\mathscr{E}[\![x]\!]\sigma &= \sigma(x) \\
\mathscr{E}[\![e_0 \oplus e_1]\!]\sigma &= \mathscr{E}[\![e_0]\!]\sigma + \mathscr{E}[\![e_1]\!]\sigma \\
\mathscr{E}[\![e_0 \otimes e_1]\!]\sigma &= \mathscr{E}[\![e_0]\!]\sigma \times \mathscr{E}[\![e_1]\!]\sigma
\end{aligned}$$

Semantics equivalences must then take into account all the possible memories where expressions are evaluated. To say that e_0 is denotationally equivalent to e_1 we must require that *for any memory* $\sigma : X \to \mathbb{N}$ we have $\mathscr{E}[\![e_0]\!]\sigma = \mathscr{E}[\![e_1]\!]\sigma$.

Exercise 1.2. The reader is invited to restate the consistency between the various semantics and the operational notions of equivalences between expressions by taking memories into account.

1.3 Applications of Semantics

Whatever care is taken to make a natural language description of programming languages precise and unambiguous, there always remain some points that are open for several different interpretations. Formal semantics can provide a useful basis for the language design, its implementation, and the analysis and verification of programs.

In the following we summarise some benefits for each of the above categories.

Language Design

The worst form of design errors are unintentional ones, where the language behaves in a way that is not expected and even less desired by its designers. The effort spent in fixing a formal semantics for a language is the best way of detecting weak points in the language design itself. Starting from the natural language descriptions of the various features, subtle ambiguities, inconsistencies, complexities and anomalies will emerge, and better ways of dealing with each feature will be discovered.

While the presence of problems can be demonstrated by exhibiting example programs, their absence can only be proved by exploiting a formal semantics. Operational semantics, denotational semantics and axiomatic semantics, in this order, are increasingly sensitive tools for detecting problems in language design.

Increasingly, language designers are using semantics definitions to formalise their creations. Famous examples include Ada [6], Scheme [17] and ML [22]. A more recent witness is the use of Horn clauses to specify the type checker in the Java Virtual Machine version 7.[6]

[6] `http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf`

Implementation

Semantics can be used to validate prototype implementations of programming languages, to verify the correctness of code analysis techniques exploited in the implementation, like type checking, and to certify many useful properties, like the correctness of compilers optimisations.

A common phenomenon is the presence of underspecified behaviour in certain circumstances. In practice, such underspecified behaviours can mine programs portability from one implementation to another.

Perhaps the most significant application of operational semantics definitions is the straightforward generation of prototypal implementations, where the behaviour of programs can be simulated and tested, even if the underlying interpreter can be inefficient. Denotational semantics can also provide itself a good starting point for automatic language implementation. Automatic generation of implementations is not the only way in which formal semantics can help implementors. If a formal model is available, then hand-crafted implementations can be related to the formal semantics, e.g., to guarantee their correctness.

Analysis and Verification

Semantics offers the main support for reasoning about programs, specifications, implementations and their properties, both mechanically and by hand. It is the unique mean to state that an implementation conforms to a specification, or that two programs are equivalent, or that a model satisfies some property.

For example, let us consider the following OCaml-like functions

```
let rec fib n = match n with
                  0 -> 0
                | 1 -> 1
                | x -> fib (x-1) + fib (x-2)


let fib n = let rec faux a b cnt = match cnt with
                                     0 -> b
                                   | x -> faux (a+b) a (x-1)
            in faux 1 0 n
```

The second program offers a much more efficient version of the Fibonacci numbers calculation (the number of recursive calls is linear in n, as opposed to the first program where the number of recursive calls is exponential in n). If the two versions can be proved equivalent from the functional point of view, then we can safely replace the first version with the better performing one.

Synergy Between Different Semantics Approaches

It would be wrong to view different semantics styles as in opposition to each other. They each have their uses and their combined use is more than the sum of parts. Roughly:

- A clear operational semantics is very helpful in implementation and in proving program and language properties.
- Denotational semantics provides the deepest insights to the language designer, being sustained by a rich mathematical theory.
- Axiomatic semantics can lead to strikingly elegant proof systems, useful in developing as well as verifying programs.[7]

As discussed above, the objective of the book is to present different models of computation, their programming paradigms, their mathematical descriptions, and some formal analysis techniques for reasoning about program properties. We shall focus on the operational and denotational semantics.

A longstanding research topic is the relationship between the different forms of semantic definitions. For example, while the denotational approach can be convenient when reasoning about programs, the operational approach can drive the implementation. It is therefore of interest whether a denotational definition is equivalent to an operational one.

In mathematical logic, one uses the concepts of soundness and completeness to relate a logic's proof system to its interpretation, and in semantics there are similar notions of soundness and adequacy to relate one semantics to another.

We show how to relate different kinds of semantics and program equivalences, reconciling whenever possible the operational, denotational and logic views by proving some relevant correspondence theorems. Moreover, we discuss the fundamental ideas and methods behind these approaches.

The operational semantics fixes an abstract and concise operational model for the execution of a program (in a given environment). We define the execution as a proof in some logical system that justifies how the effect is achieved 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. The interpretation function for a typical imperative language is a mapping that, given a program, returns a function from any initial state to the corresponding final state, if any (as programs may not terminate). We cover mostly basic cases, without delving into the variety of options and features that are available to the language designer.

The correspondence is well-exemplified over the the first two paradigms we focus on: a simple IMPerative language called IMP, and a Higher-Order Functional Language called HOFL. For both of them we define what are the programs and in the case of HOFL we also define what are the infinitely many *types* we can handle.

[7] Axiomatic semantics is mostly directed towards the programmer, but its wide application is complicated by the fact that it is often difficult (more than denotational semantics) to give a clear axiomatic semantics to languages that were not designed with this in mind.

Then, we define their operational semantics, their denotational semantics and finally, to some extent, we prove the correspondence between the two.

As explained later in more detail, in the case of the last two paradigms we consider in the monograph, for concurrent and probabilistic systems, the denotational semantics becomes more complex and we replace its role by suitable logics: two systems are then considered equivalent if they satisfy exactly the same formulas in the logic. Also the perspective of the operational semantics is shifted from the computation of a result to the observable interactions with the environment and two systems are considered as equivalent if they exhibit the same behaviour (the equivalence is called *abstract semantics*). Nicely, the behavioural equivalence induced by the operational semantics can be shown to coincide with the logical equivalence above.

1.4 Key Topics and Techniques

1.4.1 Induction and Recursion

Proving existential statements can be done by exhibiting a specific witness, but proving universally quantified statements is more difficult, because all the elements must be considered (for disproving it, we can again exhibit a single counterexample) and there can be infinitely many elements to check.

The situation is improved when the elements are generated in some finitary way. For example:

- any natural number n can be obtained by taking the successor of 0 for n times;
- any well-formed program is obtained by repeated applications of the productions of some grammar;
- any theorem derived in some logic system is obtained by applying some axioms and inference rules to form a (finite) derivation tree;
- any computation is obtained by composing single steps.

If we want to prove non-trivial properties of a program or of a class of programs, we usually have to use *induction* principles. The most general notion of induction is the so called *well-founded induction* (or *Noether* induction) and we derive from it all the other inductions principles.

In the above cases (arbitrarily large but finitely generated elements) we can exploit the induction principle to prove a universally quantified statement by showing that

base case: the statement holds in all possible elementary cases (e.g., 0, the sentences of the grammar obtained by applying productions involving non-terminal symbols only, the basic derivations of a proof system obtained by applying the axioms, the initial step of a computation);

inductive case: and that the statement holds in the composite cases (e.g. $succ(n)$, the terms of the grammar obtained by applying productions involving non-terminal symbols, the derivations of a proof system

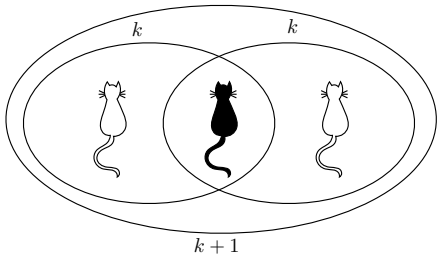
obtained by applying an inference rule to smaller derivations, a computation of $n+1$ steps, etc.), under the assumption that it holds in any simpler case (e.g., for any $k \leq n$, for any sub-terms, for any smaller derivation, for any computation whose length is smaller or equal than n).

Exercise 1.3. Induction can be deceptive. Let us consider the following argument for proving that all cats are the same colour.

Let $P(n)$ be the proposition that: *In a group of n cats, all cats are the same colour*
The statement is trivially true for $n = 1$ (base case).

For the inductive case, suppose that the statement is true for $n \leq k$. Take a group of $k+1$ cats: we want to prove that they are the same colour.

Align the cats along a line. Form two groups of k cats each: the first k cats in the line and the last k cats of the line. By inductive hypothesis, the cats in the two groups are the same colours. Since the cat in the middle of the line belongs to both groups, by transitivity all cats in the line are the same colour. Hence $P(k+1)$ is true.



By induction, $P(n)$ is true for all $n \in \omega$.
Hence, all cats are the same colour.
We know that this cannot be the case: What's wrong with the above reasoning?

The usual proof technique for proving properties of a natural semantics definition is induction on the height of the derivation trees that are generated from the semantics, or is the special case of rule induction.

base cases: P holds for each axiom, and
inductive cases: for each inference rule, if P holds for the premises, then it holds for the conclusion.

For proving properties of a denotational semantics, induction on the structure of the terms is often a convenient proof strategy.

Defining the denotational semantics of 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.

1.4.2 Semantic Domains

The choice of a suitable semantic domain is not always as easy as in the example of numerical expressions.

For example, the semantics of programs is often formulated in a functional space, from the domain of states to itself (i.e., a program is seen as a state-transformation). The functions we need to consider can be partial ones, if the programs can diverge. Note that the domain of states can also be a complex structure, e.g., a state can be an assignment of values to variables.

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, therefore to assign a meaning $[\![p]\!]$ to the program p we need to solve a recursive equation like:

$$[\![p]\!] = f([\![p]\!]). \tag{1.1}$$

In general, it can happen than such equations have none, one or many solutions. Solutions to recursive equations are called *fixpoints*.

Example 1.3. Let us consider the domain of natural numbers

$n = 2 \times n$	has only one solution: $n = 0$
$n = n + 1$	has no solution
$n = 1 \times n$	has many solutions: any n

Example 1.4. Let us consider the domain of sets of integers

$X = X \cap \{1\}$	has two solutions: $X = \varnothing$ or $X = \{1\}$
$X = \mathbb{N} \setminus X$	has no solution
$X = X \cup \{1\}$	has many solutions: any $M \supseteq \{1\}$

In order to provide a general solution to this kind of problems, we resort to the theory of *complete partial orders with bottom* and of *continuous* functions.

In the functional programming paradigm, a higher-order functional language can use functions as arguments to other functions, i.e., spaces of functions must also be considered as forming data types. This makes the language's domains more complex. Denotational semantics can be used to understand these complexities; an applied branch of mathematics called *domain theory* is used to formalise the domains with algebraic equations.

Let us consider a domain D where we interpret the elements of some data type. The idea is that two elements $x, y \in D$ are not necessarily separated, but one, say y can be a better version of what x is trying to approximate, written

$$x \sqsubseteq y$$

with the intuition that y is *consistent* with x and is (possibly) *more accurate* than x.

Concretely, a special interesting case is when one can take two partial functions f, g and say that g is a better approximation than f if whenever $f(x)$ is defined then also $g(x)$ is defined and $g(x) = f(x)$. But g can be defined on elements over which f is not.

Note that if we see (partial) functions as relations (sets of pairs $(x, f(x))$), then the above concept boils down to set inclusion.

For example, we can progressively approximate the factorial function by taking the sequence of partial functions

$$\varnothing \subseteq \{(1,1)\}\{(1,1),(2,2)\} \subseteq \{(1,1),(2,2),(3,6)\} \subseteq \{(1,1),(2,2),(3,6),(4,24)\} \subseteq \cdots$$

Now, it is quite natural to require that our notion of approximation $\sqsubseteq$ is reflexive, transitive and antisymmetric: this means that our domain D is a *partial order*.

Often there is an element, called *bottom* and denoted by $\bot$, which is less defined than any other element: in our example about partial function, the bottom element is the partial function $\varnothing$.

When we apply a function f (determined by some program) to elements of D it is also quite natural to require that the more accurate the input, the more accurate the result:

$$x \sqsubseteq y \quad \Rightarrow \quad f(x) \sqsubseteq f(y)$$

this means that our functions of interest are *monotonic*.

Now suppose we are given an infinite sequence of approximations

$$x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq \ldots$$

it seems reasonable to suppose that the sequence tends to some limit that we denote as $\bigsqcup_n x_n$ and moreover that mappings between data types are well-behaving w.r.t. limits, i.e., that data transformations are *continuous*:

$$f\left(\bigsqcup_n x_n\right) = \bigsqcup_n f(x_n)$$

Interestingly, one can prove that for a function to be continuous in several variables jointly, it is sufficient that it be continuous in each of its variables separately.

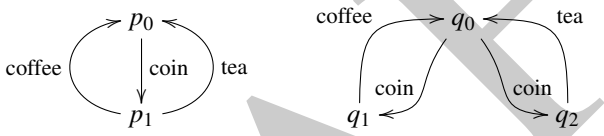
Kleene's fixpoint theorem ensures that when continuous functions are considered over complete partial orders (with bottom), then a suitable *least* fixpoint exists and tells us how to compute it. The fixpoint theory is first applied to the case of IMP semantics and then extended to handle HOFL. The case of HOFL is more complex because we are working on a more general situation where functions are first class citizens.

When defining coarsest equivalences over concurrent processes, we also present a weaker version of the fixpoint theorem by Knaster and Tarski that can be applied to monotone functions (not necessarily continuous) over complete lattices.

1.4.3 Bisimulation

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 behaviour not to depend as much as possible on the speed of processes, on the other hand we want to permit infinite computations and to differentiate among them on the basis of the interactions they can undertake. For example, in the case of IMP and HOFL all diverging programs are considered as equivalent. The typical models for nondeterminism and infinite computations are *(labelled) transition systems*. We do not consider time explicitly, but we have to introduce nondeterminism to account for races between concurrent processes.

In the case of interactive, concurrent systems, as represented by labelled transition systems, the classic notion of language equivalence from finite automata theory is not best suited as a criterion for program equivalence, because it does not account properly for non-terminating computations and non-deterministic behaviour. To see this, consider the two labelled transition systems below, which can be thought to model the behaviour of two different coffee machines:



It is evident that any sequence of actions that is executable by the first machine can be also executed on the second machine, and vice versa. However, from the point of view of the interaction with the user, the two machines behave very differently: after the introduction of the coin, the machine on the left still allows the user to choose between a coffee and a tea; while the machine on right leaves no choice to the user.

We show that a suitable notion of equivalence between concurrent, interactive systems can be defined as a *behavioural equivalence* called *bisimulation*: it takes into account the branching structure of labelled transition systems as well as infinite computations. Equivalent programs are represented by (initial) states which have correspondent observable transitions leading to equivalent states. Interestingly, there is a nice connection between fixpoint theory and the definition of the coarsest bisimulation equivalence, called *bisimilarity*. Moreover, bisimilarity finds a logical counterparts in *Hennessy-Milner logic*, in the sense that two systems are bisimilar if and only if they satisfy the same Hennessy-Milner logic formulas. Beside using bisimilarity to compare different realisations of the same system, weaker forms of bisimilarity can be used to study the compliance between an abstract specification and a concrete implementation.

The language that we employ in this setting is a *process algebra* called CCS (*Calculus for Communicating Systems*). Then, we study systems whose communication structure can change during execution. These systems are called *open-ended*. As our case study, we 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.

1.4.4 Temporal and Modal Logics

We investigate also *temporal* and *modal logics* designed to conveniently express properties of concurrent, interactive systems.

Modal logics were conceived in philosophy to study different *modes of truth*, like an assertion being false in the current world but *possibly* true in some alternate world, or another to *always* hold true in all worlds. Temporal logics are an instance of modal logics to reason about the truth of assertions over time. Typical temporal operators includes the possibility to assert that a property is true *sometimes* in the future, or that is *always* true, in all the future moments. The most popular temporal logics are LTL (Linear Temporal Logic) and CTL (Computation Tree Logic). They have been extensively studied and used for applying formal methods to industrial case studies and for the specification and verification of program correctness.

We introduce the basics of LTL and CTL and then present a modal logic with recursion, called the μ*-calculus*, that encompasses LTL and CTL. The definition of the semantics of μ-calculus exploits again the principles of domain theory and fixpoint computation.

1.4.5 Probabilistic Systems

Finally, in the last part of the book we focus on probabilistic models, where we trade nondeterminism for probability distributions, which we associate to choice points.

Probability theory is playing a big role in modern computer science. It focuses on the study of random events, which are central in areas such as artificial intelligence and network theory, e.g., to model variability in the arrival of requests and predict load distribution on servers. Probabilistic models of computation assign weight to choices and refine non-deterministic choices with probability distributions. In interactive systems, when many actions are enabled at the same time, the probability distribution models the frequency with which each alternative can be executed. Probabilistic models can also be used in conjunction with sources of non-determinism and we present several ways in which this can be achieved. We also present stochastic models, where actions take place in a continuous time setting, with an exponential distribution.

A compelling case of probabilistic systems is given by *Markov chains*, which represent random processes over time. We study two kinds of Markov chains, which differ for the way in which time is represented (discrete vs continuous) and we focus on homogeneous chains only, where the distribution depends on the current state of the system, but not on the current time. For example, in some special cases, Markov

chains can be used to estimate the probability to find the system in a given state on the long run or the probability that the system will not its change state in some time.

By analogy with labelled transition systems we are also able to define suitable notions of bisimulation and the analogous of Hennessy-Milner logic, called *Larsen-Skou logic*. Finally, by analogy with CCS, we present a high-level language for the description of continuous time Markov chains, called PEPA, which can be used to define stochastic systems in a structured and compositional way as well as by refinement from specifications. PEPA is tool-supported and has been successfully applied in many fields, e.g., in performance evaluation, decision support systems and system biology.

1.5 Chapters Contents and Reading Guide

After Chapter 2, where some notation is fixed and useful preliminaries about logical systems, goal-oriented derivations and proof strategies are explained, the book comprises four main parts: the first two parts exemplify deterministic systems; the other two models non-deterministic ones. The difference will emerge clear during the reading.

- Computational models for imperative languages, exemplified over IMP:
  - In Chapter 3 the simple imperative language IMP is introduced, its natural semantics is defined and studied together with the induced notion of program equivalence.
  - In Chapter 4 the general principle of well-founded induction is stated and declined to other widely used induction principles, like mathematical induction, structural induction and rule induction. The chapter is concluded by illustrating well-founded recursive definitions.
  - In Chapter 5 the mathematical basis for denotational semantics are presented, including the concepts and properties of complete partial orders, of least upper bounds, and of monotone and continuous functions. In particular this chapter contains Kleene's fixpoint theorem that is used extensively in the rest of the monograph and the definition of the immediate consequence operator associated with a logical system, which is exploited in Chapter 6. The presentation of Knaster-Tarski's fixpoint is instead postponed to Chapter 11.
  - In Chapter 6 the foundations introduced in Chapter 5 are exploited to define the denotational semantics of IMP and to derive a corresponding notion of program equivalence. The induction principles studied in Chapter 4 are then exploited to prove the correspondence between the operational and denotational semantics of IMP and consequently of their two induced equivalences over processes. The chapter is concluded by presenting Scott principle of *computational induction* for proving *inclusive* properties.
- Computational models for functional languages, exemplified over HOFL

  - In Chapter 7 we shift from the imperative style of programming to the declarative one. After presenting the λ-notation, useful for representing anonymous functions, the higher-order functional language HOFL is introduced, where infinitely many data-types can be constructed by pairing and function type constructors. Church type theory and Curry type theory are discussed and the unification algorithm from Chapter 2 is used for type inference. Typed terms are given a natural semantics called *lazy*, because it evaluates a parameter of a function only if needed. The alternative *eager* semantics, where actual argument are always evaluated is also discussed.
  - In Chapter 8 we extend the theory presented in Chapter 5 to allow the construction of more complex domains, as needed by the type constructors available in HOFL.
  - In Chapter 9 the foundations introduced in Chapter 8 are exploited to define the (lazy) denotational semantics of HOFL.
  - In Chapter 10 the operational and denotational semantics of HOFL are compared, by showing that notion of program equivalence induced by the former is generally stricter than the one induced by the latter and that they coincide only over terms of type integer. However, it is shown that the two semantics are equivalent w.r.t. the notion of convergence.

- Computational models for concurrent / non-deterministic / interactive languages, exemplified over CCS and pi-calculus

  - In Chapter 11 we shift the focus from sequential systems to concurrent and interactive ones. The process algebra CCS is introduced which allows to describe concurrent communicating systems. Such systems communicate by message passing over named channels. Their operational semantics is defined in the small-step style, because infinite computations must be accounted for. Communicating process are assigned labelled transition systems by inference rules in the SOS-style and several equivalences over such transition systems are discussed. In particular the notion of behavioural equivalence is put forward, in the form of bisimulation equivalence. Notably, the coarsest bisimulation, called bisimilarity, exists, it can be characterised as a fixpoint, it is a congruence w.r.t. the operators of CCS and it can be axiomatised. Its logical counterpart, called Hennessy-Milner logic, is also presented. Finally, coarser equivalences are discussed, which can be exploited to relate system specifications with more concrete implementations by abstracting away from internal moves.
  - In Chapter 12 some logics are considered that increase the expressiveness of Hennessy-Milner logic by defining properties about finite and infinite computations. First the temporal logics LTL and CTL are presented, and then the more expressive μ-calculus is studied. The notion of satisfaction for μ-calculus formulas is defined by exploiting fixpoint theory.
  - In Chapter 13 the theory of concurrent systems is extended with the possibility to communicate channel names and create new channels. Correspondingly, we move from CCS to the π-calculus, we define its small-step operational semantics and we introduce several notions of bisimulation equivalence.

- Computational models for probabilistic and stochastic process calculi
  - In Chapter 14 we shift the focus from non-deterministic systems to probabilistic ones. After introducing the basics of measure theory and the notions of random process and Markow property, two classes of random processes are studied, which differ for the way time is represented: DTMC (discrete time) and CTMC (continuous time). In both cases, it is studied how to compute stationary probability distribution over the possible states and suitable notion of bisimulation equivalence.
  - In Chapter 15, the various possibilities for defining probabilistic models of computation with observable actions and sources of non-determinism are overviewed, emphasising the difference between reactive models and generative ones. Finally a probabilistic version of Hennessy-Milner logic is presented, called Larsen-Skou logic.
  - In Chapter 16 a well-known high-level language for the specification and analysis of stochastic interactive systems, called PEPA (Performance Evaluation Process Algebra), is presented. The small-step operational semantics of PEPA is first defined and then it is shown how to associate a CTMC to each PEPA process.

1.6 Further Reading

One leitmotif of this monograph is the use of logical systems of inference rules. As derivation trees tend to grow large very fast, even for small examples, we will introduce and rely on goal-oriented derivations inspired by logic programming, as explained in Section 2.3. A nice introduction to the subject can be found in the lecture notes[8] by Frank Pfenning [26]. The first chapters cover, in a concise but clear manner, most of the concepts we shall exploit.

The reader interested in knowing more about the theory of partial orders and domains is referred to (the revised edition of) the book by Davey and Priestley [5] for a gentle introduction to the basic concepts and to the chapter by Abramsky and Jung in the Handbook of Logic in Computer Science for a full account of the subject [1]. A freely available document on domain theory that accounts also for the case of parallel and nondeterministic systems is the so-called "Pisa notes" by Plotkin [29]. The reader interested in denotational semantics methods only can then consult [35] for an introduction to the subject and [10] for a comprehensive treatment of programming language constructs, including different procedure call mechanisms.

There are several books on the semantics of imperative and functional languages [12, 41, 23, 24, 32, 40, 7]. For many years, we have adopted the book by Glynn Winskel [43] for the courses in Pisa, which is possibly the closest to our approach. It covers most of the content of Parts II (IMP) and III (HOFL) and has a chapter on CCS and modal logic (see Part IV), there discussed together with another

[8] Freely available at the time of the publication.

well-known process algebras for concurrent systems called CSP (for Communicating Sequential Processes) and introduced by Tony Hoare. The main differences are that we use goal-oriented derivations for logical systems (type systems and operational semantics) and focus on the lazy semantics of HOFL, while Winskel's book exploits derivation trees and give a detailed treatment of the eager semantics of HOFL. We briefly discuss CSP in Chapter 16 as it is the basis for PEPA. Chapter 13 and Part V are not covered in [43]. We briefly overviews elementary type systems in connection to HOFL. To deepen the study of the topic, including polymorphic and recursive types, we recommend the books by Benjamin Pierce [27] and by John Mitchell [23].

Moving to Part IV, the main and most cited reference for CCS is Robin Milner's book [20]. However, for an up-to-date presentation of CCS theory, we refer to the very recent book by Roberto Gorrieri and Christian Versari [11]. Both texts are complementary to the book by Luca Aceto et al. [2], where the material is organised so to pose the emphasis on verification aspects of concurrent systems and CCS is presented as a useful formal tool. Mobility is not considered in the above texts. The basic reference for the π-calculus is the seminal book by Robin Milner [21]. The whole body of theory that have been subsequently developed is presented at a good level of detail in the book by Davide Sangiorgi and David Walker [34]. Many free tutorials on CCS and π-calculus can also be found on the web in various languages.

Bisimulation equivalences are presented and exploited in all the above books, but their use, as well as that of the more general concept of coinduction, spans far beyond CCS and interactive systems. The new book by Davide Sangiorgi [33] explores the subject from many angles and provide good insights. The algorithmic-minded reader is also referred to the recent survey [3].

The literature on temporal and modal logics and their applications to verification and model-checking is quite vast and falls out of the scope of our book. We just point the reader to the compact survey on the modal μ-calculus by Giacomo Lenzi [18], that explains synthetically how LTL and CTL can be seen as sublogics of the μ-calculus, and to the book by Christel Baier and Joost-Pieter Katoen on model checking principles [4] where also verification of probabilistic systems is addressed.

This brings us to Part V. We think one peculiarity of this monograph is that it groups under the same umbrella several paradigms that are often treated in separation. This is certainly the case of Markov chains and probabilistic systems. Markov chains are usually studied in first courses on probability for Computer Science. Their combined use with transitions for interaction is a more advanced subject and we refer the interested reader to the well-known book by Prakash Panangaden [25].

Finally, PEPA, where the process algebraic approach merges with the representation of stochastic systems, allowing to model and measure not just the expressiveness of processes but also their performance, under many angles. The introductory text for PEPA principles is the book by Jane Hillston [14] possibly accompanied by the short presentation in [13]. For people interested in experimenting with PEPA we refer instead to [9].

References

1. Samson Abramsky and Achim Jung. Domain theory. In *Handbook of Logic in Computer Science*, pages 1–168. Clarendon Press, 1994.
2. Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, New York, NY, USA, 2007.
3. Luca Aceto, Anna Ingolfsdottir, and Jir Srba. The algorithmics of bisimilarity. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, pages 100–172. Cambridge University Press, 2011. Cambridge Books Online.
4. Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.
5. B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge mathematical text books. Cambridge University Press, 2002.
6. Véronique Donzeau-Gouge, Gilles Kahn, and Bernard Lang. On the formal definition of ADA. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark, January 14-18, 1980*, volume 94 of *Lecture Notes in Computer Science*, pages 475–489. Springer, 1980.
7. Maribel Fernández. *Programming Languages and Operational Semantics - A Concise Overview*. Undergraduate Topics in Computer Science. Springer, 2014.
8. Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.
9. Stephen Gilmore and Jane Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In Günter Haring and Gabriele Kotsis, editors, *Computer Performance Evaluation, Modeling Techniques and Tools, 7th International Conference, Vienna, Austria, May 3-6, 1994, Proceedings*, volume 794 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 1994.
10. M.J.C. Gordon. *The denotational description of programming languages: an introduction*. Springer-Verlag, 1979.
11. Roberto Gorrieri and Cristian Versari. *Introduction to Concurrency Theory - Transition Systems and CCS*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2015.
12. Matthew Hennessy. *The semantics of programming languages - an elementary introduction using structural operational semantics*. Wiley, 1990.
13. Jane Hillston. Compositional markovian modelling using a process algebra. In William J. Stewart, editor, *Computations with Markov Chains: Proceedings of the 2nd International Workshop on the Numerical Solution of Markov Chains*, pages 177–196, Boston, MA, 1995. Springer US.
14. Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, New York, NY, USA, 1996.
15. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
16. Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.
17. Richard Kelsey, William D. Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.
18. Giacomo Lenzi. The modal μ-calculus: a survey. *TASK Quarterly*, 9(3):293–316, 2005.
19. John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.
20. Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.
21. Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

22. Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990.
23. John C. Mitchell. *Foundations for programming languages*. Foundation of computing series. MIT Press, 1996.
24. Hanne Riis Nielson and Flemming Nielson. *Semantics with applications - a formal introduction*. Wiley professional computing. Wiley, 1992.
25. Prakash Panangaden. *Labelled Markov Processes*. Imperial College Press, 2009.
26. Frank Pfenning. Logic programming, 2007. Lecture notes. Available at `http://www.cs.cmu.edu/~fp/courses/lp/`.
27. Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
28. Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus, Denmark, 1981.
29. Gordon D. Plotkin. Domains (the 'Pisa' Notes), 1983. Notes for lectures at the University of Edinburgh, extending lecture notes for the Pisa summer school 1978. Available at `http://homepages.inf.ed.ac.uk/gdp/publications/Domains_a4.ps`.
30. Gordon D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004.
31. Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
32. John C. Reynolds. *Theories of programming languages*. Cambridge University Press, 1998.
33. Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 2011.
34. Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.
35. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.
36. David A. Schmidt. Programming language semantics. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2237–2254. CRC Press, 1997.
37. Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In Jerome Fox, editor, *Proceedings of the Symposium on Computers and Automata*, volume XXI, pages 19–46, Brooklyn, N.Y., 1971. Polytechnic Press.
38. Dana S. Scott. Outline of a Mathematical Theory of Computation. Technical Report PRG–2, Programming Research Group, Oxford, England, November 1970.
39. Dana S. Scott. Some reflections on strachey and his work. *Higher-Order and Symbolic Computation*, 13(1/2):103–114, 2000.
40. Aaron Stump. *Programming Language Foundations*. Wiley, 2013.
41. Robert D. Tennent. *Semantics of programming languages*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.
42. Adriaan van Wijngaarden, B. J. Mailloux, J. E. L. Peck, Cornelis H. A. Koster, Michel Sintzoff, C. H. Lindsey, Lambert G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language ALGOL 68. *Acta Inf.*, 5:1–236, 1975.
43. Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993.

Chapter 2
Preliminaries

A mathematician is a device for turning coffee into theorems.
(Paul Erdos)

Abstract In this chapter we fix some basic mathematical notation used in the rest of the book and introduce the important concepts of signature, logical system and goal-oriented derivation.

2.1 Notation

2.1.1 Basic Notation

As a general rule, we use capital letters, like D or X, to denote sets of elements and small letters, like d or x, for their elements, with membership relation $\in$. The set of natural numbers is denoted by $\mathbb{N} \stackrel{\text{def}}{=} \{0,1,2,...\}$, the set of integer numbers is denoted by $\mathbb{Z} \stackrel{\text{def}}{=} \{...,-2,-1,0,1,2,...\}$ and the set of boolean by $\mathbb{B} \stackrel{\text{def}}{=} \{\mathbf{true},\mathbf{false}\}$. We write $[m,n] \stackrel{\text{def}}{=} \{k \mid m \leq k \leq n\}$ for the interval of numbers from m to n, extremes included. If a set A is finite, we denote by $|A|$ its *cardinality,* i.e., the number of its elements. The emptyset is written $\varnothing$, with $|\varnothing| = 0$. We use the standard notation for set union, intersection, difference, cartesian product and disjoint union, which are denoted respectively by $\cup$, $\cap$, $\setminus$, $\times$ and $\uplus$. We write $A \subseteq B$ if all elements in A belong to B. We denote by $\wp(A)$ the powerset of A, i.e., the set of all subsets of A.

An indexed set of elements is written $\{e_i\}_{i\in I}$ and a family of sets is written $\{S_i\}_{i\in I}$. Set operations are extended to families of sets by writing, e.g., $\bigcup_{i\in I} S_i$ and $\bigcap_{i\in I} S_i$. If I is the interval set $[m,n]$, then we write also $\bigcup_{i=m}^{n} S_i$ and $\bigcap_{i=m}^{n} S_i$.

Given a set A, and a natural number k we denote by A^k the set of sequences of k (not necessarily distinct) elements in A. Such sequences are called *strings* and their concatenation is represented by juxtaposition. We denote by $A^* = \bigcup_{k\in\mathbb{N}} A^k$ the set of all finite (possibly empty) sequences over A. Given a string $w \in A^*$ we denote by $|w|$ its *length*, i.e., the number of its elements (including repetitions). The empty string is denoted ε, and we have $|\varepsilon| = 0$ and $A^0 = \{\varepsilon\}$ for any A. We denote by $A^+ = \bigcup_{k>0} A^k = A^* \setminus \{\varepsilon\}$ the set of all finite non-empty sequences over A.

A relation R between two sets A and B is a subset of $A \times B$. For $(a,b) \in R$ we write also aRb. A relation $f \subseteq A \times B$ can be regarded as a function if both the following properties are satisfied:

function: $\forall a \in A, \forall b_1, b_2 \in B.\ (a,b_1) \in f \wedge (a,b_2) \in f \ \Rightarrow\ b_1 = b_2$
total: $\forall a \in A, \exists b \in B.\ (a,b) \in f$

For such a function f, we write $f : A \to B$ and say that the set A is the *domain* of f, and B is its *codomain*. We write $f(a)$ for the unique element $b \in B$ such that $(a,b) \in f$, i.e., f can be regarded as the relation $\{(a, f(a)) \mid a \in A\} \subseteq A \times B$. The composition of two functions $f : A \to B$ and $g : B \to C$ is written $g \circ f : A \to C$, it is such that for any element $a \in A$ it holds $(g \circ f)(a) = g(f(a))$. A relation that satisfies the "function" property, but not necessarily the "total" property, is called *partial*. A partial function f from A to B is written $f : A \rightharpoonup B$.

2.1.2 Signatures and Terms

A one-sorted (or unsorted) *signature* is a set of function symbols $\Sigma = \{c, f, g, ...\}$ such that each symbol in Σ is assigned an *arity*, that is the number of arguments it takes. A symbol with arity zero is called a *constant*; a symbol with arity one is called *unary*; a symbol with arity two is called *binary*; a symbol with arity three is called *ternary*. For $n \in \mathbb{N}$, we let $\Sigma_n \subseteq \Sigma$ be the set of function symbols whose arity is n.

Given an infinite set of variables $X = \{x, y, z, ...\}$, the set $T_{\Sigma,X}$ is the set of terms over Σ and X, i.e., the set of all and only terms generated according to the following rules:

- each variable $x \in X$ is a term (i.e., $x \in T_{\Sigma,X}$),
- each constant $c \in \Sigma_0$ is a term (i.e., $c \in T_{\Sigma,X}$),
- if $f \in \Sigma_n$, and $t_1, ..., t_n$ are terms (i.e., $t_1, ..., t_n \in T_{\Sigma,X}$), then also $f(t_1, ..., t_n)$ is a term (i.e., $f(t_1, ..., t_n) \in T_{\Sigma,X}$).

For a term $t \in T_{\Sigma,X}$, we denote by $\mathrm{vars}(t)$ the set of variables occurring in t, and let $T_\Sigma \subseteq T_{\Sigma,X}$ be the set of terms with no variables, i.e., $T_\Sigma \stackrel{\text{def}}{=} \{t \in T_{\Sigma,X} \mid \mathrm{vars}(t) = \varnothing\}$.

Example 2.1. For example, take $\Sigma = \{0, succ, plus\}$ with 0 a constant, *succ* unary and *plus* binary. Then all of the following are terms:

- $0 \in T_\Sigma$
- $x \in T_{\Sigma,X}$
- $succ(0) \in T_\Sigma$
- $succ(x) \in T_{\Sigma,X}$
- $plus(succ(x), 0) \in T_{\Sigma,X}$
- $plus(plus(x, succ(y)), plus(0, succ(x))) \in T_{\Sigma,X}$

The set of variables of the above terms are respectively:

- $\mathrm{vars}(0) = \mathrm{vars}(succ(0)) = \varnothing$

- $\mathrm{vars}(x) = \mathrm{vars}(succ(x)) = \mathrm{vars}(plus(succ(x),0)) = \{x\}$
- $\mathrm{vars}(plus(plus(x,succ(y)),plus(0,succ(x)))) = \{x,y\}$

Instead $succ(plus(0),x)$ is not a term: can you see why?

2.1.3 Substitutions

A *substitution* $\rho : X \to T_{\Sigma,X}$ is a function assigning terms to variables.

Since the set of variables is infinite while we are interested only in terms with a finite number of variables, we consider only substitutions that are defined as identity everywhere except on a finite number of variables. Such substitutions are written

$$\rho = [x_1 = t_1, \ldots, x_n = t_n]$$

meaning that

$$\rho(x) = \begin{cases} t_i & \text{if } x = x_i \\ x & \text{otherwise} \end{cases}$$

We denote by $t\rho$, or sometimes by $\rho(t)$, the term obtained from t by simultaneously replacing each variable x with $\rho(x)$ in t.

Example 2.2. For example, consider the signature from Example 2.1, the term $t \stackrel{\text{def}}{=} plus(succ(x),succ(y))$ and the substitution $\rho \stackrel{\text{def}}{=} [x = succ(y), y = 0]$. We get:

$$t\rho = plus(succ(x),succ(y))[x = succ(y), y = 0] = plus(succ(succ(y)),succ(0))$$

We say that the term t is *more general* than the term t' if there exists a substitution ρ such that $t\rho = t'$. The "more general than" relation is reflexive and transitive, i.e., it defines a *pre-order*. Note that there are terms t and t', with $t \neq t'$, such that t is more general than t' and t' is more general than t.

We say that the substitution ρ is more general than the substitution ρ' if there exists a substitution ρ'' such that for any variable x we have that $\rho''(\rho(x)) = \rho'(x)$ (i.e., $\rho(x)$ is more general than $\rho'(x)$ as witnessed by ρ'').

2.1.4 Unification Problem

The unification problem, in its simplest formulation (syntactic, first-order unification), consists of finding a substitution ρ that identifies some terms pairwise.

Formally, given a set of potential equalities

$$G = \{l_1 \stackrel{?}{=} r_1, \ldots, l_n \stackrel{?}{=} r_n\}$$

where $l_i, r_i \in T_{\Sigma,X}$, we say that a substitution ρ is a solution of G if

$$\forall i \in [1,n].\ l_i\rho = r_i\rho.$$

The unification problem, consists in finding a *most general* substitution ρ.

We say that two sets of potential equalities G and G' are *equivalent* if they have the same set of solutions.

We denote by $\mathrm{vars}(G)$ the set of variables occurring in G, i.e.:

$$\mathrm{vars}(\{l_1 \stackrel{?}{=} r_1, ..., l_n \stackrel{?}{=} r_n\}) = \bigcup_{i=1}^{n} (\mathrm{vars}(l_i) \cup \mathrm{vars}(r_i))$$

Note that the solution does not necessarily exists, and when it exists it is not necessarily unique.

The unification algorithm takes as input a set of potential equalities G as the one above and applies some transformations until:

- either it terminates (no transformation can be applied any more) after having transformed the set G to an equivalent set of equalities

  $$G' = \{x_1 \stackrel{?}{=} t_1, ..., x_k \stackrel{?}{=} t_k\}$$

  where $x_1, ..., x_k$ are all distinct variables and $t_1, ..., t_k$ are terms with no occurrences of $x_1, ..., x_k$, i.e., such that $\{x_1, ..., x_k\} \cap \bigcup_{i=1}^{k} \mathrm{vars}(t_i) = \varnothing$: the set G' directly defines a most general solution

  $$[x_1 = t_1, ..., x_k = t_k]$$

  to the unification problem G;
- or it fails, meaning that the potential equalities cannot be unified.

In the following we denote by $G\rho$ the set of potential equalities obtained by applying the substitution ρ to all terms in G. Formally:

$$\{l_1 \stackrel{?}{=} r_1, ..., l_n \stackrel{?}{=} r_n\}\rho = \{l_1\rho \stackrel{?}{=} r_1\rho, ..., l_n\rho \stackrel{?}{=} r_n\rho\}$$

The unification algorithm tries to apply the following steps (the order is not important for the result, but it may affect complexity), to transform an initial set of potential equalities until no more steps can be applied or the algorithm fails:

delete:	$G \cup \{t \stackrel{?}{=} t\}$ is transformed to G
decompose:	$G \cup \{f(t_1, ..., t_m) \stackrel{?}{=} f(u_1, ..., u_m)\}$ is transformed to $G \cup \{t_1 \stackrel{?}{=} u_1, ..., t_m \stackrel{?}{=} u_m\}$
swap:	$G \cup \{f(t_1, ..., t_m) \stackrel{?}{=} x\}$ is transformed to $G \cup \{x \stackrel{?}{=} f(t_1, ..., t_m)\}$
eliminate:	$G \cup \{x \stackrel{?}{=} t\}$ is transformed to $G[x = t] \cup \{x \stackrel{?}{=} t\}$ if $x \in \mathrm{vars}(G) \wedge x \notin \mathrm{vars}(t)$
conflict:	$G \cup \{f(t_1, ..., t_m) \stackrel{?}{=} g(u_1, ..., u_h)\}$ leads to failure if $f \neq g \vee m \neq h$
occur check:	$G \cup \{x \stackrel{?}{=} f(t_1, ..., t_m)\}$ leads to failure if $x \in \mathrm{vars}(\, f(t_1, ..., t_m)\,)$

Example 2.3. For example, if we start from

$$G = \{plus(succ(x), x) \stackrel{?}{=} plus(y, 0)\}$$

by applying rule **decompose** we obtain

$$\{succ(x) \stackrel{?}{=} y\,,\, x \stackrel{?}{=} 0\}$$

by applying rule **eliminate** we obtain

$$\{succ(0) \stackrel{?}{=} y\,,\, x \stackrel{?}{=} 0\}$$

finally, by applying rule **swap** we obtain

$$\{y \stackrel{?}{=} succ(0)\,,\, x \stackrel{?}{=} 0\}$$

Since no further transformation is possible, we conclude that

$$\rho = [y = succ(0)\,,\, x = 0]$$

is the most general unifier for G.

2.2 Inference Rules and Logical Systems

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 the operational semantics by induction on the syntax structure of the programs).

Definition 2.1 (Inference rule). Let $x_1, x_2, \ldots, x_n, y$ be (well-formed) formulas. An *inference rule* is written, using inline notation, as

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

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

Definition 2.2 (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 an axiom r, hence there is nothing to prove in order to apply the rule: in this case we can assume y to hold.

Definition 2.3 (Logical system). A *logical system* is a set of inference rules $R = \{r_i\}_{i \in I}$.

Given a logical system, 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 (used as premises). In turn, the newly derived formulas can be used to prove the validity of other formulas.

Example 2.4 (Some inference rules). The inference rule

$$\frac{x \in E \quad y \in E \quad x \oplus y = z}{z \in E}$$

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

The rule

$$\frac{}{2 \in E}$$

is an axiom, so we know that 2 belongs to the set E.

By composing inference rules, we build *derivations*, which explain how a logical deduction is achieved.

Definition 2.4 (Derivation). Given a logical system 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 there are some derivations $d_1 \Vdash_R x_1, \ldots, d_n \Vdash_R x_n$ such that $d = (\{d_1, \ldots, d_n\} / y)$ and $(\{x_1, \ldots, x_n\} / y) \in R$.

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

Let us look more closely at the two cases in Definition 2.4. The first case tells us that if we know that:

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

i.e., if 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, in the logical system R, we have a rule for deriving y using $x_1, ..., x_n$ as premises, i.e.,

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

then we can build a derivation for y as follows:

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

Summarising all the above:

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

A derivation can roughly be seen as a tree whose root is the formula y we derive and whose leaves are the axioms we need. Correspondingly, we can define the height of a derivation tree as follows:

$$height(d) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } d = (\varnothing / y) \\ 1 + \max\{height(d_1), ..., height(d_n)\} & \text{if } d = (\{d_1, \ldots, d_n\} / y) \end{cases}$$

Definition 2.5 (Theorem). A *theorem* in a logical system 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 in R if $\exists d.d \Vdash_R y$.

Definition 2.6 (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*: we start from theorems descending from the axioms and then prove more and more theorems by applying the inference rules to already proved theorems;
- *bottom-up* or *goal-oriented*: we fix a goal, i.e., a theorem we want to prove, and we try to deduce a derivation for it by applying the inference rules backward, until each needed premise is also proved.

In the following we will mostly follow the *bottom-up* approach, because we will be given a specific goal to prove.

Example 2.5 (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 ε denotes the empty string, we write:

$$S \quad ::= \quad S\,S \quad | \quad (S) \quad | \quad \varepsilon$$

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

$$S \quad ::= \quad S\,S$$

says that given any two strings s_1 and s_2 of balanced parentheses, their juxtaposition is also a string of balanced parentheses. In other words:

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

Similarly, the second production

$$S \quad ::= \quad (S)$$

says that we can surround with brackets any string s of balanced parentheses and get again a string of balanced parentheses. In other words:

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

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

$$\frac{}{\varepsilon \in L_S}\,(3)$$

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

For example, the rule

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

is an instance of rule (1): it is obtained by replacing s_1 with $)\,($ and s_2 with $(\,($. Of course the string $)\,(\,(\,($ appearing in the conclusion does not belong to L_S, but

the rule instance is perfectly valid, because it says that "`)(((` $\in L_S$ *if* `)(` $\in L_S$ and `(((` $\in L_S$": since the premises are false, the implication is valid even if we cannot draw the conclusion `)(((` $\in L_S$.

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

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

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

Let us introduce a second formalisation of the same language (balanced parentheses) without using inference rules. To get an intuition, suppose we want to write an algorithm to check if the parentheses in a string are balanced. We can parse the string from left to right and count the number of unmatched, open parentheses in the prefix we have parsed. So, we sum 1 to the counter whenever we find an open parenthesis and subtract 1 whenever we find a closed parenthesis. If the counter is never negative, and it holds 0 when we have parsed the whole string, then the parentheses in the string are balanced.

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 $a = a_1 a_2 \ldots a_n$ is balanced if and only if both the following properties hold:

Property 1: $\forall m \in [0,n]$ we have $\sum_{i=1}^{m} f(a_i) \geq 0$
Property 2: $\sum_{i=1}^{n} f(a_i) = 0$

In fact, $\sum_{i=1}^{m} f(a_i)$ counts the difference between the number of open parentheses and closed parentheses that are present in the first m symbols of the string a. Therefore, the first property requires that in any prefix of the string a the number of open parentheses exceeds, or equals 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 the string $a =$ `(()) ()`:

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

Properties 1 and 2 are easy to check for any string and therefore define an useful procedure to decide if a string belongs to our language or not.

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

Theorem 2.1. *For any string of parentheses a of length n*

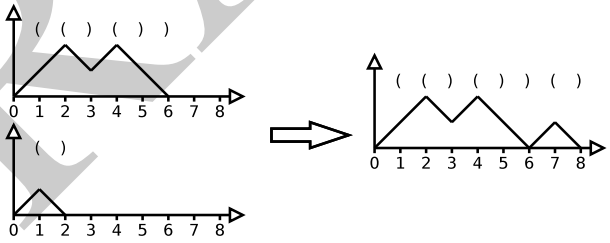
$$a \in L_S \iff \begin{cases} \sum_{i=1}^{m} f(a_i) \geq 0 & m = 0,1...n \\ \sum_{i=1}^{n} f(a_i) = 0 \end{cases}$$

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

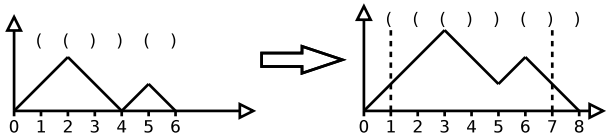
⇒) all the strings produced by the grammar satisfy the two properties;
⇐) any string that satisfy the two properties can be generated by the grammar.

Proof of ⇒) 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. This proof technique is very powerful and will be explained in detail in Chapter 4.
The two properties can be represented graphically over the cartesian plane by taking m over the x-axis and the quantity $\sum_{i=1}^{m} f(a_i)$ over the y-axis. Intuitively, the graph start at 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.

Rule (1): 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).



Rule (2): 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).

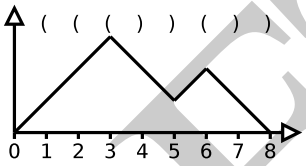


Rule (3): The third rule is just concerned with the empty string that trivially satisfies the two properties.

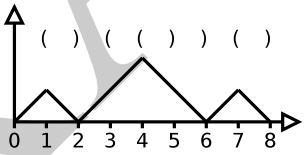
Since we have inspected all the inference rules, the proof of the first implication is concluded.

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

1. If $n = 0$, $a = \varepsilon$. 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 $a_1 \ldots a_k$ and $a_{k+1} \ldots a_n$ 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 alternatives than one we can either explore all of them in parallel (*breadth-first* approach) or try one of them and back-track in case we fail (*depth-first*).

Suppose we want to find a proof for $(\,(\,)\,)\,(\,) \in L_S$. We use the notation

$$(\,(\,)\,)\,(\,) \in L_S \quad \nwarrow$$

1. $(\,(\,)\,)\,()\in L_S \quad\nwarrow\quad \varepsilon\in L_S,\quad (\,(\,)\,)\,()\in L_S$
2. $(\,(\,)\,)\,()\in L_S \quad\nwarrow\quad (\,\in L_S,\quad (\,)\,)\,()\in L_S$
3. $(\,(\,)\,)\,()\in L_S \quad\nwarrow\quad (\,(\,\in L_S,\quad)\,)\,()\in L_S$
4. $(\,(\,)\,)\,()\in L_S \quad\nwarrow\quad (\,(\,)\in L_S,\quad)\,()\in L_S$
5. $(\,(\,)\,)\,()\in L_S \quad\nwarrow\quad (\,(\,)\,)\in L_S,\quad ()\in L_S$
6. $(\,(\,)\,)\,()\in L_S \quad\nwarrow\quad (\,(\,)\,)\,(\in L_S,\quad)\in L_S$
7. $(\,(\,)\,)\,()\in L_S \quad\nwarrow\quad (\,(\,)\,)\,()\in L_S,\quad \varepsilon\in L_S$

Fig. 2.1: Tentative derivations for the goal $(\,(\,)\,)\,()\in L_S$

to mean that we look for a goal-oriented derivation.

- Rule (1) can be applied in many different ways, by splitting the string $(\,(\,)\,)\,()$ in all possible ways. We use the notation

$$(\,(\,)\,)\,()\in L_S \quad\nwarrow\quad \varepsilon\in L_S,\ (\,(\,)\,)\,()\in L_S$$

  to mean that we reduce the proof of $(\,(\,)\,)\,()\in L_S$ to those of $\varepsilon\in L_S$ and $(\,(\,)\,)\,()\in L_S$. Then we have all the alternatives in Figure 2.1 to inspect. 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:

$$(\,(\,)\,)\,()\in L_S \quad\nwarrow\quad (\,)\,)\,(\in L_S$$

- Rule (3) cannot be applied.

We show below a successful derivation, where the empty goal is written $\square$.

$(\,(\,)\,)\,()\in L_S$	$\nwarrow$	$(\,(\,)\,)\in L_S,$	$()\in L_S$	by applying (1)
	$\nwarrow$	$(\,(\,)\,)\in L_S,$	$\varepsilon\in L_S$	by applying (2) to the second goal
	$\nwarrow$	$(\,(\,)\,)\in L_S$		by applying (3) to the second goal
	$\nwarrow$	$()\in L_S$		by applying (2)
	$\nwarrow$	$\varepsilon\in L_S$		by applying (2)
	$\nwarrow$	$\square$		by applying (3)

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: if a derivation exists, then it will be found; if no derivation exists, the strategy may not terminate.

2.3 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). As usual, we denote by Σ_n (respectively Π_n) the subset of function symbols (respectively predicate symbols) with arity n.

Definition 2.7 (Atomic formula). An *atomic formula* consists of a predicate symbol p of arity n applied to n terms with variables.

For example, if $\mathsf{p} \in \Pi_2$, $f \in \Sigma_2$ and $g \in \Sigma_1$, then $\mathsf{p}(\, f(g(x), x)\,,\, g(y)\,)$ is an atomic formula.

Definition 2.8 (Formula). A *formula* is a (possibly empty) conjunction of atomic formulas.

Definition 2.9 (Horn clause). A *Horn clause* is written $l :\!- r$ where l is an atomic formula, called the *head* of the clause, and r is a formula called the *body* of the clause.

Definition 2.10 (Logic program). A logic program is a set of Horn clauses.

The variables appearing in each clause can be instantiated with any term. A goal g is a formula whose validity we want to prove. The goal g can contain variables, which are implicitly existentially quantified.

Unification is used to "match" the head of a clause to an atomic formula of 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.

Suppose we are given a logic program L and a goal $g = a_1, ..., a_n$, where $a_1, ..., a_n$ are atomic formulas. A derivation step $g \nwarrow_{\sigma'} g'$ is obtained by selecting a sub-goal a_i, a clause $l :\!- r \in L$ and a renaming ρ such that:

- $l\rho :\!- r\rho$ is a variant of the clause $l :\!- r \in L$ whose variables are fresh;
- the unification problem $\{a_i \stackrel{?}{=} l\rho\}$ a most general solution σ;
- $\sigma' \stackrel{\text{def}}{=} \sigma_{|vars(a_i)}$;
- $g' \stackrel{\text{def}}{=} a_1, ..., a_{i-1}, r\rho\sigma, a_{i+1}, ..., a_n$.

If we can find a sequence of derivation steps

$$g \nwarrow_{\sigma_1} g_1 \nwarrow_{\sigma_2} g_2 \cdots g_{n-1} \nwarrow_{\sigma_n} \square$$

then we can conclude that the goal g is satisfiable and that the substitution $\sigma \stackrel{\text{def}}{=} \sigma_1 \cdots \sigma_n$ is a least substitutions for the variables in g such that $g\sigma$ is a valid theorem.

Example 2.6 (Sum in Prolog). Let us consider the logic program:

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

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

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

There is no match against the head of the first clause, because 0 does not unify with $s(s(0))$.

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 (i.e., the 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 be added to the goal:

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

If other subgoals were initially present, which may share variables with $\mathsf{sum}(s(s(0)), s(s(0)), v)$ then the substitution should have been applied to them too.

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 (as a subscript of the derivation step) 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:

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

By composing the computed substitutions we get

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

This gives us a proof of the theorem

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

Problems

2.1. Consider the alphabet $\{\mathtt{a},\mathtt{b}\}$ and the grammar

$$\begin{array}{lcl} A & ::= & \mathtt{a}\,A \mid \mathtt{a}\,B \\ B & ::= & \mathtt{b} \mid \mathtt{b}\,B \end{array}$$

1. Describe the form of the strings in the languages L_A and L_B.
2. Define the languages L_A and L_B formally.
3. Write the inference rules that correspond to the productions of the grammar.
4. Write the derivation for the string `a a a b b` both as a proof-tree and as a goal-oriented derivation.
5. Prove that the set of theorems associated with the inference rules coincide with the formal definitions you gave.

2.2. Consider the alphabet $\{\mathtt{0},\mathtt{1}\}$.

1. Give a context free grammar for the set of strings that contain an even number of `0` and `1`.
2. Write the inference rules that correspond to the productions of the grammar.
3. Write the derivation for the string `0 1 1 0 0 0` both as a proof-tree and as a goal-oriented derivation.
4. Prove that your logical systems characterises exactly the set of strings that contain an even number of `0` and `1`.

2.3. Consider the signature Σ such that $\Sigma_0 = \{0\}$, $\Sigma_1 = \{s\}$ and $\Sigma_n = \varnothing$ for any $n \geq 2$.

1. Let $even \in \Pi_1$. What are the theorems of the logical system below?

$$\frac{}{even(0)}\,(1) \qquad \frac{even(x)}{even(s(s(x)))}\,(2)$$

2. Let $odd \in \Pi_1$.What are the theorems of the logical system below?

$$\frac{odd(x)}{odd(s(s(x)))}\,(1)$$

3. Let $leq \in \Pi_2$. What are the theorems of the logical system below?

$$\frac{}{leq(0,x)}\,(1) \qquad \frac{leq(x,y)}{leq(s(x),s(y))}\,(2)$$

2.4. Consider the signature Σ such that $\Sigma_0 = \mathbb{N}$, $\Sigma_2 = \{node\}$ and $\Sigma_n = \varnothing$ otherwise. Let $sum, eq \in \Pi_2$. What are the theorems of the logical system below?

$$\frac{}{sum(n,n)}\,n \in \mathbb{N}\,(1) \qquad \frac{sum(x,n) \quad sum(y,m)}{sum(node(x,y),k)}\,k = n+m\,(2) \qquad \frac{sum(x,n) \quad sum(y,n)}{eq(x,y)}\,(3)$$

2.5. Consider the signature Σ such that $\Sigma_0 = \{0\}$, $\Sigma_1 = \{s\}$ and $\Sigma_n = \varnothing$ for any $n \geq 2$. Give two terms t and t', with $t \neq t'$, such that t is more general than t' and t' is also more general than t.

2.6. Consider the signature Σ such that $\Sigma_0 = \{a\}$, $\Sigma_1 = \{f,g\}$, $\Sigma_2 = \{h,l\}$ and $\Sigma_n = \varnothing$ for any $n \geq 3$. Solve the unification problems below:

1. $G_0 \stackrel{\text{def}}{=} \{x \stackrel{?}{=} f(y), h(z,x) \stackrel{?}{=} h(y,z), g(y) \stackrel{?}{=} g(l(a,a))\}$
2. $G_1 \stackrel{\text{def}}{=} \{x \stackrel{?}{=} f(y), h(z,x) \stackrel{?}{=} h(x,g(z))\}$
3. $G_2 \stackrel{\text{def}}{=} \{x \stackrel{?}{=} f(y), h(z,x) \stackrel{?}{=} h(y,f(z)), l(y,a) \stackrel{?}{=} l(a,z)\}$
4. $G_3 \stackrel{\text{def}}{=} \{x \stackrel{?}{=} f(y), h(y,x) \stackrel{?}{=} h(g(a),f(g(z))), l(z,a) \stackrel{?}{=} l(a,z)\}$

2.7. Extend the logic program for computing the sum with the definition of:

1. a predicate *prod* for computing the product of two numbers;
2. a predicate *pow* for computing the power of a base to an exponent;
3. a predicate *div* that tells if a number can be divided by another number.

2.8. Extend the logic program for computing the sum with the definition of a binary predicate $\mathsf{fib}(N,F)$ that is true if F is the Nth Fibonacci number.

2.9. Suppose that a set of facts of the form $\mathsf{parent}(\mathsf{x},\mathsf{y})$ are given, meaning that x is a parent of y.

1. Define a predicate $\mathsf{brother}(X,Y)$ which holds true iff X and Y have a parent in common.
2. Define a predicate $\mathsf{cousin}(X,Y)$ which holds true iff X and Y are cousins.
3. Define a predicate $\mathsf{ancestor}(X,Y)$ which holds true iff X is an ancestor of Y.
4. If the set of basic facts is:

```
:- parent(alice,bob) .
:- parent(alice,carl) .
:- parent(bob,diana) .
:- parent(bob,ella) .
:- parent(carl,francisco) .
```

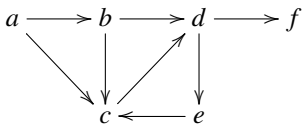
which of the following goals can be derived?

```
?- brother(ella,francisco).
?- brother(ella,diana).
?- cousin(ella,francisco).
?- cousin(ella,diana).
?- ancestor(alice,ella).
?- ancestor(carl,ella).
```

2.10. Suppose that a set of facts of the form $\mathsf{arc}(\mathsf{x},\mathsf{y})$ are given to represent a directed, acyclic graph, meaning that there is an arc from x to y.

1. Define a predicate $\mathsf{path}(X,Y)$ which holds true iff there is a path from X to Y.

2. Suppose the acyclic requirement is violated, like in the graph



defined by

```
:- arc(a,b) .
:- arc(a,c) .
:- arc(b,c) .
:- arc(b,d) .
:- arc(c,d) .
:- arc(d,e) .
:- arc(d,f) .
:- arc(e,c) .
```

Does a goal-oriented derivation for a query, like the one below, necessarily lead to the empty goal? Why?

```
?- path(a,f).
```

2.11. Consider the Horn clauses that correspond to the following statements:

1. All jumping creatures are green.
2. All small jumping creatures are martians.
3. All green martians are intelligent.
4. Ngtrks is small and green.
5. Pgvdrk is a jumping martian.

Who is intelligent?[1]

[1] Taken from `http://www.slideshare.net/SergeiWinitzki/prolog-talk`.

Part II
IMP: a simple imperative language

This part focuses on models for sequential computations that are associated to IMP, a simple imperative language. The syntax and natural semantics of IMP are studied in Chapter 3, while its denotational semantics is presented in Chapter 6, where it is also reconciled with the operational semantics. Chapter 4 explains several induction principles exploited to prove properties of programs and semantics. Chapter 5 fixes the mathematical basis of denotational semantics. The concepts in Chapters 4 and 5 are extensively used in Chapter 6 and in the rest of the monograph.

Chapter 3
Operational Semantics of IMP

Programs must be written for people to read, and only incidentally for machines to execute. (H. Abelson and G. Sussman)

Abstract This chapter introduces the formal syntax and operational semantics of a simple, structured imperative language called IMP, with static variable allocation and no sophisticated declaration constructs for data types, functions, classes, methods and the like. The operational semantics is defined in the natural style and it assumes an abstract machine with a vary basic form of memory to associate integer values to variables. The operational semantics is used to derive a notion of program equivalence and several examples of (in)equivalence proofs are shown.

3.1 Syntax of IMP

The IMP programming language is a simple imperative language (e.g., it can be seen as a bare bone version of the C language) with only three data types:

int: the set of integer numbers, ranged over by metavariables $m, n, \ldots$

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

bool: the set of boolean values, ranged over by metavariables $u, v, \ldots$

$$\mathbb{B} = \{\textbf{true}, \textbf{false}\}$$

locations: the (denumerable) set of memory locations (we consider programs that use a finite number of locations and we assume there are enough locations available for any program), ranged over by metavariables $x, y, \ldots$

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

The grammar for IMP comprises three syntactic categories:

Aexp: Arithmetic expressions, ranged over by $a, a', \ldots$
Bexp: Boolean expressions, ranged over by $b, b', \ldots$

Com : Commands, ranged over by $c, c', ...$

Definition 3.1 (IMP: syntax). The following productions define the syntax of IMP:

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

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

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

3.1.1 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 (e.g., $7/0$) or give different values (e.g., $0/0$) so that its use would introduce unnecessary complexity.

3.1.2 Boolean Expressions

A boolean expression can be a logical value v, or the equality of an arithmetic expression with another, an arithmetic expression less or equal than another one, a negation, a logical conjunction or disjunction.

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

3.1.3 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 expression b, if it is true we execute c_0 and if it is false we execute c_1. Finally we have a **while** statement, which is a command that keeps executing c until b becomes false.

3.1.4 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 3.1 (Valid expressions).

$(\textbf{while } b \textbf{ do } c_1)\ ;\ c_2$	is a valid command;
$\textbf{while } b \textbf{ do } (c_1\ ;\ c_2)$	is a valid command;
$\textbf{while } b \textbf{ do } c_1\ ;\ c_2$	is not a valid command, because it is ambiguous.

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

Example 3.2 (Validity check). Let us consider the formula:

$$\textbf{if } (x = 0) \textbf{ then } (\textbf{skip}) \textbf{ else } (x := (x - 1)) \in Com$$

We can prove its validity by the following (deterministic) derivation, where we write $\nwarrow^*$ to mean that several derivation steps are grouped into one for brevity:

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

3.2 Operational Semantics of IMP

3.2.1 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 stores. To this aim, we introduce the notion of *memory*:

$$\sigma \;\in\; \Sigma = (\mathbf{Loc} \to \mathbb{Z})$$

A memory σ is an element of the set Σ which contains all the functions from locations to integer numbers. A particular σ is just a function from locations to integer numbers so it 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 such that the location x contains the value 5 and the location y the value 10 and elsewhere is stored 0, we write:

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

In this way we can represent any interesting memory by a finite set of pairs.

We let $()$ denote the memory such that all locations are assigned the value 0.

Definition 3.2 (Memory update). 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 $\sigma[^{n}/_{x}][^{m}/_{x}] = \sigma[^{m}/_{x}]$. In fact:

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

Moreover, when $x \neq y$, then the order of updates is not important, i.e., $\sigma[^n/_x][^m/_y] = \sigma[^m/_y][^n/_x]$. For this reason, we often use the more compact notation $\sigma[^n/_x, ^m/_y]$.

3.2.2 Inference Rules

Now we are going to give the *operational semantics* to IMP using a logical system. It is called "big-step" semantics (see Section 1.2) because it leads to the result in one single proof.

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

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

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

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

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

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

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

Next we show each inference rule and comment on it.

3.2.2.1 Inference Rules for Arithmetic Expressions

We start with the rules about arithmetic expressions.

$$\frac{}{\langle n, \sigma \rangle \to n}\ (\text{num}) \tag{3.1}$$

The axiom 3.1 (num) 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) no matter which σ.

$$\frac{}{\langle x, \sigma \rangle \to \sigma(x)} \text{(ide)} \tag{3.2}$$

The axiom 3.2 (ide) is also quite intuitive: the evaluation of an identifier x in the memory σ results in the value stored in x.

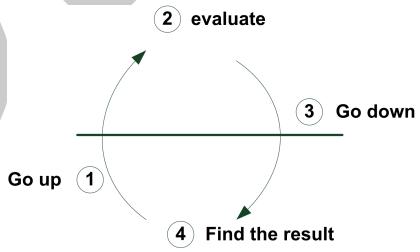
$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n} \, n = n_0 + n_1 \, \text{(sum)} \tag{3.3}$$

The rule 3.3 (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. Note that we exploit the side condition $n = n_0 + n_1$ to indicate the relation between the target n of the conclusion and the targets of the premises. We present an equivalent, but more compact, version of the rule (sum), where the target of the conclusion is obtained as the sum of the targets of the premises. In the following we shall adopt the second format (3.4).

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n_0 + n_1} \, \text{(sum)} \tag{3.4}$$

We remark the difference between the two occurrences of the symbol $+$ in the rule: in the source of the conclusion (i.e., $a_0 + a_1$) it denotes a piece of syntax, in the target of the conclusion (i.e., $n_0 + n_1$) 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 compose the results and finally go down again to draw the conclusion:



In this case we suppose we want to evaluate, in the 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 to get n_1 and then the final result will be $n_0 + n_1$. Note that the same memory σ is duplicated and distributed to the two evaluations of a_0 and a_1, which may occur independently in any order.

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

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

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 - a_1, \sigma\rangle \to n_0 - n_1}\ (\text{dif}) \tag{3.5}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \times a_1, \sigma\rangle \to n_0 \times n_1}\ (\text{prod}) \tag{3.6}$$

3.2.2.2 Inference Rules for Boolean Expressions

The rules for boolean expressions are also similar to the previous ones and need no particular comment, except for noting that the premises of rules (equ) and (leq) refer the judgements of arithmetic expressions.

$$\frac{}{\langle v, \sigma\rangle \to v}\ (\text{bool}) \tag{3.7}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 = a_1, \sigma\rangle \to (n_0 = n_1)}\ (\text{equ}) \tag{3.8}$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \leq a_1, \sigma\rangle \to (n_0 \leq n_1)}\ (\text{leq}) \tag{3.9}$$

$$\frac{\langle b, \sigma\rangle \to v}{\langle \neg b, \sigma\rangle \to \neg v}\ (\text{not}) \tag{3.10}$$

$$\frac{\langle b_0, \sigma\rangle \to v_0 \quad \langle b_1, \sigma\rangle \to v_1}{\langle b_0 \vee b_1, \sigma\rangle \to (v_0 \vee v_1)}\ (\text{or}) \tag{3.11}$$

$$\frac{\langle b_0, \sigma\rangle \to v_0 \quad \langle b_1, \sigma\rangle \to v_1}{\langle b_0 \wedge b_1, \sigma\rangle \to (v_0 \wedge v_1)}\ (\text{and}) \tag{3.12}$$

3.2.2.3 Inference Rules for Commands

Next, we move to the inference rules for commands.

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma}\ (\text{skip}) \tag{3.13}$$

The rule 3.13 (skip) is very simple: it leaves the memory σ unchanged.

$$\frac{\langle a, \sigma\rangle \to m}{\langle x := a, \sigma\rangle \to \sigma[{}^{m}/_{x}]}\ (\text{assign}) \tag{3.14}$$

The rule 3.14 (assign) 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 $\sigma(x)$). Note that the premise refers to the judgements of arithmetic expressions.

$$\frac{\langle c_0, \sigma\rangle \to \sigma'' \quad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c_0; c_1, \sigma\rangle \to \sigma'}\ (\text{seq}) \tag{3.15}$$

The rule 3.15 (seq) for the sequential composition (concatenation) of commands is quite interesting. We start by evaluating the first command c_0 in the memory σ. As a result we get an updated memory σ'' which we use for evaluating the second command c_1. In fact the order of evaluation of the two command is important and it would not make sense to evaluate c_1 in the original memory σ, because the effects of executing c_0 would be lost. Finally, the memory σ' obtained by evaluating c_1 in σ'' is returned as the result of evaluating $c_0; c_1$ in σ.

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

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c_0, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}\ (\text{ifft}) \tag{3.16}$$

$$\frac{\langle b,\sigma\rangle \to \textbf{false} \quad \langle c_1,\sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\sigma\rangle \to \sigma'}\ (\text{ifff}) \tag{3.17}$$

The rule 3.16 (iftt) 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 3.17 (ifff) checks that b evaluated to false and then returns as result the memory σ' obtained by evaluating the command c_1 in σ.

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

$$\frac{\langle b,\sigma\rangle \to \textbf{true} \quad \langle c,\sigma\rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c,\sigma''\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c,\sigma\rangle \to \sigma'}\ (\text{whtt}) \tag{3.18}$$

$$\frac{\langle b,\sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c,\sigma\rangle \to \sigma}\ (\text{whff}) \tag{3.19}$$

The rule 3.18 (whtt) 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 3.19 (whff) applies to the case where the guard evaluates to false: then the cycle terminates and the memory σ is returned unchanged.

Remark 3.1. There is an important difference between the rule 3.18 and all the other inference rules we have encountered so far. All the other rules take 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 3.18 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:

$$\to \subseteq (\mathit{Aexp} \times \Sigma \times \mathbb{Z}) \cup (\mathit{Bexp} \times \Sigma \times \mathbb{B}) \cup (\mathit{Com} \times \Sigma \times \Sigma)$$

We will see later that the computation is deterministic, in the sense that given any expression / commands and any memory as input there is at most one result (exactly one in case of arithmetic and boolean expressions).

3.2.3 Examples

Example 3.3 (Semantic evaluation of a command). Let us consider the (extra-bracketed) command

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

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

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

or exploiting the usual convention for indented code, as:

$$\begin{array}{l} c \stackrel{\text{def}}{=} x := 0\ ; \\ \quad \textbf{while}\ 0 \le y\ \textbf{do}\ (\\ \qquad x := x + (2 \times y) + 1\ ; \\ \qquad y := y - 1 \\ \quad) \end{array}$$

Without too much difficulties, the experienced reader can guess the relation between the value of y at the beginning of the execution and that of x at the end of the execution: The program computes the square of (the value initially stored in) y plus 1 (when $y \ge 0$) and stores it in x. In fact, by exploiting the well-known equalities $0^2 = 0$ and $(n+1)^2 = n^2 + 2n + 1$, the value of $(y+1)^2$ is computed as the sum of the first $y+1$ odd numbers $\sum_{i=0}^{y}(2i+1)$. For example, for $y = 3$ we have $4^2 = 1+3+5+7 = 16$.

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

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

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

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

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 $\langle c, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma$, with σ unknown, 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 $\left({}^{27}/_{x}, {}^{2}/_{y}\right)$ and the command c to be executed, 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 at the end of the computation.

To convince the reader that the notation for goal-oriented derivations introduced in Section 2.3 is more effective than the tree-like notation, we first 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 rapidly large, we split the derivation in smaller pieces that are proved separately. We use "?" to mark the missing parts of the derivations.

$$\dfrac{\dfrac{\dfrac{}{\langle 0, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to 0}\,\text{num}}{\langle x := 0, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \left({}^{27}/_{x}, {}^{2}/_{y}\right)\left[{}^{0}/_{x}\right] = \sigma_1}\,\text{assign} \qquad \dfrac{?}{\langle c_1, \sigma_1\rangle \to \sigma}}{\langle c, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma}\,\text{seq}$$

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 $\sigma_1 = \left({}^{27}/_{x}, {}^{2}/_{y}\right)\left[{}^{0}/_{x}\right] = \left({}^{0}/_{x}, {}^{2}/_{y}\right)$.

$$\dfrac{\dfrac{\dfrac{}{\langle 0, \sigma_1\rangle \to 0}\,\text{num} \quad \dfrac{}{\langle y, \sigma_1\rangle \to \sigma_1(y) = 2}\,\text{ide}}{\langle 0 \leq y, \sigma_1\rangle \to (0 \leq 2) = \mathbf{true}}\,\text{leq} \qquad \dfrac{?}{\langle c_2, \sigma_1\rangle \to \sigma_2} \qquad \dfrac{?}{\langle c_1, \sigma_2\rangle \to \sigma}}{\langle c_1, \sigma_1\rangle \to \sigma}\,\text{whtt}$$

Next we need to prove the goals $\langle c_2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma_2$ and $\langle c_1, \sigma_2\rangle \to \sigma$. Let us focus on $\langle c_2, \sigma_1\rangle \to \sigma_2$ first:

$$\dfrac{\dfrac{\dfrac{?}{\langle a_1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m'} \quad \dfrac{}{\langle 1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to 1}\,\text{num}}{\langle a, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m = m'+1}\,\text{sum}}{\dfrac{\langle c_3, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \left({}^{0}/_{x}, {}^{2}/_{y}\right)\left[{}^{m}/_{x}\right] = \sigma_3}{\langle c_2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma_2}\,\text{seq}}\,\text{assign} \quad \dfrac{\dfrac{?}{\langle y-1, \sigma_3\rangle \to m''}}{\langle c_4, \sigma_3\rangle \to \sigma_3\left[{}^{m''}/_{y}\right] = \sigma_2}\,\text{assign}$$

We show separately the details for the pending derivations of $\langle a_1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m'$ and $\langle y-1, \sigma_3\rangle \to m''$:

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

Since $m' = 4$, then it means that $m = m'+1 = 5$ and hence $\sigma_3 = \left({}^{0}/_{x}, {}^{2}/_{y}\right)\left[{}^{5}/_{x}\right] = \left({}^{5}/_{x}, {}^{2}/_{y}\right)$.

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

Since $m'' = 1$ we know that $\sigma_2 = \left({}^{5}/_{x}, {}^{2}/_{y}\right)\left[{}^{m''}/_{y}\right] = \left({}^{5}/_{x}, {}^{2}/_{y}\right)\left[{}^{1}/_{y}\right] = \left({}^{5}/_{x}, {}^{1}/_{y}\right)$.

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

$$\dfrac{\dfrac{\vdots}{\langle 0 \le y, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \textbf{true}}\,\text{leq} \quad \dfrac{\vdots}{\langle c_2, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \left({}^{8}/_{x}, {}^{0}/_{y}\right) = \sigma_4}\,\text{seq} \quad \dfrac{?}{\langle c_1, \sigma_4\rangle \to \sigma}}{\langle c_1, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \sigma}\,\text{whtt}$$

Hence $\sigma_4 = \left({}^{8}/_{x}, {}^{0}/_{y}\right)$ and next we prove $\langle c_1, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \sigma$.

$$\dfrac{\dfrac{\vdots}{\langle 0 \le y, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \textbf{true}}\,\text{leq} \quad \dfrac{\vdots}{\langle c_2, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \left({}^{9}/_{x}, {}^{-1}/_{y}\right) = \sigma_5}\,\text{seq} \quad \dfrac{?}{\langle c_1, \sigma_5\rangle \to \sigma}}{\langle c_1, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \sigma}\,\text{whtt}$$

Hence $\sigma_5 = \left({}^{9}/_{x}, {}^{-1}/_{y}\right)$. Finally:

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

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 (written $\nwarrow^{*}$) omitting trivial steps.

$$\begin{aligned}
\langle c, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma \quad &\nwarrow \langle x := 0, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma_1,\ \langle c_1, \sigma_1\rangle \to \sigma \\
&\nwarrow_{\sigma_1 = \left({}^{27}/_{x}, {}^{2}/_{y}\right)[{}^{n}/_{x}]} \langle 0, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to n,\ \langle c_1, \left({}^{27}/_{x}, {}^{2}/_{y}\right)[{}^{n}/_{x}]\rangle \to \sigma \\
&\nwarrow_{n=0,\ \sigma_1 = \left({}^{0}/_{x}, {}^{2}/_{y}\right)} \langle c_1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma \\
&\nwarrow \langle 0 \le y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \mathbf{true}, \\
&\qquad \langle c_2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma_2,\ \langle c_1, \sigma_2\rangle \to \sigma \\
&\nwarrow \langle 0, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to n_1,\ \langle y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to n_2, \\
&\qquad n_1 \le n_2,\ \langle c_2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma_2,\ \langle c_1, \sigma_2\rangle \to \sigma \\
&\nwarrow^{*}_{n_1=0,\ n_2=2} \langle c_3, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma_3,\ \langle c_4, \sigma_3\rangle \to \sigma_2, \\
&\qquad \langle c_1, \sigma_2\rangle \to \sigma \\
&\nwarrow_{\sigma_3 = \left({}^{0}/_{x}, {}^{2}/_{y}\right)[{}^{m}/_{x}]} \langle x + (2 \times y) + 1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m, \\
&\qquad \langle c_4, \left({}^{0}/_{x}, {}^{2}/_{y}\right)[{}^{m}/_{x}]\rangle \to \sigma_2,\ \langle c_1, \sigma_2\rangle \to \sigma \\
&\nwarrow^{*}_{m = 0+(2\times 2)+1 = 5,\ \sigma_3 = \left({}^{5}/_{x}, {}^{2}/_{y}\right)} \langle c_4, \left({}^{5}/_{x}, {}^{2}/_{y}\right)\rangle \to \sigma_2,\ \langle c_1, \sigma_2\rangle \to \sigma \\
&\nwarrow^{*}_{\sigma_2 = \left({}^{5}/_{x}, {}^{2}/_{y}\right)[{}^{1}/_{y}] = \left({}^{5}/_{x}, {}^{1}/_{y}\right)} \langle c_1, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \sigma \\
&\nwarrow^{*}_{\sigma_4 = \left({}^{5}/_{x}, {}^{1}/_{y}\right)[{}^{8}/_{x}][{}^{0}/_{y}] = \left({}^{8}/_{x}, {}^{0}/_{y}\right)} \langle c_1, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \sigma \\
&\nwarrow^{*}_{\sigma_5 = \left({}^{8}/_{x}, {}^{0}/_{y}\right)[{}^{9}/_{x}][{}^{-1}/_{y}] = \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 \le y, \left({}^{9}/_{x}, {}^{-1}/_{y}\right)\rangle \to \mathbf{false} \\
&\nwarrow^{*} \ \square
\end{aligned}$$

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:

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

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

Example 3.4 (Non termination). Let us consider the command

$$c \stackrel{\text{def}}{=} \textbf{while true do skip}$$

Given σ, the only possible derivation goes as follows:

$$\begin{array}{rl} \langle c,\sigma\rangle \to \sigma' & \nwarrow \ \langle \textbf{true},\sigma\rangle \to \textbf{true},\ \langle \textbf{skip},\sigma\rangle \to \sigma_1,\ \langle c,\sigma_1\rangle \to \sigma' \\ & \nwarrow \ \langle \textbf{skip},\sigma\rangle \to \sigma_1,\ \langle c,\sigma_1\rangle \to \sigma' \\ & \nwarrow_{\sigma_1=\sigma} \ \langle c,\sigma\rangle \to \sigma' \end{array}$$

After a few steps of derivation we reach the same goal from which we started and there are no alternatives to try!

In this case, we can prove that $\langle c,\sigma\rangle \not\to$. 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:

$$\begin{array}{rl} \langle c,\sigma\rangle \to \sigma' & \nwarrow \ \langle \textbf{true},\sigma\rangle \to \textbf{true},\ \langle \textbf{skip},\sigma\rangle \to \sigma_1,\ \langle c,\sigma_1\rangle \to \sigma' \\ & \vdots \\ (*) & \nwarrow \ \langle c,\sigma\rangle \to \sigma' \\ & \vdots \\ & \nwarrow \ \Box \end{array}$$

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

3.3 Abstract Semantics: Equivalence of 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"? The equivalence between two

commands is an important issue because it allows, e.g., to replace a program with an equivalent but more efficient one. Informally, two programs are equivalent if they *behave in the same way*. But can we make this idea more precise?

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 a certain memory, like **while** $x \neq 0$ **do** $x := x - 1$ when evaluated in a store σ such that $\sigma(x) < 0$. 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 3.3 (Equivalence of expressions and commands). We say that the arithmetic expressions a_1 and a_2 are *equivalent*, written $a_1 \sim a_2$ if and only if for any memory σ they evaluate in the same way. Formally:

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

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

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

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

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

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

3.3.1 Examples: Simple Equivalence Proofs

The first example we show is concerned with fully specified programs that operate on unspecified memories.

Example 3.5 (Equivalent commands). Let us try to prove that the following two commands are equivalent:

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

It is immediate to prove that

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

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

The evaluation of c_1 in σ depends on $\sigma(x)$: if $\sigma(x) = 0$ we must apply the rule 3.19 (whff), otherwise the rule 3.18 (whtt) must be applied. Since we do not know the value of $\sigma(x)$, we consider the two cases separately. The corresponding hypotheses are called *path conditions* and outline a very important technique for the symbolic analysis of programs.

Case $\sigma(x) \neq 0$) Let us inspect a possible derivation for $\langle c_1, \sigma \rangle \to \sigma'$. Since $\sigma(x) \neq 0$ we select the rule (whtt) at the first step:

$$\begin{array}{rl}
\langle c_1, \sigma \rangle \to \sigma' \quad \nwarrow & \langle x \neq 0, \sigma \rangle \to \mathbf{true}, \quad \langle x := 0, \sigma \rangle \to \sigma_1, \\
& \langle c_1, \sigma_1 \rangle \to \sigma' \\
\nwarrow^*_{\sigma_1 = \sigma[^0/_x]} & \langle c_1, \sigma\,[^0/_x] \rangle \to \sigma' \\
\nwarrow_{\sigma' = \sigma[^0/_x]} & \langle x \neq 0, \sigma\,[^0/_x] \rangle \to \mathbf{false} \\
\nwarrow^* & \sigma\,[^0/_x]\,(x) = 0 \\
\nwarrow & \Box
\end{array}$$

Case $\sigma(x) = 0$) Let us inspect a derivation for $\langle c_1, \sigma \rangle \to \sigma'$. Since $\sigma(x) = 0$ we select the rule (whff) at the first step:

$$\begin{array}{rl}
\langle c_1, \sigma \rangle \to \sigma' \quad \nwarrow_{\sigma' = \sigma} & \langle x \neq 0, \sigma \rangle \to \mathbf{false} \\
\nwarrow^* & \sigma(x) = 0 \\
\nwarrow & \Box
\end{array}$$

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 compare the results.

3.3.2 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 3.6 (Parametric proofs (1)). Let us consider the commands:

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

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

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 3.19 (whff), or the rule 3.18 (whtt). Which rule to use depends on the evaluation of b. Since we do not know what b is, we must take into account both possibilities and consider the two cases separately.

$\langle b, \sigma \rangle \to \textbf{false}$) For c_1 we have:

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

For c_2 we have:

$$\langle \textbf{if } b \textbf{ then } (c; c_1) \textbf{ else skip}, \sigma \rangle \to \sigma' \quad \nwarrow \quad \langle b, \sigma \rangle \to \textbf{false}, \quad \langle \textbf{skip}, \sigma \rangle \to \sigma'$$
$$\nwarrow^{*}_{\sigma'=\sigma} \ \square$$

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

$\langle b, \sigma \rangle \to \textbf{true}$) For c_1 we have:

$$\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \to \sigma' \nwarrow \langle b, \sigma \rangle \to \textbf{true}, \quad \langle c, \sigma \rangle \to \sigma_1, \quad \langle c_1, \sigma_1 \rangle \to \sigma'$$
$$\nwarrow \langle c, \sigma \rangle \to \sigma_1, \quad \langle c_1, \sigma_1 \rangle \to \sigma'$$

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

$$\begin{array}{rl} \langle \textbf{if } b \textbf{ then } (c;c_1) \textbf{ else skip}, \sigma\rangle \to \sigma' \nwarrow & \langle b, \sigma\rangle \to \textbf{true}, \\ & \langle c; c_1, \sigma\rangle \to \sigma' \\ \nwarrow & \langle c; c_1, \sigma\rangle \to \sigma' \\ \nwarrow & \langle c, \sigma\rangle \to \sigma_1, \\ & \langle c_1, \sigma_1\rangle \to \sigma' \end{array}$$

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

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

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

Example 3.7 (Parametric proofs (2)). Let us consider the commands:

$$\begin{array}{l} c_1 \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } c \\ c_2 \stackrel{\text{def}}{=} \textbf{if } b \textbf{ then } c_1 \textbf{ else skip} \end{array}$$

Is it true that $\forall b \in \mathit{Bexp}, c \in \mathit{Com}.\ c_1 \sim c_2$?

We have already examined the different derivations for c_1 in the previous example. Moreover, the evaluation of c_2 when $\langle b, \sigma\rangle \to \textbf{false}$ is also analogous to that of the command c_2 in Example 3.6. Therefore we focus on the analysis of c_2 for the case $\langle b, \sigma\rangle \to \textbf{true}$. Trivially:

$$\begin{array}{rl} \langle \textbf{if } b \textbf{ then } c_1 \textbf{ else skip}, \sigma\rangle \to \sigma' \nwarrow & \langle b, \sigma\rangle \to \textbf{true}, \quad \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma' \\ \nwarrow & \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma' \end{array}$$

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

3.3.3 Examples: Inequality Proofs

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

Example 3.8 (Inequality proof). Let us consider the commands:

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

Let us prove that $c_1 \not\sim c_2$.
For c_2 we have

$$\langle x := 0, \sigma\rangle \to \sigma' \;\nwarrow_{\sigma' = \sigma[^n/_x]}\; \langle 0, \sigma\rangle \to n \;\nwarrow_{n=0}\; \square$$

That is: $\forall \sigma.\ \langle x := 0, \sigma\rangle \to \sigma[^0/_x]$.
Next, we focus on the first part of c_1

$$w \stackrel{\text{def}}{=} \textbf{while } x > 0 \textbf{ do } x := 1$$

If $\sigma(x) \leq 0$ it is immediate to check that

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

The derivation is sketched below:

$$\begin{array}{rl} \langle w, \sigma\rangle \to \sigma' \;\nwarrow_{\sigma' = \sigma} & \langle x > 0, \sigma\rangle \to \textbf{false} \\ \nwarrow & \langle x, \sigma\rangle \to n,\ \langle 0, \sigma\rangle \to m,\ n \leq m \\ \nwarrow_{n = \sigma(x)} & \langle 0, \sigma\rangle \to m,\ \sigma(x) \leq m \\ \nwarrow_{m=0} & \sigma(x) \leq 0 \\ \nwarrow & \square \end{array}$$

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

$$\begin{array}{rl} \langle w, \sigma\rangle \to \sigma' \quad \nwarrow & \langle x > 0, \sigma\rangle \to \textbf{true},\ \langle x := 1, \sigma\rangle \to \sigma'',\ \langle w, \sigma''\rangle \to \sigma' \\ \nwarrow^* & \langle x := 1, \sigma\rangle \to \sigma'',\ \langle w, \sigma''\rangle \to \sigma' \\ \nwarrow^*_{\sigma'' = \sigma[^1/_x]} & \langle w, \sigma[^1/_x]\rangle \to \sigma' \end{array}$$

Let us continue the derivation for $\langle w, \sigma[^1/_x]\rangle \to \sigma'$:

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

Now, note that we got the same subgoal $\langle w, \sigma[^1/_x]\rangle \to \sigma'$ already inspected: hence it is not possible to conclude the derivation, which will loop.

Summing up all the above we conclude that:

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

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).

$$\begin{array}{rl}\langle w; x:=0, \sigma\rangle \to \sigma' \nwarrow & \langle w, \sigma\rangle \to \sigma'',\ \langle x:=0, \sigma''\rangle \to \sigma' \\ \nwarrow^*_{\sigma''=\sigma} & \langle x:=0, \sigma\rangle \to \sigma' \\ \nwarrow^*_{\sigma'=\sigma[^0/_x]} & \Box\end{array}$$

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:

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

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 $c_1 \not\sim c_2$.

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

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

3.3.4 Examples: Diverging Computations

What does it happen if the program has infinitely many different looping situations? How should we handle the memories for which this happens?

Let us rephrase the definition of equivalence between commands:

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

Next we see an example where this situation emerges.

Example 3.9 (Proofs of non-termination). Let us consider the commands:

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

Is it true that $c_1 \sim c_2$? On the one hand, note that c_1 can only store 1 in x, whereas c_2 can keep incrementing the value stored in x, so one may be 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 $x > 0$:

Case $\sigma(x) \leq 0$) If $\sigma(x) \leq 0$, we know already from Example 3.8 that $\langle c_1, \sigma \rangle \to \sigma$:

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

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

Case $\sigma(x) > 0$) If $\sigma(x) > 0$, we know already from Example 3.8 that $\langle c_1, \sigma \rangle \not\to$
Now we must check if c_2 diverges too when $\sigma(x) > 0$:

$$
\begin{array}{rl}
\langle c_2,\sigma\rangle \to \sigma' & \nwarrow \ \langle x>0,\sigma\rangle \to \textbf{true},\\
& \quad \langle x:=x+1,\sigma\rangle \to \sigma_1, \quad \langle c_2,\sigma_1\rangle \to \sigma'\\
& \nwarrow^{*} \ \langle x:=x+1,\sigma\rangle \to \sigma_1, \quad \langle c_2,\sigma_1\rangle \to \sigma'\\
& \nwarrow^{*}_{\sigma_1=\sigma\left[{}^{\sigma(x)+1}/_{x}\right]} \ \langle c_2,\sigma\left[{}^{\sigma(x)+1}/_{x}\right]\rangle \to \sigma'\\
& \nwarrow \ \langle x>0,\sigma\left[{}^{\sigma(x)+1}/_{x}\right]\rangle \to \textbf{true},\\
& \quad \langle x:=x+1,\sigma\left[{}^{\sigma(x)+1}/_{x}\right]\rangle \to \sigma_2,\\
& \quad \langle c_2,\sigma_2\rangle \to \sigma'\\
& \nwarrow^{*} \ \langle x:=x+1,\sigma\left[{}^{\sigma(x)+1}/_{x}\right]\rangle \to \sigma_2,\\
& \quad \langle c_2,\sigma_2\rangle \to \sigma'\\
\nwarrow^{*}_{\sigma_2=\sigma_1\left[{}^{\sigma_1(x)+1}/_{x}\right]=\sigma\left[{}^{\sigma(x)+2}/_{x}\right]} & \ \langle c_2,\sigma\left[{}^{\sigma(x)+2}/_{x}\right]\rangle \to \sigma'\\
& \ \ldots
\end{array}
$$

Now the situation is more subtle: we keep looping, but without crossing the same subgoal twice, because the memory is updated with different values for x at each iteration. However, using induction, that will be the subject of Section 4.1.3, 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 ith iteration the values stored in the memory at location x will be $\sigma(x)+i$, from which we can conclude that the expression $x>0$ will hold true (since by assumption $\sigma(x)>0$ and thus $\sigma(x)+i>0$). Once the proof is completed, we can conclude that c_2 diverges and therefore $c_1 \sim c_2$. Below we outline a simpler technique to prove non-termination, that can be used under some circumstances.

Let us consider the command $w \stackrel{\text{def}}{=} \textbf{while}\ b\ \textbf{do}\ c$. As we have seen in the last example, to prove the non-termination of w we can exploit the induction hypotheses over memory states to define the inference rule below: the idea is that 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 σ'' which is also in S, then we can conclude that w diverges when evaluated in any of the memories $\sigma \in S$.

$$
\frac{\sigma \in S \quad \forall \sigma' \in S.\langle b,\sigma'\rangle \to \textbf{true} \quad \forall \sigma' \in S, \forall \sigma''.(\langle c,\sigma'\rangle \to \sigma'' \ \Rightarrow \ \sigma'' \in S)}{\langle w,\sigma\rangle \not\to} \tag{3.20}
$$

Note that the property

$$\forall \sigma''.(\langle c, \sigma' \rangle \to \sigma'' \;\Rightarrow\; \sigma'' \in S)$$

is satisfied even when $\langle c, \sigma' \rangle \not\to$, because there is no σ'' such that the left-hand side of the implication holds.

Example 3.10. Let us consider again the command c_2 from Example 3.9:

$$c_2 \stackrel{\text{def}}{=} \textbf{while } x > 0 \textbf{ do } x := x + 1.$$

We set $S = \{\sigma \mid \sigma(x) > 0\}$, take $\sigma \in S$ and prove the premises of the rule for divergence to conclude that $\langle w, \sigma \rangle \not\to$.

1. We must show that $\forall \sigma' \in S.\langle x > 0, \sigma' \rangle \to \textbf{true}$, which follows by definition of S.
2. We need to prove that $\forall \sigma' \in S, \forall \sigma''.(\langle x := x + 1, \sigma' \rangle \to \sigma'' \;\Rightarrow\; \sigma'' \in S)$. Take $\sigma' \in S$, i.e., such that $\sigma'(x) > 0$, and assume $\langle x := x + 1, \sigma' \rangle \to \sigma''$. Then it must be $\sigma'' = \sigma'[{}^{\sigma'(x)+1}/_{x}]$ and we have $\sigma''(x) = \sigma'[{}^{\sigma'(x)+1}/_{x}](x) = \sigma'(x) + 1 > 0$ because $\sigma'(x) > 0$ by hypothesis. Hence $\sigma'' \in S$.

Remind that, in general, program termination is semi-decidable (and non-termination possibly non semi-decidable), so we cannot have a proof technique for demonstrating the convergence or divergence of any program.

Example 3.11 (Collatz's algorithm). Consider the algorithm below, which is known as *Collatz's algorithm*, or also as *Half Or Triple Plus One*

$$d \stackrel{\text{def}}{=} x := y \;;\; k := 0 \;;\; \textbf{while } x > 0 \textbf{ do } (x := x - 2 \;;\; k := k + 1)$$
$$c \stackrel{\text{def}}{=} \textbf{while } y \neq 1 \textbf{ do } (d \;;\; \textbf{if } x = 0 \textbf{ then } y := k \textbf{ else } y := (3 \times y) + 1)$$

The command d, when executed in a memory σ with $\sigma(y) > 0$, terminates by producing either a memory σ' with $\sigma'(x) = 0$ and $\sigma(y) = 2 \times \sigma'(k)$ (when $\sigma(y)$ is even), or a memory σ'' with $\sigma''(x) = -1$ (when $\sigma(y)$ is odd). The command c exploits d to update at each iteration the value of y to either the half of y (when $\sigma(y)$ is even) or three times y plus one (when $\sigma(y)$ is odd). It is an open mathematical conjecture to prove that the command c terminates when executed in any memory σ with $\sigma(y) > 0$. The conjecture has been checked by computers and proved true[1] for all starting values of y up to 5×2^{60}.

Problems

3.1. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } y > 0 \textbf{ do } (r := r \times x \;;\; y := y - 1)$$

[1] Source `http://en.wikipedia.org/wiki/Collatz_conjecture`, last visited July 2015.

Let $c \stackrel{\text{def}}{=} (r := 1 \ ; \ w)$ and $\sigma \stackrel{\text{def}}{=} [{}^{9}/_{x}, {}^{2}/_{y}]$. Use goal-oriented derivation, according to the operational semantics of IMP, to find the memory σ' such that $\langle c, \sigma \rangle \to \sigma'$, if it exists.

3.2. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } y \geq 0 \textbf{ do if } y = 0 \textbf{ then } y := y + 1 \textbf{ else skip}$$

For which memories σ, σ' do we have $\langle w, \sigma \rangle \to \sigma'$?

3.3. Prove that for any $b \in \mathit{Bexp}, c \in \mathit{Com}$ we have $c \sim \textbf{if } b \textbf{ then } c \textbf{ else } c$.

3.4. Prove that for any $b \in \mathit{Bexp}, c \in \mathit{Com}$ we have $c_1 \sim c_2$, where:

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

3.5. Prove that for any $b \in \mathit{Bexp}, c \in \mathit{Com}$ we have $c_1 \sim c_2$, where:

$$c_1 \stackrel{\text{def}}{=} c \ ; \ \textbf{while } b \textbf{ do } c$$
$$c_2 \stackrel{\text{def}}{=} (\textbf{while } b \textbf{ do } c) \ ; \ c$$

3.6. Prove that $c_1 \not\sim c_2$, where:

$$c_1 \stackrel{\text{def}}{=} \textbf{while } x > 0 \textbf{ do } x := 0$$
$$c_2 \stackrel{\text{def}}{=} \textbf{while } x \geq 0 \textbf{ do } x := 0$$

3.7. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } x \leq y \textbf{ do } (x := x + 1 \ ; \ y := y + 2)$$

Find the largest set S of memories such that the command w diverges. Use the inference rule for divergence to prove non-termination.

3.8. Prove that $c_1 \not\sim c_2$, where:

$$c_1 \stackrel{\text{def}}{=} \textbf{while } x > 0 \textbf{ do } x := x + 1$$
$$c_2 \stackrel{\text{def}}{=} \textbf{while } x \geq 0 \textbf{ do } x := x + 2$$

3.9. Suppose we extend IMP with the arithmetic expression a_0/a_1 for integer division, whose operational semantics is:

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0/a_1, \sigma \rangle \to n} \ n_0 = n_1 \times n \ (\text{div}) \tag{3.21}$$

1. Prove that the semantics of extended arithmetic expressions is not deterministic. In other words, give a counterexample to the property below:

$$\forall a \in Aexp, \forall \sigma \in \Sigma, \forall n, m \in \mathbb{Z}.\ (\langle a, \sigma \rangle \to n \ \wedge\ \langle a, \sigma \rangle \to m \quad \Rightarrow \quad n = m)$$

2. Prove that the semantics of extended arithmetic expressions is not always defined. In other words, give a counterexample to the property below:

$$\forall a \in Aexp, \forall \sigma \in \Sigma, \exists n \in \mathbb{Z}.\ \langle a, \sigma \rangle \to n$$

3.10. Define a small-step operational semantics for IMP. To this aim, introduce a special symbol $\star$ as a termination marker and consider judgements of either the form $\langle c, \sigma \rangle \to \langle c', \sigma' \rangle$ or $\langle c, \sigma \rangle \to \langle \star, \sigma' \rangle$. Define the semantics in such a way that the evaluation is deterministic and that $\langle c, \sigma \rangle \to^* \langle \star, \sigma' \rangle$ if and only if $\langle c, \sigma \rangle \to \sigma'$ in the usual big-step semantics seen for IMP.

Chapter 4
Induction and Recursion

To understand recursion, you must first understand recursion.
(traditional joke)

Abstract 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 book. We start by introducing Noether principle of well-founded induction, from which we then derive induction principles over natural numbers, terms of a signature and derivations in a logical system. The chapter ends by presenting well-founded recursion.

4.1 Noether Principle of Well-founded Induction

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 then derive several induction methods.

4.1.1 Well-founded Relations

We recall some key mathematical notions and definitions.

Definition 4.1 (Binary relation). A *binary relation* (*relation* for short) $\prec$ 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 $b \succ a$. Moreover, we write $a \not\prec b$ in place of $(a,b) \notin \prec$.

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.

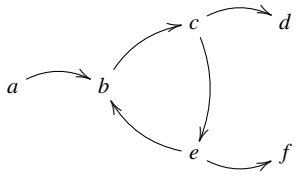


Fig. 4.1: Graph of a relation

Example 4.1. The graph in Figure 4.1 represents the relation $\prec$ over the set $\{a,b,c,d,e,f\}$ with $a \prec b$, $b \prec c$, $c \prec d$, $c \prec e$, $e \prec f$, $e \prec b$.

Definition 4.2 (Infinite descending chain). Given a relation $\prec$ over the set A, an *infinite descending chain* is an infinite sequence $\{a_i\}_{i\in\mathbb{N}}$ of elements in A such that

$$\forall i \in \mathbb{N}.\ a_{i+1} \prec a_i$$

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

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

Definition 4.3 (Well-founded relation). A relation is *well-founded* if it has no infinite descending chains.

Definition 4.4 (Transitive closure). Let $\prec$ be a relation over A. The *transitive closure* of $\prec$, written $\prec^+$, is defined by the following inference rules

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

By the first rule, $\prec$ is always included in $\prec^+$. It can be proved that $(\prec^+)^+$ always coincides with $\prec^+$.

Definition 4.5 (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

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

It can be proved that both $\prec$ and $\prec^+$ are included in $\prec^*$ and that $(\prec^*)^*$ always coincides with $\prec^*$.

Example 4.2. Consider the usual "less than" $<$ relation over integers. Since we have, e.g., the infinite descending chain

$$4 > 2 > 0 > -2 > -4 > \ldots$$

it is not well-founded. Note that its transitive closure $<^+$ is the same as $<$.

Example 4.3. Consider the usual "less than" $<$ relation over natural numbers. We cannot have an infinite descending chain $\{a_i\}_{i\in\mathbb{N}}$ because there are only a finite number of elements less than a_0. Hence the relation $<$ over $\mathbb{N}$ is well-founded. Note that its transitive closure $<^+$ is the same as $<$.

Example 4.4. Consider the usual "less than or equal to" $\leq$ relation over natural numbers. Since we have, e.g., the infinite descending chain

$$4 \geq 2 \geq 0 \geq 0 \geq 0 \geq ...$$

it is not well-founded. Note also, than any infinite descending chain must include only a finite number of elements, because there are only a finite number of elements less than or equal to a_0, and therefore there exists some $k \in \mathbb{N}$ such that $\forall i \geq k.\ a_i = a_k$. Note that $\leq$ is the reflexive and transitive closure of $<$, i.e., $<^*=\leq$.

Theorem 4.1. *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*

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

The proof of the above theorem is left as an exercise (see Problem 4.4).

With respect to the oriented graph associated with the relation $\prec$, we note that $a \prec^+ b$ means that there is a non-empty finite path from a to b, while $a \prec^* b$ means that there is a (possibly empty) finite path from a to b.

Theorem 4.2 (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. □

Example 4.5. Consider the "immediate precedence" relation $\prec$ over natural numbers, such that $n \prec n+1$ for all $n \in \mathbb{N}$. Note that the transitive closure of $\prec$ is the usual "less than" $<$ relation over natural numbers, i.e., $\prec^+=<$. By Theorem 4.2 and Example 4.3 the relation $\prec$ over $\mathbb{N}$ is well-founded.

Definition 4.6 (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 (i.e., $\forall a \in A.\ a \not\prec^+ a$).

Theorem 4.3 (Well-founded relations are acyclic). *If the relation $\prec$ is well-founded, then it is acyclic.*

Proof. We need to prove that:

$$\neg\exists a \in A.\ a \prec^+ a$$

By contradiction, we assume there is $a \in A$ such that $a \prec^+ a$. This means that $\prec^+$ is not well-founded, because we have an infinite descending chain

$$a \succ^+ a \succ^+ a \succ^+ a \ldots$$

By Theorem 4.2, $\prec$ is not well-founded, leading to a contradiction, because $\prec$ is well-founded by hypothesis. □

For example, the relation in Figure 4.1 is not acyclic and thus it is not well-founded.

Theorem 4.4 (Well-founded relations over finite sets). *Let A be a finite set and let $\prec$ be acyclic, then $\prec$ is well-founded.*

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

Definition 4.7 (Minimal element). Let $\prec$ be a relation over the set A. Given a set $Q \subseteq A$, we say that $m \in Q$ is *minimal* if there is no element $x \in Q$ such that $x \prec m$, i.e., $\forall x \in Q.\ x \not\prec m$.

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

Lemma 4.1 (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. Since any double implication $P \Leftrightarrow Q$ is equivalent to $\neg P \Leftrightarrow \neg Q$, the statement of this 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*.

We prove each implication (of the transformed statement) separately.

$\Rightarrow$) 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 $a_i \succ a_{i+1}$.

$\Leftarrow$) 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.). So we can build an infinite descending chain. □

Example 4.6 (Natural numbers). Both $n \prec n+1$ (the immediate precedence relation) and $n < n+1+k$ (the usual "less than" relation), with $n,k \in \mathbb{N}$, are simple examples of well-founded relations. In fact, from every element $n \in \mathbb{N}$ we can start a descending chain of length at most n.

Definition 4.8 (Terms over one-sorted signatures). Let $\Sigma = \{\Sigma_n\}_{n\in\mathbb{N}}$ a one-sorted signature, i.e., a set of ranked operators f such that $f \in \Sigma_n$ if f takes n arguments. We define the set of Σ-terms as the set T_Σ that is defined inductively by the following inference rule:

$$\frac{t_i \in T_\Sigma \quad i = 1,\ldots,n \quad f \in \Sigma_n}{f(t_1,\ldots,t_n) \in T_\Sigma} \tag{4.1}$$

Definition 4.9 (Terms over many sorted signatures). Let

- S be a set of *sorts* (i.e. the set of the different data types we want to consider);
- $\Sigma = \{\Sigma_{s_1\ldots s_n,s}\}_{s_1,\ldots,s_n,s\in S}$ be 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 gives 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 are in the same situation as in Definition 4.8 and 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 specialise the above rule 4.1 for each operator, i.e. we can consider the set of inference rules:

$$\left\{ \frac{t_i \in T_{\Sigma,s_i} \quad i = 1,\ldots,n}{f(t_1,\ldots,t_n) \in T_{\Sigma,s}} \right\}_{f\in\Sigma_{s_1\ldots s_n,s}} \tag{4.2}$$

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

$$\frac{}{a \in T_{\Sigma,s}} \tag{4.3}$$

Example 4.7 (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 instances such as:

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

The programs we consider are (well-formed) terms over a suitable signature Σ (possibly many-sorted). Therefore it is useful to define a well-founded containment relation between a term and its subterms. For example, we will exploit this relation when dealing with structural induction in Section 4.1.5.

Example 4.8 (Terms and subterms). 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 4.9 (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 ordered by immediate precedence. The lexicographic order relation is defined as (see Figure 4.2):

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

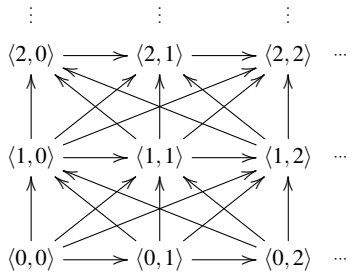


Fig. 4.2: Graph of the lexicographic order relation over pairs of natural numbers.

By Theorem 4.2, the relation $\prec$ is well-founded if and only if its transitive closure is such. Note that the relation $\prec^+$ has no cycle and any descending chain is bound by the only minimal element $\langle 0,0\rangle$. For example, we have:

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

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

Still, given any nonempty set $Q \subseteq \mathbb{N} \times \mathbb{N}$, 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 $Q_1 = \{n_1 \mid \langle n_1, n_2\rangle \in Q\}$ and m_2 is the minimum of the set $Q_2 = \{n_2 \mid \langle m_1, n_2\rangle \in Q\}$. 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

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

is a (the only) minimal element of Q. By Lemma 4.1 the relation $\prec^+$ is well-founded and so is $\prec$ (by Theorem 4.2).

4.1.2 Noether Induction

Theorem 4.5. *Let $\prec$ be a well-founded relation over the set A and let P be a unary predicate over A. Then:*

$$(\forall a \in A.\ (\forall b \prec a.\ P(b)) \Rightarrow P(a)) \quad \Leftrightarrow \quad \forall a \in A.\ P(a)$$

Proof. We prove the two implications separately:

⇒) 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 4.1 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 $\forall a \in A.\ (\forall b \prec a.\ P(b)) \to P(a)$ and instead the predicate $(\forall b \prec m.\ P(b)) \to P(m)$ is false. Therefore Q must be empty and $\forall a \in A.\ P(a)$ must hold.

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

From the first implication, it follows the validity of the following induction principle.

Definition 4.10 (Noether induction). Let $\prec$ be a well-founded relation over the set A and let P be a unary predicate over A. Then the following inference rule is called *Noether induction*.

$$\frac{\forall a \in A.\ (\forall b \prec a.\ P(b)) \Rightarrow P(a)}{\forall a \in A.\ P(a)} \tag{4.4}$$

We call a *base case* any element $a \in A$ such that the set of its predecessors $\{b \in A \mid b \prec a\}$ is empty.

4.1.3 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

$$A = \mathbb{N} \qquad n \prec m \Leftrightarrow m = n + 1$$

In this case:

- if we take $a = 0$ then $(\forall b \prec a.\ P(b)) \Rightarrow P(a)$ amounts to $P(0)$, because there is no $b \in \mathbb{N}$ such that $b \prec 0$;
- if we take $a = n + 1$ for some $n \in \mathbb{N}$, then $(\forall b \prec a.\ P(b)) \Rightarrow P(a)$ amounts to $P(n) \Rightarrow P(n+1)$.

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

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

Definition 4.11 (Weak mathematical induction).

$$\frac{P(0) \qquad \forall n \in \mathbb{N}.\ (P(n) \Rightarrow P(n+1))}{\forall n \in \mathbb{N}.\ P(n)} \tag{4.5}$$

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

4.1.4 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:

$$A = \mathbb{N} \qquad n \prec m \Leftrightarrow \exists k \in \mathbb{N}.\ m = n+k+1$$

In this case:

- if we take $a = 0$ then $(\forall b \prec a.\ P(b)) \Rightarrow P(a)$ amounts to $P(0)$, as for the case of weak mathematical induction;
- if we take $a = n+1$ for some $n \in \mathbb{N}$, then $(\forall b \prec a.\ P(b)) \Rightarrow P(a)$ amounts to $(P(0) \wedge P(1) \wedge \cdots \wedge P(n)) \Rightarrow P(n+1)$, i.e., using a more concise notation to $(\forall i \leq n.\ P(i)) \Rightarrow P(n+1)$.

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

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

Definition 4.12 (Strong mathematical induction).

$$\frac{P(0) \qquad \forall n \in \mathbb{N}.\ (\forall i \leq n.\ P(i)) \Rightarrow P(n+1)}{\forall n \in \mathbb{N}.\ P(n)} \tag{4.6}$$

The adjective "strong" comes from the fact that for proving $P(n+1)$ we can now exploit the *stronger* hypothesis $P(0) \wedge P(1) \wedge \ldots \wedge P(n)$ instead of just $P(n)$.

4.1.5 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:

$$A = T_\Sigma \qquad t_i < f(t_1, \ldots, t_n) \quad i = 1, \ldots, n$$

Definition 4.13 (Structural induction).

$$\frac{\forall t \in T_\Sigma.\ (\forall t' < t.\ P(t')) \Rightarrow P(t)}{\forall t \in T_\Sigma.\ P(t)} \tag{4.7}$$

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

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

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 4.10. Let us consider the grammar of IMP arithmetic expressions:

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

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

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

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

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

Example 4.11 (Termination of arithmetic expressions). Let us consider the case of arithmetic expressions seen above and prove that the evaluation of expressions always terminates (a property that is also called *normalisation*):[1]

$$\forall a \in Aexp.\ \forall \sigma \in \Sigma.\ \exists m \in \mathbb{N}.\ \langle a, \sigma \rangle \to m$$

[1] We recall that the (overloaded) symbol Σ stands here for the set of memories and not for a generic signature.

In this case we let

$$P(a) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \exists m \in \mathbb{N}.\ \langle a, \sigma \rangle \to m.$$

We prove that $\forall a \in Aexp.\ P(a)$ by structural induction. This amounts to prove that

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

$$P(a_0) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \exists m_0 \in \mathbb{N}.\ \langle a_0, \sigma \rangle \to m_0$$
$$P(a_1) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \exists m_1 \in \mathbb{N}.\ \langle a_1, \sigma \rangle \to m_1.$$

  We want to prove that

$$P(a_0 + a_1) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \exists m \in \mathbb{N}.\ \langle a_0 + a_1, \sigma \rangle \to m.$$

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

Example 4.12 (Determinism of arithmetic expressions). Let us consider again the case of IMP arithmetic expressions and prove that their evaluation is deterministic:

$$\forall a \in Aexp.\ \forall \sigma \in \Sigma.\ \forall m, m' \in \mathbb{N}.\ (\langle a, \sigma \rangle \to m \wedge \langle a, \sigma \rangle \to m') \Rightarrow m = m'$$

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 exactly one value. We let

$$P(a) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \forall m, m' \in \mathbb{N}.\ (\langle a, \sigma \rangle \to m \wedge \langle a, \sigma \rangle \to m') \Rightarrow m = m'$$

We proceed by structural induction.

$a = n$) We want to prove that

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

holds. Let us take generic σ, m, m'. We assume the premises $\langle n, \sigma \rangle \to m$ and $\langle n, \sigma \rangle \to m'$ and prove that $m = m'$. In fact, there is only one

rule that can be used to evaluate an integer number, and it always returns the same value. Therefore $m = n = m'$.

$a = x$) We want to prove that

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

holds. We assume the premises $\langle x, \sigma \rangle \to m$ and $\langle x, \sigma \rangle \to m'$ and prove that $m = m'$. Again, there is only one rule that can be applied, whose outcome depends on σ. Since σ is the same in both cases, $m = \sigma(x) = m'$.

$a = a_0 + a_1$) We assume the inductive hypotheses

$$P(a_0) \stackrel{\text{def}}{=} \forall \sigma, m_0, m_0'.\ (\langle a_0, \sigma \rangle \to m_0 \wedge \langle a_0, \sigma \rangle \to m_0') \Rightarrow m_0 = m_0'$$
$$P(a_1) \stackrel{\text{def}}{=} \forall \sigma, m_1, m_1'.\ (\langle a_1, \sigma \rangle \to m_1 \wedge \langle a_1, \sigma \rangle \to m_1') \Rightarrow m_1 = m_1'$$

and we want to prove that $P(a_0 + a_1)$, i.e., that:

$$\forall \sigma, m, m'.\ (\langle a_0 + a_1, \sigma \rangle \to m \wedge \langle a_0 + a_1, \sigma \rangle \to m') \Rightarrow m = m'$$

We assume the premises $\langle a_0 + a_1, \sigma \rangle \to m$ and $\langle a_0 + a_1, \sigma \rangle \to m'$ and prove that $m = m'$. By the first premise, it must be $m = m_0 + m_1$ for some m_0, m_1 such that $\langle a_0, \sigma \rangle \to m_0$ and $\langle a_1, \sigma \rangle \to m_1$, because there is only one rule applicable to $a_0 + a_1$; analogously, by the second premise, we must have $m' = m_0' + m_1'$ for some m_0', m_1' such that $\langle a_0, \sigma \rangle \to m_0'$ and $\langle a_1, \sigma \rangle \to m_1'$. By inductive hypothesis $P(a_0)$ we know that $m_0 = m_0'$ and by $P(a_1)$ we have $m_1 = m_1'$. Thus, $m = m_0 + m_1 = m_0' + m_1' = m'$ and thus $P(a_0 + a_1)$ holds.

The remaining cases for $a = a_0 - a_1$ and $a = a_0 \times a_1$ follow exactly the same pattern as that of $a = a_0 + a_1$.

4.1.6 Induction on Derivations

We can define an induction principle over the set of derivations of a logical system. See Definitions 2.1 and 2.4 for the notion of inference rule and of derivation.

Definition 4.14 (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 4.13 (Immediate sub-derivation). Let us consider the derivation

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

the two derivations

$$\dfrac{}{\langle 1,\sigma\rangle \to 1}\,\text{num} \qquad \dfrac{}{\langle 2,\sigma\rangle \to 2}\,\text{num}$$

are immediate sub-derivations of the derivation that exploits rule (sum).

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

Definition 4.15 (Proper sub-derivation). We say that d' is a *proper sub-derivation* of d if and only if $d' \prec^+ d$.

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

For example, the induction principle based on immediate sub-derivation can be phrased as follows.

Definition 4.16 (Induction on derivations). Let R be a set of inference rules and D the set of derivations defined on R, then:

$$\dfrac{\forall \{x_1,\ldots,x_n\}/y \in R.\ (\forall d_i \Vdash_R x_i.\ P(d_1)\wedge\ldots\wedge P(d_n)) \Rightarrow P(\{d_1,\ldots,d_n\}/y)}{\forall d \in D.\ P(d)} \tag{4.8}$$

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

Induction on derivations shares similarities with structural induction. The analogy comes from viewing the (instances of) inference rules as symbolic operators to construct derivations, with axioms playing the role of constants.

4.1.7 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. the theorems of the logical system) and we need to prove that the application of any such rule will not compromise the validity of a given predicate.

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

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

The rule induction principle states that to show the property P holds for all elements of I_R, we can prove the following:

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

Definition 4.17 (Rule induction). Let R be a logical system. The principle of rule induction is:

$$\frac{\forall (X/y) \in R.\ (X \subseteq I_R \ \wedge\ \forall x \in X.\, P(x)) \Rightarrow P(y)}{\forall x \in I_R.\, P(x)} \tag{4.9}$$

The principle of rule induction is a useful variant of the induction on derivation. In fact by assuming that $X \subseteq I_R$ it follows that there is a derivation d_i for each theorem $x_i \in X$, so that a longer derivation for y is built by applying the rule $(X/y) \in R$ to $d_1,\ldots,d_n$.

Note that in many cases we will use the simpler but less powerful rule

$$\frac{\forall (X/y) \in R.\ (\forall x \in X.\, P(x)) \Rightarrow P(y)}{\forall x \in I_R.\, P(x)} \tag{4.10}$$

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 the restriction.

Example 4.14 (Determinism of IMP commands). We have seen in Example 4.12 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

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

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, i.e., statements of the form $\langle c,\sigma\rangle \to \sigma'$:

$$Q(\langle c,\sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1 \in \Sigma.\ \langle c,\sigma\rangle \to \sigma_1 \Rightarrow \sigma' = \sigma_1$$

We proceed by rule induction:

rule skip): we want to show that

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

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

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

Let us take a generic memory σ_1 and assume the premise $\langle x := a, \sigma \rangle \to \sigma_1$ of the implication holds. We 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

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

we want to show that

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

We assume the premise $\langle c_0; c_1, \sigma \rangle \to \sigma_1$ and prove that $\sigma_1 = \sigma'$. 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:

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

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

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

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

rule ifft): assuming

$$\langle b, \sigma \rangle \to \mathbf{true}$$
$$Q(\langle c_0, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1 .\ \langle c_0, \sigma \rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma'$$

we want to show that

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

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

$$\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma_1 \quad \nwarrow^* \quad \langle c_0, \sigma\rangle \to \sigma_1$$

But then, exploiting the inductive hypothesis

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

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

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

rule whff): assuming

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

we want to show that

$$Q(\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma$$

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

$$\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma_1 \quad \nwarrow^*_{\sigma_1 = \sigma} \quad \square$$

rule whtt): assuming

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

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

$$Q(\langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma'$$

we want to show that

$$Q(\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\ \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma'$$

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

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

But now we can apply the first inductive hypotheses:

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

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

$$Q(\langle \textbf{while } b \textbf{ do } c, \sigma'' \rangle \to \sigma') \stackrel{\text{def}}{=} \forall \sigma_1.\, \langle \textbf{while } b \textbf{ do } c, \sigma'' \rangle \to \sigma_1 \Rightarrow \sigma_1 = \sigma'$$

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

4.2 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). A special class of functions defined on natural numbers according to the principle of well-founded recursion is that of *primitive recursive functions*.

Definition 4.18 (Primitive recursive functions). The primitive recursive functions are those (n-ary) functions over natural numbers obtained according to (any finite application of) the following rules:

zero: The constant 0 is primitive recursive.

succ.: The successor function $s : \mathbb{N} \to \mathbb{N}$ with $s(n) = n+1$ is primitive recursive.

proj.: For any $i,k \in \mathbb{N}, 1 \leq i \leq k$, the projection functions $\pi_i^k : \mathbb{N}^k \to \mathbb{N}$ with

$$\pi_i^k(n_1,...,n_k) = n_i$$

are primitive recursive.

comp.: Given a k-ary primitive recursive function $f : \mathbb{N}^k \to \mathbb{N}$, and k primitive recursive functions $g_1,...,g_k : \mathbb{N}^m \to \mathbb{N}$ of arity m, the m-ary function h obtained by composing f with $g_1,...,g_k$ as shown below is primitive recursive:

$$h(n_1,...,n_m) \stackrel{\text{def}}{=} f(g_1(n_1,...,n_m),...,g_k(n_1,...,n_m))$$

pr.rec.: Given a k-ary primitive recursive function $f : \mathbb{N}^k \to \mathbb{N}$ and a $(k+2)$-ary primitive recursive function $g : \mathbb{N}^{k+2} \to \mathbb{N}$, the $(k+1)$-ary function $h : \mathbb{N}^{k+1} \to \mathbb{N}$ defined as the primitive recursion of f and g below is primitive recursive:

$$\begin{aligned} h(0,n_1,...,n_k) &= f(x_1,...,x_k) \\ h(s(n),n_1,...,n_k) &= g(n,h(n,n_1,\ldots,n_k),n_1,\ldots,n_k) \end{aligned}$$

Note that $\pi_1^1 : \mathbb{N} \to \mathbb{N}$ is the usual identity function. It can be proved that every primitive recursive function is total and computable.

Example 4.15. Addition can be recursively defined with the rules:

$$add(0,m) \stackrel{\text{def}}{=} m$$
$$add(n+1,m) \stackrel{\text{def}}{=} add(n,m)+1.$$

This does not fit immediately into the above scheme of primitive recursive functions, but we can rephrase the definition as:

$$add(0,n_1) \stackrel{\text{def}}{=} \pi^1_1(n_1)$$
$$add(s(n),n_1) \stackrel{\text{def}}{=} s(\pi^3_2(n,add(n,n_1),n_1))$$

In the primitive recursive style, *add* plays the role of h, the identity function π^1_1 plays the role of f and the composition of s with π^3_2 plays the role of g (so that it receives the unnecessary arguments n and n_1).

Let us make the well-founded recursion more precise.

Definition 4.19 (Set of predecessors). Given a well founded relation $\prec \subseteq A \times A$, the set of *predecessors* of a set $I \subseteq A$ is the set

$$\prec^{-1} I = \{ b \in A \mid \exists a \in I.\ b \prec a \}$$

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

Theorem 4.6 (Well-founded recursion). *Let $\prec \subseteq A \times A$ be a well-founded relation over A. Let us consider a function F with $F(a,h) \in B$ for any*

- $a \in A$
- $h :\!\prec^{-1} \{a\} \to B$ *(i.e., h is any function whose domain is the set of predecessors of a and whose codomain is B)*

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

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

Proof. The proof is divided in two parts: 1) we first demonstrate that if such a function f exists, then it is unique; and 2) we prove its existence.

1. Uniqueness follows if we can prove the predicate $\forall a.\ P(a)$, where

$$P(a) \stackrel{\text{def}}{=} (\forall y \prec^* a.\ (\ f(y) = F(y, f \restriction \prec^{-1} \{y\}) \ \wedge\ g(y) = F(y, g \restriction \prec^{-1} \{y\})\ )\ )$$
$$\Rightarrow\ f(a) = g(a)$$

In fact, suppose there are two functions $f, g : A \to B$ such that:

$$\forall a \in A.\ f(a) = F(a, f \restriction \prec^{-1} \{a\})$$
$$\forall a \in A.\ g(a) = F(a, g \restriction \prec^{-1} \{a\})$$

Clearly, for any $a \in A$ the premise

$$(\ \forall y \prec^* a.\ (\ f(y) = F(y, f \restriction \prec^{-1} \{y\}) \ \wedge\ g(y) = F(y, g \restriction \prec^{-1} \{y\})\ )\ )$$

is true and thus we can conclude $f(a) = g(a)$.
The proof that $\forall a.\ P(a)$ goes by well-founded induction on $\prec$. For $a \in A$, we assume that $\forall b \prec a.\ P(b)$ and we want to prove $P(a)$. Suppose that

$$(\ \forall y \prec^* a.\ (\ f(y) = F(y, f \restriction \prec^{-1} \{y\}) \ \wedge\ g(y) = F(y, g \restriction \prec^{-1} \{y\})\ )\ )$$

We need to prove that $f(a) = g(a)$. For $b \prec a$ we must have $f(b) = g(b)$, because $P(b)$ holds by inductive hypothesis. Thus

$$f \restriction \prec^{-1} \{a\} = g \restriction \prec^{-1} \{a\}$$

and therefore

$$f(a) = F(y, f \restriction \prec^{-1} \{a\}) = F(y, g \restriction \prec^{-1} \{a\}) = g(a)$$

2. For the existence, we build a family of functions

$$\{f_a : \prec^{*-1} \{a\} \to B\}_{a \in A}$$

and then take $f \stackrel{\text{def}}{=} \bigcup_{a \in A} f_a$. The existence of the functions f_a is guaranteed by proving that the following property holds for all $x \in A$:

$$Q(x) \stackrel{\text{def}}{=} \exists f_x : \prec^{*-1} \{x\} \to B.\ \forall y \prec^* x.\ f_x(y) = F(y, f_x \restriction \prec^{-1} \{y\})$$

The proof goes by well-founded recursion. We assume $\forall b \prec a.\ Q(b)$ and prove that $Q(a)$ holds. Let $b \prec a$ and $f_b : \prec^{*-1} \{b\} \to B$ be the function such that

$$\forall y \prec^* b.\ f_b(y) = F(y, f_b \restriction \prec^{-1} \{y\}).$$

We build a relation $h \subseteq A \times B$ defined by

$$h \stackrel{\text{def}}{=} \bigcup_{b \prec a} f_b$$

Now for any $b \prec a$ there is at least one pair of the form $(b, c) \in h$ for some $c \in B$, because $b \prec^* b$. By the uniqueness property proved before, we have that such a pair is unique. Hence h satisfies the function property. Finally, we let

$$f_a \stackrel{\text{def}}{=} h \cup \{(a, F(a, h))\}$$

to get a function $f_a : \prec^{*-1} \{a\} \to B$ such that

$$\forall y \prec^* a.\ f_a(y) = F(y, f_a \restriction \prec^{-1} \{y\})$$

proving that $Q(a)$ holds. □

Theorem 4.6 guarantees that, if we (recursively) define f over any $a \in A$ only in terms of the predecessors of a, then f is uniquely determined on all a. 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 fixpoint 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 4.16 (Product as primitive recursion). Let us consider the Peano formula that defines the product of natural numbers

$$p(0,y) \stackrel{\text{def}}{=} 0$$
$$p(x+1,y) \stackrel{\text{def}}{=} y + p(x,y)$$

Let us write the definition in a slightly different way

$$p_y(0) \stackrel{\text{def}}{=} 0$$
$$p_y(x+1) \stackrel{\text{def}}{=} y + p_y(x)$$

Let us recast the Peano formula seen above to the formal scheme of well-founded recursion.

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

Example 4.17 (Structural recursion). Let us consider the signature Σ for binary trees $A = T_\Sigma$, where $\Sigma_0 = \{0,1,\ldots\}$ and $\Sigma_2 = \mathsf{cons}$ (where $\mathsf{cons}(x,y)$ is the constructor for building a tree out of its left and right subtree). Take the well-founded relation $x_i \prec \mathit{cons}(x_1,x_2)$, $i = 1,2$. Let $B = \mathbb{N}$.

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

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

where $\mathit{atom}(x)$ returns true if x is a leaf; $\mathit{car}(x)$ denotes the left subtree of x; $\mathit{cdr}(x)$ the right subtree of x. The same function defined in the structural recursion style is

$$\mathit{sum}(n) \stackrel{\text{def}}{=} n$$
$$\mathit{sum}(\mathsf{cons}(x,y)) \stackrel{\text{def}}{=} \mathit{sum}(x) + \mathit{sum}(y)$$

or, according to the well-founded recursive scheme,

$$F(n, sum \restriction \varnothing) \stackrel{\text{def}}{=} n$$
$$F(\mathsf{cons}(x,y), sum \restriction \{x,y\}) \stackrel{\text{def}}{=} sum(x) + sum(y)$$

For example, for $q \stackrel{\text{def}}{=} \mathsf{cons}(3, \mathsf{cons}(\mathsf{cons}(2,3),4))$ we have:

$$\begin{aligned} sum(q) &= sum(3) + sum(\mathsf{cons}(\mathsf{cons}(2,3),4)) \\ &= 3 + (sum(\mathsf{cons}(2,3)) + sum(4)) \\ &= 3 + ((sum(2) + sum(3)) + 4) \\ &= 3 + (2+3) + 4) \\ &= 12 \end{aligned}$$

Example 4.18 (Ackermann function). The Ackermann function is one of the earliest examples of a computable, total recursive function that is not primitive recursive: it grows faster than any such function. The Ackermann function $ack(z,x,y) = ack_y(z,x)$ is defined by well-founded recursion (exploiting the lexicographic order over pairs of natural numbers) by letting

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

We have

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

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

Intuitively, $ack(1,x,y)$ applies addition of y for x times.

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

In other words, $ack(2,x,y)$ applies multiplication for y for x times. Likewise, $ack(3,x,y)$ applies exponentiation to the yth power for x times, and so on.

For example, we have:

$$
\begin{aligned}
ack(0,0,0) &= 0+0=0\\
ack(1,1,1) &= 1\times 1=1\\
ack(2,2,2) &= 2^2=4\\
ack(3,3,3) &= 3^{3^3}=3^{27}\simeq 7.6\times 10^{12}
\end{aligned}
$$

Problems

4.1. Consider the logical system R corresponding to the rules of the grammar:

$$S ::= aB \mid bA \qquad A ::= a \mid aS \mid bAA \qquad B ::= b \mid bS \mid aBB$$

where the well formed formulas are of the form $x \in L_X$, where X is either S or A or B and where x is a string on the alphabet $\{a,b\}$.

1. Write down explicitly the rules in R.
2. Prove by rule induction—in one direction—and by mathematical induction on the length of the strings—in the other direction—that the strings generated by S are all the nonempty strings with the same number of a's and b's (i.e., prove the formal predicate $P(x \in L_S) \stackrel{\text{def}}{=} x|_a = x|_b \neq 0$, where $x|_s$ denotes the number of occurrences of the symbol s in the string x), while A generates all the strings with an additional a (formally $P(x \in L_A) \stackrel{\text{def}}{=} x|_a = 1 + x|_b$) and B with an additional b.
3. Finally prove by induction on derivations that

   $$P(d/(x \in L_X)) \stackrel{\text{def}}{=} |d| \leq |x|$$

   i.e., the depth of any derivation d is smaller or equal that the length of the string x generated by it.

4.2. Define by well-founded recursion the function

$$locs : Com \longrightarrow \wp(\mathbf{Loc})$$

that, given a command, returns the set of locations that appear on the left-hand side of some assignment.

Then, prove that $\forall c \in Com, \forall \sigma, \sigma' \in \Sigma$

$$\langle c, \sigma\rangle \to \sigma' \qquad \text{implies} \qquad \forall y \notin locs(c).\ \sigma(y) = \sigma'(y).$$

4.3. Let w denote the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } x \neq 0 \textbf{ do } (x := x-1\ ;\ y := y+1).$$

Prove by rule induction that $\forall \sigma, \sigma' \in \Sigma$

$$\langle w, \sigma \rangle \to \sigma' \qquad \text{implies} \qquad \sigma(x) \geq 0 \ \wedge \ \sigma' = \sigma \left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x} \right].$$

4.4. Let R be a binary relation over the set A, i.e., $R \subseteq A \times A$. Let R^+, called the *transitive closure* of R, be the relation defined by the following two rules:

$$\frac{x \, R \, y}{x \, R^+ \, y} \qquad \frac{x \, R^+ \, y \quad y \, R^+ \, z}{x \, R^+ \, z}$$

1. Prove that for any x and y

$$x \, R^+ \, y \quad \Leftrightarrow \quad \exists k > 0. \, \exists z_0, \ldots, z_k. \, x = z_0 \ \wedge \ z_0 \, R \, z_1 \ \wedge \ \ldots \wedge \ z_{k-1} \, R \, z_k \ \wedge \ z_k = y$$

   (Hint: Prove the implication $\Rightarrow$ by rule induction and the implication $\Leftarrow$ by induction on the length k of the R-chain).
2. Give the rules that define instead a relation R' such that

$$x \, R' \, y \quad \Leftrightarrow \quad \exists k \geq 0. \, \exists z_0, \ldots, z_k. \, x = z_0 \ \wedge \ z_0 \, R \, z_1 \ \wedge \ \ldots \wedge \ z_{k-1} \, R \, z_k \ \wedge \ z_k = y.$$

4.5. Let IMP$^-$ be the language obtained from IMP by removing the **while** construct. Exploit the operational semantics to prove that in IMP$^-$, for every command c it holds the termination property

$$\forall \sigma \in \Sigma. \, \exists \sigma' \in \Sigma. \, \langle c, \sigma \rangle \to \sigma'$$

4.6. Let us consider the following rules, where m, n and k are positive natural numbers.

$$\frac{}{(m,m) \to m} \qquad \frac{(n,m) \to k}{(m,n) \to k} \ m < n \qquad \frac{(m-n,n) \to k}{(m,n) \to k} \ m > n$$

Prove by rule induction that, for any $n, m, k > 0$,

$$(m,n) \to k \qquad \text{implies} \qquad k = \gcd(m,n).$$

(Hint: Prove that any common divisor of m and n with $m > n$ is also a common divisor of $m-n$ and n and vice versa). **Note**: $\gcd(m,n)$ denotes the greatest common divisor of m and n, i.e., if we write $d \,|\, j$ to mean that d divides j (in other words, that there exists h such that $j = d \times h$), then $\gcd(n,m)$ is the natural number d such that $d|m \wedge d|n$ and for any d' such that $d'|m \wedge d'|n$ we have $d' \leq d$.

4.7. Prove that, according to the operational semantics of IMP, for any boolean expression b, command c and stores σ, σ'

$$\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \to \sigma' \qquad \text{implies} \qquad \langle b, \sigma' \rangle \to \mathit{false}$$

Explain which induction principle you exploit in the proof.

4.8. Exploit the property from Problem 4.7 to prove that for any $b \in \mathit{Bexp}$ and $c \in \mathit{Com}$ we have $c_1 \sim c_2 \sim c_3$ where:

$$c_1 \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } c$$
$$c_2 \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } (c \,;\textbf{while } b \textbf{ do } c)$$
$$c_3 \stackrel{\text{def}}{=} (\textbf{while } b \textbf{ do } c) \,;\, \textbf{while } b \textbf{ do } c$$

4.9. Define by well-founded recursion the function

$$locs : Aexp \longrightarrow \wp(\mathbf{Loc})$$

that, given an arithmetic expression a, returns the set of locations that occur in a. Use structural induction to show that $\forall a \in Aexp.\ \forall \sigma, \sigma' \in \Sigma.\ \forall n, m \in \mathbb{Z}$

$$\langle a, \sigma \rangle \to n \ \wedge\ \langle a, \sigma' \rangle \to m \ \wedge\ \forall x \in locs(a).\ \sigma(x) = \sigma'(x) \qquad \text{implies} \qquad n = m.$$

4.10. Consider the IMP program

$$w \stackrel{\text{def}}{=} \textbf{while } \neg(x = y) \textbf{ do } (x := x + 1 \,;\, y := y - 1)$$

Define the set of stores $T = \{\sigma \mid ...\}$ for which the program w terminates and:

1. Prove formally that for any store $\sigma \in T$ there exists σ' such that $\langle w, \sigma \rangle \to \sigma'$. (Hint: use well-founded induction on T)
2. Prove (by using the rule for divergence) that $\forall \sigma \notin T.\ \langle w, \sigma \rangle \not\to$.

4.11. Let us consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } x \neq 0 \textbf{ do } x := x - y.$$

1. Prove that, for any σ, σ', if $\langle w, \sigma \rangle \to \sigma'$ then there exists an integer k such that $\sigma(x) = k \times \sigma(y)$.
2. Give a store σ such that $\sigma(x) = k \times \sigma(y)$ for some integer k but such that $\langle w, \sigma \rangle \not\to$. Explain why the program diverges for the given σ.
3. Define a command c such that, for any σ, σ', $\langle c, \sigma \rangle \to \sigma'$ iff $\sigma(x) = k \times \sigma(y)$ for some integer k. Sketch the proof of the double implication.

4.12. Recall that the *height* or *depth* of a derivation d is recursively defined as follows:

$$depth(\varnothing / y) \stackrel{\text{def}}{=} 1$$
$$depth(\{d_1, ..., d_n\} / y) \stackrel{\text{def}}{=} 1 + \max\{depth(d_1), ..., depth(d_n)\}$$

Given the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } x > 0 \textbf{ do } x := x - 1$$

prove by induction on n that for any $\sigma \in \Sigma$ with $\sigma(x) = n \geq 0$ the derivation of

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

has depth $n+3$.

4.13. The *binomial coefficients* $\binom{n}{k}$, with $n,k \in \mathbb{N}$ and $0 \leq k \leq n$, are defined by:

$$\binom{n}{0} \stackrel{\text{def}}{=} 1 \qquad \binom{n}{n} \stackrel{\text{def}}{=} 1 \qquad \binom{n+1}{k+1} \stackrel{\text{def}}{=} \binom{n}{k} + \binom{n}{k+1}.$$

Prove that the above definition is given by well-founded recursion, specifying the well-founded relation and the corresponding function $F(b,h)$.

4.14. Consider the well-known sequence of Fibonacci numbers, defined as:

$$F(0) \stackrel{\text{def}}{=} 1 \qquad F(1) \stackrel{\text{def}}{=} 1 \qquad F(n+2) \stackrel{\text{def}}{=} F(n+1) + F(n).$$

where $n \in \mathbb{N}$. Explain why the above definition is given by well-founded recursion and make explicit the well-founded relation to be used.

Chapter 5
Partial Orders and Fixpoints

Good old Watson! You are the one fixed point in a changing age.
(Sherlock Holmes)

Abstract This chapter is devoted to the introduction of the foundations of the denotational semantics of computer languages. The concepts of complete partial orders with bottom and of monotone and continuous functions are introduced and then the main fixpoint theorem is presented. The chapter is concluded by studying the immediate consequence operator that is used to relate logical systems and fixpoint theory.

5.1 Orders and Continuous Functions

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 composite program is given by combining the meanings of its constituents*. The compositional property of denotational semantics is obtained by defining the semantics by structural recursion. Obviously there are particular issues in defining the interpretation of the "while" construct of IMP, since the semantics of this construct, as we saw in the previous chapters, is inherently recursive. General recursion is forbidden in structural recursion, which allows only the use of sub-terms. The solution to this problem is given by solving equations of the type $x = f(x)$, namely by finding the fixpoint(s) of suitable functions f. On the one hand we would like to ensure that each recursive definition that we introduce has 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 when more than one

are available, and to provide an approximation method for computing it, which is given by the fixpoint theorem (Theorem 5.6).

5.1.1 Orders

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

Definition 5.1 (Partial order). A *partial order* is a pair $(P, \sqsubseteq_P)$ where P is a set and $\sqsubseteq_P \subseteq P \times P$ is a binary relation (i.e., it is a set of pairs of elements of P) which is:

reflexive: $\forall p \in P.\ p \sqsubseteq_P p$
antisymmetric: $\forall p, q \in P.\ p \sqsubseteq_P q \wedge q \sqsubseteq_P p \Longrightarrow p = q$
transitive: $\forall p, q, r \in P.\ p \sqsubseteq_P q \wedge q \sqsubseteq_P r \Longrightarrow p \sqsubseteq_P r$

We call $(P, \sqsubseteq_P)$ a *poset* (for *partially ordered set*).

We will conveniently omit the subscript P from $\sqsubseteq_P$ when no confusion can arise. We write $p \sqsubset q$ when $p \sqsubseteq q$ and $p \neq q$.

Example 5.1 (Powerset). Let $(\wp(S), \subseteq)$ be the powerset of a set S together with the inclusion relation. It is easy to see that $(\wp(S), \subseteq)$ is a poset.

reflexive: $\forall s \subseteq S.\ s \subseteq s$
antisymmetric: $\forall s_1, s_2 \subseteq S.\ s_1 \subseteq s_2 \wedge s_2 \subseteq s_1 \Rightarrow s_1 = s_2$
transitive: $\forall s_1, s_2, s_3 \subseteq S.\ s_1 \subseteq s_2 \subseteq s_3 \Rightarrow s_1 \subseteq s_3$

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

Remark 5.1 (Partial orders vs well-founded relations). Partial order relations should not be confused with the well-founded relations studied in the previous chapter. In fact:

- Any well-founded relation (on a non-empty set) is not reflexive (otherwise an infinite descending chain could be constructed by iterating over the same element).
- Any well-founded relation is antisymmetric (the premise $p \sqsubseteq q \wedge q \sqsubseteq p$ must be always false, otherwise an infinite descending chain could be constructed).
- A well-founded relation can be transitive, but it is not necessarily so (e.g., the immediate precedence relation over natural numbers is well-founded but not transitive, instead the 'less than' relation is well-founded and transitive).
- Any (non-empty) partial order has an infinite descending chain (take any element p and the chain $p \sqsupseteq p \sqsupseteq p \ldots$) and is thus non well-founded.
- If we take the relation $\sqsubset$ induced by a partial order $\sqsubseteq$, then it can be well-founded, but it is not necessarily so (e.g., the strict inclusion relation over $\wp(\mathbb{N})$ has an infinite descending chain whose ith element is the set $\{n \mid n \in \mathbb{N} \wedge n \geq i\}$).

- If we take the reflexive and transitive closure $\prec^*$ of a well-founded relation $\prec$, then it is a partial order (reflexivity and transitivity are obvious, for the antisymmetric property, suppose that there are two elements $p \neq q$ such that $p \prec^* q \wedge q \prec^* p$ then there would be a cycle over p using $\prec$, contradicting the assumption that $\prec$ is well-founded).

Two elements $p, q \in P$ are called *comparable* if $p \sqsubseteq q$ or $q \sqsubseteq p$. When any two elements of a partial order are comparable, then it is called a *total* order.

Definition 5.2 (Total order). Let $(P, \sqsubseteq)$ be a partial order such that:

$$\forall p, q \in P.\ p \sqsubseteq q \ \vee\ q \sqsubseteq p$$

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

Example 5.2. Given a set S, its powerset $(\wp(S), \subseteq)$ ordered by inclusion is a total order if and only if $|S| \leq 1$. In fact, in one direction suppose that $(\wp(S), \subseteq)$ is a total order and take $p, q \in S$; clearly $\{p\} \subseteq \{q\} \vee \{q\} \subseteq \{p\}$ holds only when $p = q$, i.e., S must have at most one element. Vice versa, if $S = \varnothing$ then $\wp(S) = \{\varnothing\}$ and $\varnothing \subseteq \varnothing$; if $S = \{p\}$ for some p, then $\wp(S) = \{\varnothing, \{p\}\}$ and $\varnothing \subseteq \varnothing \subseteq \{p\} \subseteq \{p\}$.

Theorem 5.1 (Subsets of an order). *Let $(P, \sqsubseteq_P)$ be a partial order and let $Q \subseteq P$. Then $(Q, \sqsubseteq_Q)$ is a partial order, with $\sqsubseteq_Q \stackrel{\text{def}}{=} \sqsubseteq_P \cap (Q \times Q)$. Similarly, if $(P, \sqsubseteq_P)$ is a total order then $(Q, \sqsubseteq_Q)$ is a total order.*

The proof is left as an easy exercise to the reader (see Problem 5.1).

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

Example 5.3 (Natural Numbers). Let $(\mathbb{N}, \leq)$ be the set of natural numbers with the usual order; $(\mathbb{N}, \leq)$ is a total order.

reflexive:	$\forall n \in \mathbb{N}.\ n \leq n$
antisymmetric:	$\forall n, m \in \mathbb{N}.\ n \leq m \wedge m \leq n \Longrightarrow m = n$
transitive:	$\forall n, m, z \in \mathbb{N}.\ n \leq m \wedge m \leq z \Longrightarrow n \leq z$
total:	$\forall n, m \in \mathbb{N}.\ n \leq m \vee m \leq n$

Example 5.4 (Discrete order). Let $(P, \sqsubseteq)$ be a partial order defined as follows:

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

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

Example 5.5 (Flat order). A *flat order* is a partial order $(P, \sqsubseteq)$ for which there exists an element $\bot \in P$ such that

$$\forall p, q \in P.\ p \sqsubseteq q \Leftrightarrow\ p = \bot \vee p = q$$

The element $\bot$ is called *bottom* and it is unique. In fact, suppose that two such elements $\bot_1, \bot_2$ exist. Then, we have $\bot_1 \sqsubseteq \bot_2$ and also $\bot_2 \sqsubseteq \bot_1$; thus by antisymmetry we have $\bot_1 = \bot_2$.

5.1.2 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 order 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 manageable, indeed we repeat many times redundant information. For example in the usual natural numbers order we would have $n+1$ incoming arrows and infinite outgoing arrows, for a node labelled by n. We need a more compact notation, which leaves implicit some information that can be inferred by exploiting the property of partial orders. This notation is represented by the Hasse diagrams. The idea is to omit: 1) every reflexive arc (from a node to itself), because we know by reflexivity that such an arc is present for every node; and 2) every arc from a to c when there is a node b with an arc from a to b and one from b to c, because the presence of the arc from a to c can be inferred by transitivity.

Definition 5.3 (Hasse Diagram). Given a poset $(P, \sqsubseteq)$, let R be the binary relation defined by:

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

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

$$H \stackrel{\text{def}}{=} \sqsubseteq \setminus R$$

Note that the first rule can be written more concisely as

$$\frac{x \sqsubset y \quad y \sqsubset z}{xRz}$$

The Hasse diagram omits the information deducible by transitivity and reflexivity. A simple example of Hasse diagram is in Figure 5.1.

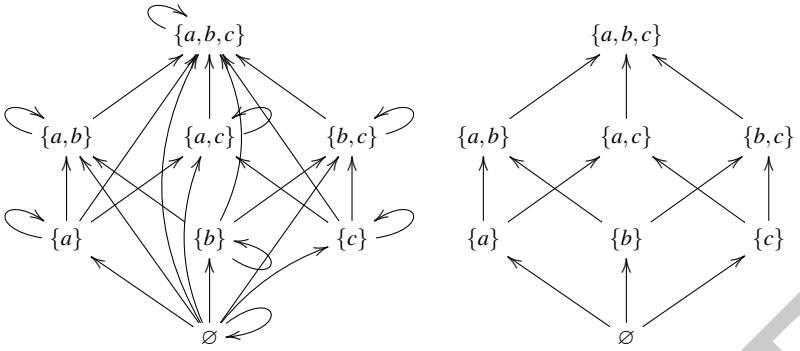
To ensure that all the needed information is contained in the Hasse diagram we rely on the following theorem.

Theorem 5.2 (Order relation, Hasse diagram Equivalence). *Let $(P, \sqsubseteq)$ a partial order with P a finite set, and let H be its Hasse diagram. Then, the transitive and reflexive closure H^* of H is equal to $\sqsubseteq$.*

Proof. Formally, we want to prove the two inclusions $H^* \subseteq \sqsubseteq$ and $\sqsubseteq \subseteq H^*$ separately, where the relation H^* is defined by the inference rules below:

$$\frac{}{xH^*x} \qquad \frac{xHy}{xH^*y} \qquad \frac{xH^*y \wedge yH^*z}{xH^*z}$$

$H^* \subseteq \sqsubseteq$: Suppose $x\,H^*\,y$. Then, there exists (see Problem 4.4) $k \in \mathbb{N}$ and $z_0, \ldots, z_k$ such that



(a) $\wp(\{a,b,c\})$ ordered by inclusion (b) Hasse diagram for $\wp(\{a,b,c\})$

Fig. 5.1: Hasse diagram for the powerset over $\{a,b,c\}$ ordered by inclusion

$$x = z_0 \ \wedge \ z_0 \, H \, z_1 \ \wedge \ \ldots \wedge \ z_{k-1} \, H \, z_k \ \wedge \ z_k = y$$

Since $H \subseteq \sqsubseteq$ by definition, we have

$$x = z_0 \ \wedge \ z_0 \sqsubseteq z_1 \ \wedge \ \ldots \wedge \ z_{k-1} \sqsubseteq z_k \ \wedge \ z_k = y$$

Hence, by transitivity of $\sqsubseteq$ it follows that $x \sqsubseteq y$.

$\sqsubseteq \subseteq H^*$: Given $x \sqsubseteq y$, let us denote by $]x,y[$ the set of elements strictly contained between x and y, i.e.,

$$]x,y[\stackrel{\text{def}}{=} \{z \mid x \sqsubset z \ \wedge \ z \sqsubset y\}.$$

Clearly $]x,y[$ is finite because P is finite. We prove that $x H^* y$ by mathematical induction on the number of elements in $]x,y[$.

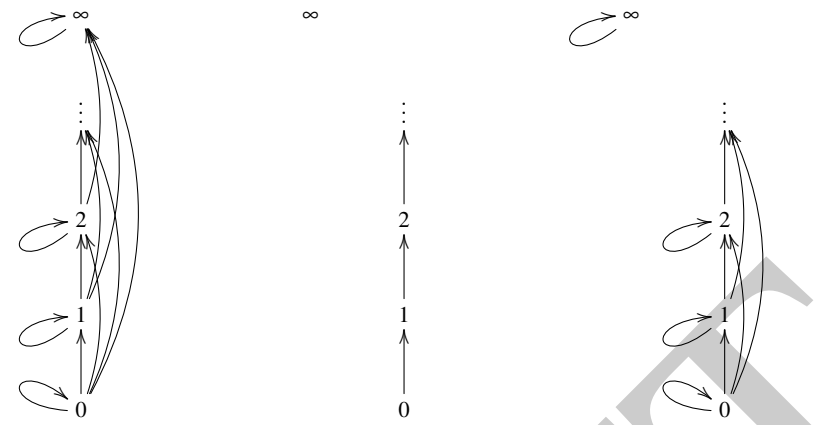
Base case: When $]x,y[$ is empty, it means that $(x,y) \notin R$. Hence $x \, H \, y$ and thus $x H^* y$.

Inductive case: Suppose $]x,y[$ has $n+1$ elements. Take $z \in]x,y[$. Clearly the sizes of $]x,z[$ and $]z,y[$ are strictly smaller than that of $]x,y[$, and since $x \sqsubset z$ and $z \sqsubset y$, by inductive hypothesis it follows that $x H^* z$ and $z H^* y$. Hence $x H^* y$. □

The above theorem only allows to represent finite orders.

Example 5.6 (Infinite order). Let us see that the Hasse diagrams does not work well with infinite orders. Let $\Omega = (\mathbb{N} \cup \{\infty\}, \leq)$ be the usual order on natural numbers extended with a top element ∞ such that $n \leq \infty$ and $\infty \leq \infty$, i.e.,

$$0 \leq 1 \leq 2 \leq \cdots \leq n \leq \cdots \leq \infty.$$



(a) $(\mathbb{N}\cup\{\infty\},\leq)$ (b) H induced by $(\mathbb{N}\cup\{\infty\},\leq)$ (c) H^* induced by $(\mathbb{N}\cup\{\infty\},\leq)$

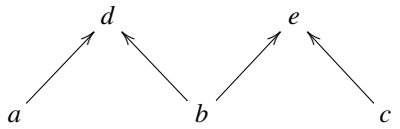
Fig. 5.2: Infinite orders and Hasse diagrams

From Definition 5.3 it follows that for any $n \in \mathbb{N}$ we have $n\,R\,\infty$ (because $n < n+1 < \infty$) and that for any $n,k \in \mathbb{N}$ it holds $n\,R\,n+2+k$ (because $n < n+1 < n+2+k$). Moreover, for any $x \in \mathbb{N}\cup\{\infty\}$ we have $x\,R\,x$. In particular, 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 original 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). The situation is illustrated in Figure 5.2

Definition 5.4 (Least element). Let $(P,\sqsubseteq_P)$ be a partial order and take $Q \subseteq P$. An element $\ell \in Q$ is a *least* element of $(Q,\sqsubseteq_Q)$ if:

$$\forall q \in Q.\ \ell \sqsubseteq_Q q$$

Example 5.7 (No least element). Let us consider the order associated with the Hasse diagram:



The sets $\{a,b,d\}$ and $\{a,b,c,d,e\}$ have 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 5.3 (Uniqueness of the least element). *Let $(P,\sqsubseteq)$ be a partial order. P has at most one least element.*

Proof. Let $\ell_1, \ell_2 \in P$ be both least elements of P, then $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_1$. Now by using the antisymmetric property we get $\ell_1 = \ell_2$. □

The counterpart of the least element is the concept of *greatest* element. We can define the greatest element as the least element of the reverse order $\sqsubseteq^{-1}$ (defined by letting $x \sqsubseteq^{-1} y \Leftrightarrow y \sqsubseteq x$).

Definition 5.5 (Minimal element). Let $(P, \sqsubseteq_P)$ be a partial order and take $Q \subseteq P$. An element $m \in Q$ is a *minimal element* of $(Q, \sqsubseteq_Q)$ if:

$$\forall q \in Q.\ q \sqsubseteq_Q m \Rightarrow q = m$$

As for the least element we have the dual of minimal elements, called *maximal elements*: They are the minimal elements of the reverse order $\sqsubseteq^{-1}$.

Remark 5.2 (Least vs minimal elements). Note that the definition of minimal and least element (maximal and greatest) are quite different.

- The least element ℓ is the (unique) smallest element of a set.
- A minimal element m is just such that no smaller element can be found in the set, i.e., $\forall q \in Q. q \not\sqsubseteq m$ (but there is no guarantee that all the elements $q \in Q$ are comparable with m).
- The least element of an order is obviously minimal, but a minimal element is not necessarily the least.

Definition 5.6 (Upper bound). Let $(P, \sqsubseteq)$ be a partial order and $Q \subseteq P$ be a subset of P, then $u \in P$ is an *upper bound* of Q if:

$$\forall q \in Q.\ q \sqsubseteq u$$

Note that unlike a maximal element and the greatest element an upper bound does not necessarily belong to the subset Q of elements we are considering.

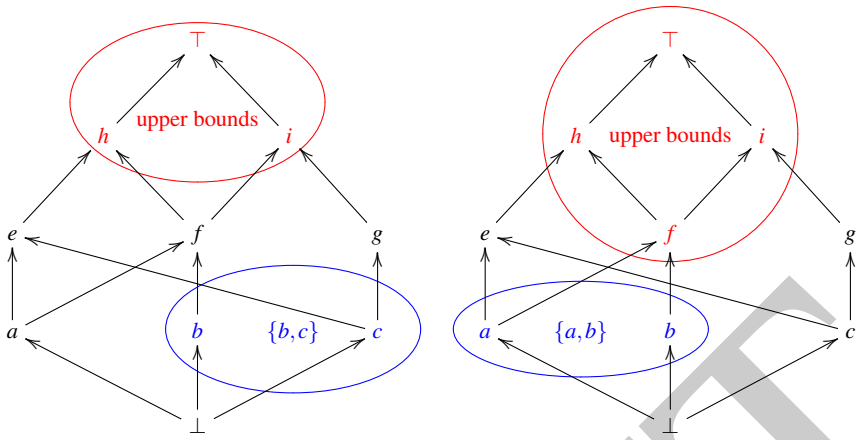
Definition 5.7 (Least upper bound). Let $(P, \sqsubseteq)$ be a partial order and $Q \subseteq P$ be a subset of P. Then, $p \in P$ is the *least upper bound* of Q if and only if p is the least element of the upper bounds of Q. Formally, we require that:

1. p is an upper bound of Q ($\forall q \in Q.\ q \sqsubseteq p$);
2. for any upper bound u of Q, then $p \sqsubseteq u$ ($\forall u \in P.\ (\forall q \in Q.\ q \sqsubseteq u) \Rightarrow p \sqsubseteq u$);

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

It follows immediately from Theorem 5.3 that the least upper bound, when it exists, is unique.

Example 5.8 (lub). Now we will clarify the concept of *lub* with two examples. Let us consider the order represented by the Hasse diagram in Figure 5.3 (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 comparable) so the set $\{b, c\}$ has no lub. In Figure 5.3 (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 is thus the lub of $\{a, b\}$.



(a) The subset $\{b,c\}$ has no lub

(b) The subset $\{a,b\}$ has lub f

Fig. 5.3: Two subsets of a poset, and their upper bounds

5.1.3 Chains

One of the main concept in the study of partial orders is that of a *chain*, which is formed by taking a subset of totally ordered elements.

Definition 5.8 (Chain). Let $(P,\sqsubseteq)$ be a partial order, we call *chain* a function $C : \mathbb{N} \to P$ such that:

$$\forall n \in \mathbb{N}.\ C(n) \sqsubseteq C(n+1)$$

We will often write $C = \{d_i\}_{i\in\mathbb{N}}$, where $\forall i \in \mathbb{N}.\ d_i = C(i)$, i.e.,

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

Definition 5.9 (Finite chain). Let $C : \mathbb{N} \to P$ be a chain such that the image of C is a finite set, then we say that C is a *finite chain*. Otherwise we say that C is infinite.

Note that a finite chain has still infinitely many elements $\{d_i\}_{i\in\mathbb{N}}$, but only finitely many different ones. In particular, it has one index k and one element d such that $\forall i \in \mathbb{N}.\ d_{k+i} = d$.

Example 5.9 (Finite and infinite chains). Take the partial order $(\mathbb{N},\leq)$. The chain of even numbers

$$0 \leq 2 \leq 4 \leq \cdots$$

is an infinite chain. Instead, the constant chain

$$1 \leq 1 \leq 1 \leq \cdots$$

is a finite chain.

Definition 5.10 (Limit of a chain). Let C be a chain. The lub of the image of C, if it exists, is called *the limit of* C. If d is the limit of the chain $C = \{d_i\}_{i\in\mathbb{N}}$, we write $d = \bigsqcup_{i\in\mathbb{N}} d_i$.

Remark 5.3. Each finite chain has a limit. Indeed each finite chain has a finite totally ordered image: obviously this set has a lub (the greatest element of the set).

Lemma 5.1 (Prefix independence of the limit). *Let $n \in \mathbb{N}$ and let C and C' be two chains such that $C = \{d_i\}_{i\in\mathbb{N}}$ and $C' = \{d_{n+i}\}_{i\in\mathbb{N}}$. Then C and C' have the same limit, if any.*

Proof. Let us prove a stronger property, namely that the chains C and C' 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.

Vice versa if c is an upper bound of C', we need to show that $\forall j \in \mathbb{N}.\ j \leq n \Rightarrow d_j \sqsubseteq c$. Since $d_n \sqsubseteq c$ and $\forall j \in \mathbb{N}.\ j \leq n \Rightarrow d_j \sqsubseteq d_n$ by transitivity of $\sqsubseteq$ it follows that 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. □

The main consequence of Lemma 5.1 is that we can always eliminate from or add a finite prefix to a chain preserving the limit.

A stronger result guarantees that any infinite subsequence of a chain C has the same set of upper bounds as C and thus the same limit, if any (see Problem 5.13).

5.1.4 Complete Partial Orders

The aim of partial orders and continuous functions is to provide a framework that allows the definition of the denotational semantics when recursive equations are needed. 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 fixpoint solutions to recursive equations.

Definition 5.11 (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).

From Remark 5.3, it follows immediately that if a partial order has only finite chains then it is complete.

Definition 5.12 (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. To avoid ambiguities, sometimes we will denote the bottom element of the CPO D by $\perp_D$.

Example 5.10 (Powerset completeness). Let us consider again the previous example of powerset (Example 5.1). We show that the partial order $(\wp(S), \subseteq)$ is complete. Take any chain $\{S_i\}_{i\in\mathbb{N}}$ of subsets of S. Then:

$$lub(S_0 \subseteq S_1 \subseteq S_2 \ldots) = \{d \mid \exists k \in \mathbb{N}.\, d \in S_k\} = \bigcup_{i\in\mathbb{N}} S_i \in \wp(S)$$

Example 5.11 (Partial order without upper bounds). Now let us take the usual order on natural numbers $(\mathbb{N}, \leq)$. Obviously all its finite chains have a limit (i.e., the greatest element of the chain). Vice versa 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. So we add the element ∞ and extend the order relation by letting $x \leq \infty$ for all $x \in \mathbb{N} \cup \{\infty\}$. The new poset $(\mathbb{N} \cup \{\infty\}, \leq)$ is a CPO, because ∞ is the limit of any infinite chain.

Remark 5.4 (A subset of a CPO is not necessarily a CPO). We have seen that when we restrict an order relation to a subset of elements we still get a PO (see Theorem 5.1). The previous example shows that, in general, the same property does not hold at the level of CPOs. The problem is due to the fact that, taken a chain whose elements are all in the subset, the lub of the chain is not necessarily in the subset.

Example 5.12 (Partial order without least upper bound). Let us define the partial order $(\mathbb{N} \cup \{\infty_1, \infty_2\}, \sqsubseteq)$ as follows:

$$(\sqsubseteq \restriction \mathbb{N}) = \leq, \quad \forall x \in \mathbb{N} \cup \{\infty_1\}.\, x \sqsubseteq \infty_1, \quad \forall x \in \mathbb{N} \cup \{\infty_2\}.\, x \sqsubseteq \infty_2$$

Where $\sqsubseteq \restriction \mathbb{N}$ 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 are not comparable, hence there is no least upper bound.

The next example illustrates a fundamental CPO, that will be exploited in the next chapters: the set of partial functions on natural numbers:

Example 5.13 (Partial functions). Let $\mathbf{Pf} \stackrel{\text{def}}{=} (\mathbb{N} \rightharpoonup \mathbb{N})$ be the set of partial functions from natural numbers to natural numbers. Recall that a partial function is a relation $f \subseteq \mathbb{N} \times \mathbb{N}$ with the *functional* property:

$$\forall n, m, k \in \mathbb{N}.\, n\, f\, m \wedge n\, f\, k \;\Rightarrow\; m = k$$

So the set **Pf** can be viewed as:

$$\mathbf{Pf} \stackrel{\text{def}}{=} \{f \subseteq \mathbb{N} \times \mathbb{N} \mid \forall n, m, k \in \mathbb{N}.\, n\, f\, m \wedge n\, f\, k \Rightarrow m = k\}$$

Let us denote by $f(n)\downarrow$ the predicate $\exists m \in \mathbb{N}.\, (n, m) \in f$ (i.e., $f(n)\downarrow$ holds when the function f is defined on n). Now it is easy to define a partial order $\sqsubseteq$ on **Pf**. We let:

$$f \sqsubseteq g \;\Leftrightarrow\; (\forall n \in \mathbb{N}.\, f(n)\downarrow \Rightarrow\; (g(n)\downarrow \wedge f(n) = g(n)))$$

Thus f precedes g if whenever f is defined on n also g is defined on n and $f(n) = g(n)$. When $f(n)$ is not defined, then $g(n)$ can be defined and take any value. When both f and g are seen as (functional) relations, then the above definition boils down to check that f is included in g. Of course, the poset $(\wp(\mathbb{N} \times \mathbb{N}), \subseteq)$ has the empty relation as bottom element (i.e., the function undefined everywhere), and each infinite chain has as limit the countable union of the relations in the chain.

To show that **Pf** is complete, we need to show that the limits of chains whose elements are in **Pf** satisfy also the functional property, i.e., they are elements of **Pf**.

Theorem 5.4. *Let $f_0 \subseteq f_1 \subseteq f_2 \subseteq \ldots$ be a chain in* **Pf***, i.e., each relation f_i satisfies the functional property, i.e.,*

$$\forall i \in \mathbb{N}.\ \forall n,m,k \in \mathbb{N}.\ n\, f_i\, m \wedge n\, f_i\, k \Rightarrow m = k.$$

Then, the relation $f \stackrel{\text{def}}{=} \cup_{i \in \mathbb{N}} f_i$ satisfies the functional property, namely:

$$\forall n,m,k \in \mathbb{N}.\ n\, f\, m \wedge n\, f\, k \Rightarrow m = k.$$

Proof. Let us take generic $n,m,k \in \mathbb{N}$ such that the premise $n\, f\, m \wedge n\, f\, k$ of the implication holds. We need to prove the consequence $m = k$. By $n\, f\, m$, it exists $j \in \mathbb{N}$ with $n\, f_j\, m$ and, by $n\, f\, k$ it exists $h \in \mathbb{N}$ with $n\, f_h\, k$. We take $o = \max\{j,h\}$ then it holds $n\, f_o\, m \wedge n\, f_o\, k$. Since $f_o \in$ **Pf**, it satisfies the functional property and thus from $n\, f_o\, m \wedge n\, f_o\, k$ we can conclude that $m = k$. □

Example 5.14 (Partial functions as total functions). Let us show a second way to define a CPO on the partial functions on natural numbers. Let $\mathbb{N}_\perp \stackrel{\text{def}}{=} \mathbb{N} \cup \{\perp\}$ and $(\mathbb{N}_\perp, \sqsubseteq_{\mathbb{N}_\perp})$ be the flat order obtained by adding $\perp$ to the discrete order of the natural numbers. In other words we have $x \sqsubseteq_{\mathbb{N}_\perp} y$ iff $x = y$ or $x = \perp$. Then take the set of total functions $\mathbf{Tf} = (\mathbb{N} \to \mathbb{N}_\perp)$. Equivalently:

$$\mathbf{Tf} \stackrel{\text{def}}{=} \{ f \subseteq \mathbb{N} \times \mathbb{N}_\perp \mid (\forall n,m,k \in \mathbb{N}.\ n\, f\, m \wedge n\, f\, k \Rightarrow m = k) \wedge (\forall n \in \mathbb{N}.\ \exists x \in \mathbb{N}_\perp.\ n\, f\, x) \}$$

We define the following order on **Tf**

$$f \sqsubseteq g \ \Leftrightarrow\ \forall n \in \mathbb{N}.\ f(n) \sqsubseteq_{\mathbb{N}_\perp} g(n).$$

That is, if $f(n) = \perp$ then $g(n)$ can assume any value, including $\perp$; otherwise it must be $g(n) = f(n)$. The bottom element of the order is the function that returns $\perp$ for every argument. Note that the above order 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. The proof is left as an exercise to the reader (see Problem 5.11).

Example 5.15 (Limit of a chain of partial functions). Let $\{f_i : \mathbb{N} \to \mathbb{N}_\perp\}_{i \in \mathbb{N}}$ be a chain in **Tf** such that for any $i \in \mathbb{N}$ we have:

$$f_i(n) \stackrel{\text{def}}{=} \begin{cases} 3 & \text{if } n \leq i \wedge 2 \mid n \\ \bot & \text{otherwise} \end{cases}$$

where the predicate $k \mid n$ is true when k divides n (i.e., $2 \mid n$ is true when n is even and false otherwise). Let us consider some evaluations of the functions f_i with $i \in [0,4]$:

$$\begin{array}{lllll} f_0(0)=3 & f_0(1)=\bot & f_0(2)=\bot & f_0(3)=\bot & f_0(4)=\bot \cdots \\ f_1(0)=3 & f_1(1)=\bot & f_1(2)=\bot & f_1(3)=\bot & f_1(4)=\bot \cdots \\ f_2(0)=3 & f_2(1)=\bot & f_2(2)=3 & f_2(3)=\bot & f_2(4)=\bot \cdots \\ f_3(0)=3 & f_3(1)=\bot & f_3(2)=3 & f_3(3)=\bot & f_3(4)=\bot \cdots \\ f_4(0)=3 & f_4(1)=\bot & f_4(2)=3 & f_4(3)=\bot & f_4(4)=3 \cdots \end{array}$$

Thus the limit of the chain is the function f that returns 3 when applied to even numbers and $\bot$ otherwise:

$$f(n) \stackrel{\text{def}}{=} \begin{cases} 3 & \text{if } 2 \mid n \\ \bot & \text{otherwise} \end{cases}$$

In general, the limit $f \stackrel{\text{def}}{=} \bigsqcup_{i \in \mathbb{N}} f_i$ of a chain in **Tf** is a function $f : \mathbb{N} \to \mathbb{N}_\bot$ such that $f(n) = m$ for some $m \neq \bot$ if and only if there exists an index $k \in \mathbb{N}$ with $f_k(n) = m$. Note also that when $i \leq j$ and $f_i(n) \neq \bot$ it must be the case that $f_j(n) = f_i(n)$. On the contrary, when $i \leq j$ and $f_j(n) = \bot$ it means that $f_i(n) = \bot$.

5.2 Continuity and Fixpoints

5.2.1 Monotone and Continuous Functions

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

Definition 5.13 (monotonicity). Let $f : D \to E$ be a function over two CPOs $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$, we say that f is *monotone* if

$$\forall d, d' \in D.\ d \sqsubseteq_D d' \Rightarrow f(d) \sqsubseteq_E f(d')$$

We say that a monotone function *preserves the order*. So if $\{d_i\}_{i \in \mathbb{N}}$ is a chain on $(D, \sqsubseteq_D)$ and $f : D \to E$ is a monotone function, then $\{f(d_i)\}_{i \in \mathbb{N}}$ is a chain on $(E, \sqsubseteq_E)$. Often we will consider functions whose domain and codomain coincide (i.e., $E = D$), in which case we just say that f is a function on $(D, \sqsubseteq_D)$.

Example 5.16 (Non monotone function). Let us define a CPO $(\{\bot, 0, 1\}, \sqsubseteq)$ such that $\bot \sqsubseteq 0$, $\bot \sqsubseteq 1$ and $x \sqsubseteq x$ for any $x \in \{\bot, 0, 1\}$. Now define a function f over $(\{\bot, 0, 1\}, \sqsubseteq)$ as follows:

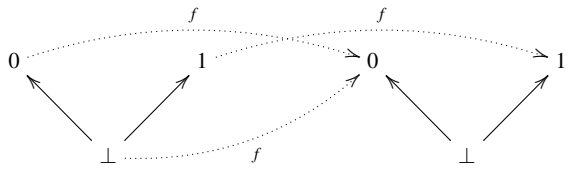


Fig. 5.4: A non monotone function

$$f(\bot) = 0 \quad f(0) = 0 \quad f(1) = 1$$

This function is not monotone, indeed $\bot \sqsubseteq 1$ but $f(\bot) = 0$ and $f(1) = 1$ are not comparable (see Figure 5.4), so the function f does not preserve the order.

Continuity guarantees that taking the image of the limit of a chain is the same as taking the limit of the images of the elements in the chain.

Definition 5.14 (Continuity). Let $(D, \sqsubseteq_D)$ and $(E \sqsubseteq_E)$ be two CPOs and let $f : D \to E$ be a monotone function. We say that f is a continuous function if for each chain in $(D, \sqsubseteq)$ we have:

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

As it is the case for most definitions of continuity, the operations of applying function and taking the limit can be exchanged. For this reason, we say that a continuous function *preserves limits*. Moreover, note that the limit $\bigsqcup_{i \in \mathbb{N}} d_i$ is taken in D, while the limit $\bigsqcup_{i \in \mathbb{N}} f(d_i)$ is taken in E.

Remark 5.5. Let $(D, \sqsubseteq)$ be a CPO that has only finite chains. Then any chain $\{d_i\}_{i \in \mathbb{N}}$ in D is such that there are $d \in D$ and $k \in \mathbb{N}$ such that $\forall i \in \mathbb{N}.\ d_{i+k} = d$ and it has a limit (d) that is also an element of the chain. Thus any monotone function $f : D \to E$ is continuous, because $\forall i \in \mathbb{N}.\ f(d_{i+k}) = f(d)$ (i.e., the chain $\{f(d_i)\}_{i \in \mathbb{N}}$ is finite and its limit is $f(d)$).

Interestingly, continuous functions are closed under composition.

Theorem 5.5 (Continuity of composition). *Let $(D, \sqsubseteq_D)$, $(E \sqsubseteq_E)$, and $(F \sqsubseteq_F)$ be three CPOs, and $f : D \to E$, $g : E \to F$ be two continuous functions. Their composition*

$$h \stackrel{\text{def}}{=} g \circ f : D \to F$$

defined by letting $h(d) = g(f(d))$ for all $d \in D$ is continuous.

Proof. Let $\{d_i\}_{i \in \mathbb{N}}$ be a chain in D. We want to prove that $h(\bigsqcup_{i \in \mathbb{N}} d_i) = \bigsqcup_{i \in \mathbb{N}} h(d_i)$. We have:

$$\begin{aligned} h(\bigsqcup_{i\in\mathbb{N}} d_i) &= g(f(\bigsqcup_{i\in\mathbb{N}} d_i)) && \text{by definition of } h = g\circ f \\ &= g(\bigsqcup_{i\in\mathbb{N}} f(d_i)) && \text{by continuity of } f \\ &= \bigsqcup_{i\in\mathbb{N}} g(f(d_i)) && \text{by continuity of } g \\ &= \bigsqcup_{i\in\mathbb{N}} h(d_i) && \text{by definition of } h = g\circ f \end{aligned}$$

□

Remark 5.6. The composition $g \circ f$ is sometimes denoted also by $f;g$.

Example 5.17 (A monotone function which is not continuous). Let $(\mathbb{N}\cup\{\infty\},\leq)$ be the CPO from Example 5.11. Define a function $f : \mathbb{N}\cup\{\infty\} \to \mathbb{N}\cup\{\infty\}$ such that:

$$f(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x \in \mathbb{N} \\ 1 & \text{if } x = \infty \end{cases}$$

It is immediate to check that f is monotone:

- for $n,m \in \mathbb{N}$, if $n \leq m$ we have $f(n) = 0 \leq 0 = f(m)$;
- for $n \in \mathbb{N}$, we have $n \leq \infty$ and $f(n) = 0 \leq 1 = f(\infty)$;
- for $\infty \leq \infty$ we have of course $f(\infty) \leq f(\infty)$.

Let us consider the chain $\{d_i\}_{i\in\mathbb{N}}$ of even numbers:

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

whose limit is ∞. The chain $\{f(d_i)\}_{i\in\mathbb{N}}$ is instead the constant chain

$$0 \leq 0 \leq 0 \leq 0 \leq \ldots$$

whose limit is 0. So we have

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

The monotone function f does not preserve the limits and thus it is not continuous.

5.2.2 Fixpoints

Now we are ready to study fixpoints of continuous functions.

Definition 5.15 (Pre-fixpoint and fixpoint). Let f be a continuous function over a $CPO_\perp$ $(D,\sqsubseteq)$. An element p is a *pre-fixpoint* if

$$f(p) \sqsubseteq p.$$

An element $d \in D$ is a *fixpoint* of f if

$$f(d) = d.$$

Of course any fixpoint of f is also a pre-fixpoint of f, i.e., the set of fixpoints of f is included in the set of its pre-fixpoints.

We will denote by $\mathrm{gfp}(f)$ the greatest fixpoint of f and by $\mathrm{lfp}(f)$ the least fixpoint of f, when they exist.

Let $f : D \to D$ and $d \in D$. We denote by $f^n(d)$ the repeated application of f to d for n times, i.e.,

$$f^0(d) \stackrel{\text{def}}{=} d$$
$$f^{n+1}(d) \stackrel{\text{def}}{=} f(f^n(d))$$

Lemma 5.2. *Let $(D, \sqsubseteq)$ be a partial order and $f : D \to D$ be a monotone function. The elements $\{f^n(\bot)\}_{n\in\mathbb{N}}$ form a chain in D.*

Proof. The property $\forall n \in \mathbb{N}.\ f^n(\bot) \sqsubseteq f^{n+1}(\bot)$ can be readily proved by mathematical induction on n.

Base case: For $n = 0$ we have $f^0(\bot) = \bot \sqsubseteq f^1(\bot) = f(\bot)$, as $\bot$ is the least element of D.

Inductive case: Let us assume that the property holds for n, i.e., that

$$f^n(\bot) \sqsubseteq f^{n+1}(\bot).$$

We want to prove that the property holds for $n+1$, i.e., that

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

In fact by definition we have $f^{n+1}(\bot) = f(f^n(\bot))$ and $f^{n+2}(\bot) = f(f^{n+1}(\bot))$. Since f is monotone and by the inductive hypothesis we have:

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

□

When $(D, \sqsubseteq)$ is complete then the chain $\{f^n(\bot)\}_{n\in\mathbb{N}}$ must have a limit $\bigsqcup_{n\in\mathbb{N}} f^n(\bot)$.

Next theorem ensures that the least fixpoint of a continuous function always exists and that it is computed by the above limit.

Theorem 5.6 (Kleene's Fixpoint theorem). *Let $f : D \to D$ be a continuous function on a CPO$_\bot$ D. Then, let*

$$\mathrm{fix}(f) = \bigsqcup_{n\in\mathbb{N}} f^n(\bot).$$

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

1. *$\mathrm{fix}(f)$ is a fixpoint of f, namely*

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

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

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

Since any fixpoint is a pre-fixpoint, $\mathrm{fix}(f)$ *is also the least fixpoint of* f*.*

Proof. We prove the two items separately.

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

$$\begin{aligned} f(\mathrm{fix}(f)) &= f(\bigsqcup_{n\in\mathbb{N}} f^n(\bot)) && \text{(by definition of fix)} \\ &= \bigsqcup_{n\in\mathbb{N}} f(f^n(\bot)) && \text{(by continuity of } f\text{)} \\ &= \bigsqcup_{n\in\mathbb{N}} f^{n+1}(\bot) && \text{(by definition of } f^{n+1}\text{)} \end{aligned}$$

   So we need to compute the limit of the chain:

$$f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq f^3(\bot) \sqsubseteq \ldots$$

   Since the limit is independent from any finite prefix of the chain, it coincides with the limit of the chain

$$f^0(\bot) = \bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq f^3(\bot) \sqsubseteq \ldots$$

$$\begin{aligned} \bigsqcup_{n\in\mathbb{N}} f^{n+1}(\bot) &= \bigsqcup_{n\in\mathbb{N}} f^n(\bot) && \text{(by Lemma 5.1)} \\ &= \mathrm{fix}(f) && \text{(by definition of fix)} \end{aligned}$$

2. We want to prove that $\mathrm{fix}(f)$ is the least pre-fixpoint. We prove that any pre-fixpoint of f is an upper bound of the chain $\{f^n(\bot)\}_{n\in\mathbb{N}}$. Let d be a pre-fixpoint of f, i.e.,

$$f(d) \sqsubseteq d \tag{5.1}$$

   By mathematical induction we show that

$$\forall n \in \mathbb{N}. f^n(\bot) \sqsubseteq d$$

   i.e., that d is an upper bound for the chain $\{f^n(\bot)\}_{n\in\mathbb{N}}$:

   base case: obviously $f^0(\bot) = \bot \sqsubseteq d$
   inductive case: let us assume $f^n(\bot) \sqsubseteq d$, we want to prove that $f^{n+1}(\bot) \sqsubseteq d$:

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

Since d is an upper bound for $\{f^n(\bot)\}_{n\in\mathbb{N}}$ and $\mathrm{fix}(f)$ is the limit (i.e., the least upper bound) of the same chain, it must be $\mathrm{fix}(f) \sqsubseteq d$.

□

Now let us make two examples which show that bottom element and the continuity property are required to compute the least fixpoint.

Example 5.18 (Bottom is necessary). Let $(\{\mathbf{true}, \mathbf{false}\}, \sqsubseteq)$ be the discrete order of boolean values. Obviously it is complete (because only finite chains of the form $x \sqsubseteq x \sqsubseteq x \sqsubseteq \dots$ exist) and it has no bottom element, as **true** and **false** are not comparable. All functions over such domain are continuous. The identity function has two fixpoints, but there is no least fixpoint. On the contrary, the negation function has no fixpoint.

Example 5.19 (Continuity is necessary). Let us consider the $\mathrm{CPO}_\bot$ $(\mathbb{N}\cup\{\infty_1,\infty_2\}, \sqsubseteq)$ where:

$$\sqsubseteq\restriction \mathbb{N} = \leq, \quad \forall d \in \mathbb{N}\cup\{\infty_1\}.\ d \sqsubseteq \infty_1, \quad \forall d \in \mathbb{N}\cup\{\infty_1,\infty_2\}.\ d \sqsubseteq \infty_2.$$

The bottom element is 0. We define a monotone function f as follows (see Figure 5.5):

$$f(n) \stackrel{\text{def}}{=} \begin{cases} n+1 & \text{if } n \in \mathbb{N} \\ \infty_2 & \text{otherwise} \end{cases}$$

Note that f is not continuous. Let us consider the chain of even numbers $\{d_i\}_{i\in\mathbb{N}}$. It follows that $\{f(d_i)\}_{i\in\mathbb{N}}$ is the chain of odd numbers. We have:

$$\bigsqcup_{i\in\mathbb{N}} d_i = \infty_1 \qquad \bigsqcup_{i\in\mathbb{N}} f(d_i) = \infty_1$$

Therefore:

$$f\left(\bigsqcup_{i\in\mathbb{N}} d_i\right) = f(\infty_1) = \infty_2 \neq \infty_1 = \bigsqcup_{i\in\mathbb{N}} f(d_i)$$

Note that f has only one fixpoint, indeed:

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

But such fixpoint is not reachable by taking $\bigsqcup_{n\in\mathbb{N}} f^n(0) = \infty_1$.

5.3 Immediate Consequence Operator

In this section we reconcile two different approaches for defining semantics: inference rules, like those used for defining the operational semantics of IMP, and fixpoint theory, that will be applied to define the denotational semantics of IMP. We show

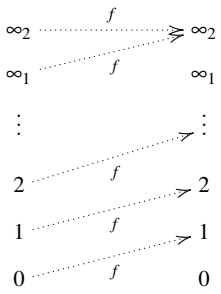


Fig. 5.5: Continuity is necessary

that the set of theorems of a logical system R can be defined as the least fixpoint of a suitable operator, called *immediate consequence operator* and denoted $\widehat{R}$.

5.3.1 The Operator $\widehat{R}$

Let us consider a set F of well-formed formulas and a set R of inference rules over them. We define an operator $\widehat{R}$ over $(\wp(F), \subseteq)$, the $CPO_\perp$ of sets of well-formed formulas ordered by inclusion.

Definition 5.16 (Immediate consequence operator $\widehat{R}$). Let R be a logical system. We define a function $\widehat{R} : \wp(F) \to \wp(F)$ as follows (for any $S \subseteq F$):

$$\widehat{R}(S) \stackrel{\text{def}}{=} \{y \mid \exists (X/y) \in R.\, X \subseteq S\}$$

The function $\widehat{R}$ is called *immediate consequence operator*.

The operator $\widehat{R}$, when applied to a set of well-formed formulas S, calculates a new set of formulas by applying the inference rules of R to the facts in S in all possible ways, i.e., $\widehat{R}(S)$ is the set of conclusions we can derive in one step from the hypothesis in S using rules in R. We will show that the set of theorems of R is equal to the least fixpoint of the immediate consequence operator $\widehat{R}$.

To apply the fixpoint theorem, we need to prove that $\widehat{R}$ is monotone and continuous.

Theorem 5.7 (Monotonicity of $\widehat{R}$). *$\widehat{R}$ is a monotone function.*

Proof. Let $S_1 \subseteq S_2$. We want to show that $\widehat{R}(S_1) \subseteq \widehat{R}(S_2)$. Let us assume $y \in \widehat{R}(S_1)$, then there exists a rule $(X/y) \in R$ with $X \subseteq S_1$. So we have $X \subseteq S_2$ and $y \in \widehat{R}(S_2)$. □

Theorem 5.8 (Continuity of $\widehat{R}$). *Let R be a logical system such that for any $(X/y) \in R$ the set of premises X is finite. Then $\widehat{R}$ is continuous.*

Proof. Let $\{S_i\}_{i\in\mathbb{N}}$ be a chain in $\wp(F)$. We want to prove that

$$\bigcup_{i\in\mathbb{N}} \widehat{R}(S_i) = \widehat{R}(\bigcup_{i\in\mathbb{N}} S_i).$$

As usual we prove the two inclusions separately:

$\subseteq$) Let $y \in \bigcup_{i\in\mathbb{N}} \widehat{R}(S_i)$ so there exists a natural number k such that $y \in \widehat{R}(S_k)$. Since

$$S_k \subseteq \bigcup_{i\in\mathbb{N}} S_i$$

by monotonicity

$$\widehat{R}(S_k) \subseteq \widehat{R}\left(\bigcup_{i\in\mathbb{N}} S_i\right)$$

hence $y \in \widehat{R}(\bigcup_{i\in\mathbb{N}} S_i)$.

$\supseteq$) Let $y \in \widehat{R}(\bigcup_{i\in\mathbb{N}} S_i)$ so there exists a rule $X/y \in R$ with $X \subseteq \bigcup_{i\in\mathbb{N}} S_i$. Since X is finite, there exists a natural number k such that $X \subseteq S_k$. In fact, for every $x \in X$ there will be a natural number k_x with $x \in S_{k_x}$ and letting $k = \max\{k_x\}_{x\in X}$ we have $X \subseteq S_k$. Since $X \subseteq S_k$ we have $y \in \widehat{R}(S_k) \subseteq \bigcup_{i\in\mathbb{N}} \widehat{R}(S_i)$ as required. □

5.3.2 Fixpoint of $\widehat{R}$

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

$$I_R \stackrel{\text{def}}{=} \bigcup_{i\in\mathbb{N}} I_R^i$$

where

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

Note that the generic I_R^n contains all theorems provable with derivations of depth[1] at most n, and I_R contains all theorems provable by using the rule system R.

Theorem 5.9. *Let R a rule system, it holds:*

$$\forall n \in \mathbb{N}.\ I_R^n = \widehat{R}^n(\varnothing)$$

Proof. By induction on n

[1] See Problem 4.12.

base case: $I_R^0 = \widehat{R}^0(\varnothing) = \varnothing$.
inductive case: We assume $I_R^n = \widehat{R}^n(\varnothing)$ and want to prove $I_R^{n+1} = \widehat{R}^{n+1}(\varnothing)$. Then:

$$\begin{aligned} I_R^{n+1} &= \widehat{R}(I_R^n) \cup I_R^n && \text{(by definition of } I_R^{n+1}) \\ &= \widehat{R}(\widehat{R}^n(\varnothing)) \cup \widehat{R}^n(\varnothing) && \text{(by inductive hypothesis)} \\ &= \widehat{R}^{n+1}(\varnothing) \cup \widehat{R}^n(\varnothing) && \text{(by definition of } \widehat{R}^{n+1}) \\ &= \widehat{R}^{n+1}(\varnothing) && \text{(because } \widehat{R}^{n+1}(\varnothing) \supseteq \widehat{R}^n(\varnothing)) \end{aligned}$$

In the last step of the proof we have exploited the property $\widehat{R}^{n+1}(\varnothing) \supseteq \widehat{R}^n(\varnothing)$, which is an instance of Lemma 5.2 (by taking $D = \wp(F)$, $\sqsubseteq = \subseteq$, $\bot = \varnothing$ and $f = \widehat{R}$). □

Theorem 5.10 (Fixpoint of $\widehat{R}$). *Let R a logical system, it holds:*

$$\mathrm{fix}(\widehat{R}) = I_R$$

Proof. By continuity of $\widehat{R}$ (Theorem 5.8) and the fixpoint theorem (Theorem 5.6), we know that the least fixpoint of $\widehat{R}$ exists and that

$$\mathrm{fix}(\widehat{R}) \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \widehat{R}^n(\varnothing)$$

Then, by Theorem 5.9:

$$I_R \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} I_R^n = \bigcup_{n \in \mathbb{N}} \widehat{R}^n(\varnothing) \stackrel{\text{def}}{=} \mathrm{fix}(\widehat{R})$$

as required. □

Example 5.20 (Rule system with discontinuous $\widehat{R}$). Let us consider the logical system R below:

$$\frac{\varnothing}{P(1)} \qquad \frac{P(x)}{P(x+1)} \qquad \frac{\forall n \in \mathbb{N}.\ P(1+2\times n)}{P(0)}$$

To ensure the continuity of $\widehat{R}$, Theorem 5.8 requires that the system has only rules with finitely many premises. The third rule of our system instead has infinitely many premises; it corresponds to

$$\frac{P(1) \quad P(3) \quad P(5) \quad \cdots}{P(0)}$$

The continuity of $\widehat{R}$, namely the fact that for all chains $\{S_i\}_{i \in \mathbb{N}}$ we have $\bigcup_{i \in \mathbb{N}} \widehat{R}(S_i) = \widehat{R}(\bigcup_{i \in \mathbb{N}} S_i)$, 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:

i	0		1		2	$\cdots$
S_i	$\{P(1)\}$	$\subseteq$	$\{P(1),P(3)\}$	$\subseteq$	$\{P(1),P(3),P(5)\}$	$\cdots$
$\widehat{R}(S_i)$	$\{P(1),P(2)\}$	$\subseteq$	$\{P(1),P(2),P(4)\}$	$\subseteq$	$\{P(1),P(2),P(4),P(6)\}$	$\cdots$

From which we get:

$$\bigcup_{i\in\mathbb{N}} S_i = \{P(1),P(3),P(5),\ldots\}$$
$$\widehat{R}(\bigcup_{i\in\mathbb{N}} S_i) = \{P(1),P(2),P(4),\ldots \underbrace{P(0)}_{\text{3rd rule}}\}$$
$$\bigcup_{i\in\mathbb{N}} \widehat{R}(S_i) = \{P(1),P(2),P(4),P(6),\ldots\}$$

because the third rule applies only when the predicate P holds for all the odd numbers, as in $\bigcup_{i\in\mathbb{N}} S_i$. Let us now compute the limit of $\widehat{R}$

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

In fact, we have:

$$\begin{aligned}
\widehat{R}^0(\varnothing) &= \varnothing \\
\widehat{R}^1(\varnothing) &= \{P(1)\} \\
\widehat{R}^2(\varnothing) &= \{P(1),P(2)\} \\
\widehat{R}^3(\varnothing) &= \{P(1),P(2),P(3)\} \\
&\cdots
\end{aligned}$$

But $\text{fix}(\widehat{R})$ is not a fixpoint of $\widehat{R}$, because $P(0) \notin \text{fix}(\widehat{R})$ but $P(0) \in \widehat{R}(\text{fix}(\widehat{R}))$!

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

Example 5.21 (Balanced parentheses). Let us consider the grammar for balanced parentheses, from Example 2.5

$$S \quad ::= \quad \varepsilon \quad | \quad (S) \quad | \quad SS$$

The corresponding logical system is:

$$\frac{\varnothing}{\varepsilon \in L_S} \qquad \frac{s \in L_S}{(s) \in L_S} \qquad \frac{s_1 \in L_S \quad s_2 \in L_S}{s_1 s_2 \in L_S}$$

So we can use the $\widehat{R}$ operator and the fixpoint theorem to find all the strings generated by the grammar by letting $L_S = \text{fix}(\widehat{R})$:

$$
\begin{aligned}
L_{S0} &= \widehat{R}^0(\varnothing) = \varnothing \\
L_{S1} &= \widehat{R}(S_0) = \{\varepsilon\} \\
L_{S2} &= \widehat{R}(S_1) = \{\varepsilon,\ (\,)\} \\
L_{S3} &= \widehat{R}(S_2) = \{\varepsilon,\ (\,),\ ((\,)),\ (\,)(\,)\} \\
&\ldots
\end{aligned}
$$

Problems

5.1. Prove Theorem 5.1. *Hint:* The proof is easy, because the axioms of partial and total orders are all universally quantified.

5.2. Let $(\wp(\mathbb{N}), \subseteq)$ be the CPO$_\perp$ of sets of natural numbers, ordered by inclusion. Assume a set $X \subseteq \mathbb{N}$ is fixed. Let $f, g : \wp(\mathbb{N}) \longrightarrow \wp(\mathbb{N})$ be the functions:

$$
\begin{aligned}
f(S) &\stackrel{\text{def}}{=} S \cap X \\
g(S) &\stackrel{\text{def}}{=} (\mathbb{N} \setminus S) \cap X
\end{aligned}
$$

1. Are f and g monotone?
2. Are they continuous?
3. Do the answers to the above questions depend on the given set X?

5.3. Define three functions $f_i : D_i \to D_i$ over three suitable CPO D_i for $i \in [1,3]$ (not necessarily with bottom) such that

1. f_1 is continuous, has fixpoints but not a least fixpoint;
2. f_2 is continuous and it has no fixpoint;
3. f_3 is monotone but not continuous.

5.4. Define a partial order $\mathscr{D} = (D, \sqsubseteq)$ that is not complete.

1. Let $x \prec y$ be irreflexive relation obtained by reversing the order, i.e.

$$x \prec y \quad \text{if and only if} \quad y \sqsubseteq x \wedge x \neq y.$$

   Is $\mathscr{D}' = (D, \prec)$ a well-founded relation?
2. In general, is it possible that $\mathscr{D}'$ is well-founded for some $\mathscr{D}$?

5.5. Let $V^* \cup V^\infty$ be the set of finite (V^*) and infinite (V^∞) strings over the alphabet $V = \{a, b, c\}$, and let $\alpha \sqsubseteq \alpha\beta$, where juxtaposition in $\alpha\beta$ denotes string concatenation and $\alpha\beta = \alpha$ if α is infinite.

1. Is the structure $(V^* \cup V^\infty, \sqsubseteq)$ a partial order?
2. If yes, is it a complete partial order?
3. Does there exist a bottom element?

4. Which are the maximal elements?

5.6. Let $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$ be two CPOs such that $D_1, D_2 \subseteq D$. Consider the structures:

- $(D_1 \cup D_2\ ,\ \sqsubseteq)$, where $x \sqsubseteq y$ iff $x \sqsubseteq_1 y \vee x \sqsubseteq_2 y$
- $(D_1 \cap D_2\ ,\ \preceq)$, where $x \preceq y$ iff $x \sqsubseteq_1 y \wedge x \sqsubseteq_2 y$

1. Are they always partial orders?
2. If so, are they complete?

In case of negative answers, exhibit some counterexample.

5.7. Let X and Y be sets and $X_\perp$ and $Y_\perp$ be the corresponding flat domains. Show that a function $f : X_\perp \to Y_\perp$ is continuous if and only if one, or both, of the following conditions holds:

1. f is *strict*, i.e., $f(\perp) = \perp$.
2. f is constant.

5.8. Let $\{\top\}$ be a one-element set and $\{\top\}_\perp$ the corresponding flat domain. Let Ω be the domain of *vertical natural numbers* (see Examples 5.6 and 5.11)

$$0 \leq 1 \leq 2 \leq 3 \leq \ldots \leq \infty.$$

Show that the set of continuous functions from Ω to $\{\top\}_\perp$ is in bijection with Ω.
Hint: Define what the possible continuous functions from Ω to $\{\top\}_\perp$ are.

5.9. Let $D = \{n \in \mathbb{N} \mid n > 0\} \cup \{\infty_0\}$ and $\sqsubseteq$ be the relation over D such that:

- for any pair of natural numbers $n, m \in D$, we let $n \sqsubseteq m$ iff n divides m;
- for any $x \in D$, we let $x \sqsubseteq \infty_0$.

Is $(D, \sqsubseteq)$ a CPO$_\perp$? Explain.

5.10. Consider the set $\mathbb{N} \times \mathbb{N}$ of pairs of natural numbers with the lexicographic order relation $\sqsubseteq$ defined by letting:

$$(n, m) \sqsubseteq (n', m') \quad \text{if} \quad n < n' \vee (n = n' \wedge m < m')$$

1. Prove that $\sqsubseteq$ is a partial order with bottom.
2. Show that the chain $\{(0, k)\}_{k \in \mathbb{N}}$ has a lub.
3. Exhibit a chain without lub.
4. Consider the subset $[0, n] \times \mathbb{N}$, with the same order, and then show, also in this case, a chain without lub .
5. Finally, prove that $[0, n] \times (\mathbb{N} \cup \infty)$ with the same order (where $x \leq \infty$ for any $x \in \mathbb{N}$), is complete with bottom, and show a monotone, non continuous function on it.

5.11. Prove that the set **Tf** of total functions from $\mathbb{N}$ to $\mathbb{N}_\perp$ defined in Example 5.14 forms a complete partial order.

5.12. Consider the set **PI** of partial injective functions from $\mathbb{N}$ to $\mathbb{N}$. A partial injective function f can be seen as a relation $\{(x,y) \mid x,y \in \mathbb{N} \wedge y = f(x)\} \subseteq \mathbb{N} \times \mathbb{N}$ such that

- $(x,y),(x,y') \in f$ implies $y = y'$, (i.e., f is a partial function), and
- $(x,y),(x',y) \in f$ implies $x = x'$, (i.e., f is injective).

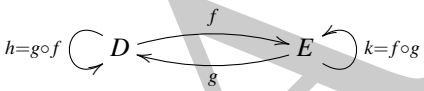
Accordingly, the elements of **PI** can be ordered by inclusion.

1. Prove that $(\mathbf{PI}, \subseteq)$ is a complete partial order.
2. Prove that the function $F : \mathbf{PI} \to \mathbf{PI}$ with $F(f) = \{(2 \times x, y) \mid (x,y) \in f\}$ is monotone and continuous.
   (Hint: Consider F as computed by the immediate consequences operator $\widehat{R}$, with R consisting only of the rule $(x,y)/(2 \times x, y)$.)

5.13. Let $(D, \sqsubseteq)$ be a CPO, $\{d_i\}_{i \in \mathbb{N}}$ a chain in D and $f : \mathbb{N} \to \mathbb{N}$ a function such that for all $i, j \in \mathbb{N}$ if $i < j$ then $f(i) < f(j)$. Prove that:

$$\bigsqcup_{i \in \mathbb{N}} d_{f(i)} = \bigsqcup_{i \in \mathbb{N}} d_i.$$

5.14. Let D, E be two $\mathrm{CPO}_\perp$ and $f : D \to E$, $g : E \to D$ be two continuous functions between them. Their compositions $h = g \circ f : D \to D$ and $k = f \circ g : E \to E$ are known to be continuos and thus have least fixpoints.



Let $e_0 = \text{fix}\, k \in E$. Prove that $g(e_0) = \text{fix}\, h \in D$ by showing that:

1. $g(e_0)$ is a fixpoint for h, and
2. that $g(e_0)$ is the least pre-fixpoint for h.

Chapter 6
Denotational Semantics of IMP

The point is that, mathematically speaking, functions are independent of their means of computation and hence are "simpler" than the explicitly generated, step-by-step evolved sequences of operations on representations. (Dana Scott)

Abstract In this chapter we give a more abstract, purely mathematical semantics to IMP, called *denotational semantics*. The operational semantics is close 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. The denotational semantics takes a program and returns the transformation function over memories associated with that program: given an initial state as the argument, the final state is returned as the 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). We conclude the chapter by reconciling the equivalences induced by the operational and the denotational semantics and by stating the principle of computational induction.

6.1 λ-Notation

In the following we shall rely on λ*-notation* as a *(meta-)language* for writing anonymous functions. When considering HOFL, then λ-notation will be used both at the level of the programming language and at the level of the denotational semantics, as meta-language.

The λ-calculus was introduced by Alonzo Church (1903-1995) in order to answer one of the questions posed by David Hilbert (1862–1943) in his program, known as Entscheidungsproblem (German for *decision problem*). Roughly, the problem consisted in the existence of an algorithm to decide whether a given statement of a first-order logic (possibly enriched with a finite number of axioms) is deducible or not from the axioms of logic. Alan Turing (1912-1954) proved that no effectively calculable algorithm can exist that solves the problem, where "calculable" meant

computable by a Turing machine. Independently, Alonzo Church answered negatively assuming that "calculable" meant a function expressible in the λ-calculus.

6.1.1 λ-Notation: Main Ideas

The λ-calculus is built around the idea of expressing a calculus of functions, where it is not necessary to assign names to functions, i.e., where functions can be expressed anonymously. Conceptually, this amounts to have the possibility of:

- forming (anonymous) functions by *abstraction* over names in an expression; and
- *applying* a function to an argument

Building on the two basic considerations above, Church developed a theory of functions based on rules for computation, as opposed to the classical set-theoretic view of functions as sets of pairs (argument, result).

Example 6.1. Let us start with a simple example from arithmetic. Take a polynomial such as

$$x^2 - 2x + 5.$$

What is the value of the above expression when x is replaced by 2? We compute the result by plugging in '2' for 'x' in the expression to get

$$2^2 - 2 \times 2 + 5 = 5.$$

In λ-notation, when we want to express that the value of an expression depends on some value to be plugged in, we use abstraction. Syntactically, this corresponds to prefix the expression by the special symbol λ and the name of the formal parameter, as, e.g., in:

$$\lambda x.\ (x^2 - 2x + 5)$$

The informal reading is:

> wait for a value v to replace x and then compute $v^2 - 2v + 5$.

We want to be able to pass some actual parameter to the function above, i.e., to apply the function to some value v. To this aim, we denote application by juxtaposition:

$$(\lambda x.\ (x^2 - 2x + 5))\ 2$$

means that the function $(\lambda x.\ (x^2 - 2x + 5))$ is applied to 2 (i.e., that the actual parameter 2 must replace the occurrences of the formal parameter x in $x^2 - 2x + 5$, to obtain $2^2 - 2 \times 2 + 5 = 5$.)

Note that:

- by writing $\lambda x.\ t$ we are declaring x as a formal parameter appearing in t;
- the symbol λ has no particular meaning (any other symbol could have been used);

- we say that λx 'binds' the (occurrences of the) variable x in t;
- the scope of the formal parameter x is just t; if x occurs also "outside" t, then it refers to another (homonymous) identifier.

Example 6.2. Let us consider another example:

$$(\lambda x.\ \lambda y.\ (x^2 - 2y + 5))\ 2$$

This time we have a function that is waiting for two arguments (first x, then y), but to which we pass one value (2). We have

$$(\lambda x.\ \lambda y.\ (x^2 - 2y + 5))\ 2 = \lambda y.\ (2^2 - 2y + 5) = \lambda y.\ (9 - 2y)$$

that is, the result of applying $\lambda x.\ \lambda y.\ (x^2 - 2y + 5)$ to 2 is still a function $(\lambda y.\ (9 - 2y))$.

In λ-calculus we can pass functions as arguments and return functions as results.

Example 6.3. Take the term $\lambda f.\ (f\ 2)$: it waits for a function f that will be applied to the value 2. If we pass the function $(\lambda x.\ \lambda y.\ (x^2 - 2y + 5))$ to $\lambda f.\ (f\ 2)$, written:

$$(\lambda f.\ (f\ 2))\ (\lambda x.\ \lambda y.\ (x^2 - 2y + 5))$$

then we get the function $\lambda y.\ (9 - 2y)$ as a result.

Definition 6.1 (Lambda terms). We define *lambda terms* as the terms generated by the grammar:

$$t \quad ::= \quad x \quad | \quad \lambda x.\, t \quad | \quad (t_0\ t_1) \quad | \quad t \to (t_0, t_1)$$

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.
- $t_0\ t_1$: is the *application* of a function t_0 to its argument t_1.
- $t \to t_0, t_1$ is the conditional operator, i.e. the "if-then-else" construct in lambda notation.

Note that we omit some parentheses when no ambiguity can arise.

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.

Note that while we can have different terms t and t' that define the same function, Church proved that the problem of deciding whether $t = t'$ is undecidable.

Definition 6.2 (Conditional expressions). Let t, t_0 and t_1 be three lambda terms, we define:

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

All the notions used in this definition, like "true" and "false" can be formalised in lambda notation only, by using lambda abstraction, as shown in Section 6.1.1.1 for the interested reader. In the following we will take the liberty to assume that data types such as integers and booleans are available in the lambda-notation as well as the usual operations on them.

Remark 6.1 (Associativity of abstraction and application). In the following, to limit the number of parentheses and keep the notation more readable, we assume that application is left-associative, and lambda-abstraction is right-associative, i.e.,

$$\begin{array}{ll} t_1\ t_2\ t_3\ t_4 & \text{is read as } (((t_1\ t_2)\ t_3)\ t_4) \\ \lambda x_1.\ \lambda x_2.\ \lambda x_3.\ \lambda x_4.\ t & \text{is read as } \lambda x_1.\ (\lambda x_2.\ (\lambda x_3.\ (\lambda x_4.\ t))) \end{array}$$

Remark 6.2 (Precedence of application). We will also assume that application has precedence over abstraction, i.e.:

$$\lambda x.\ t\ t' = \lambda x.\ (t\ t')$$

6.1.1.1 λ-Notation: Booleans and Church Numerals

In the above examples, we have enriched standard arithmetic expressions with abstraction and application. In general, it would be possible to encode booleans and numbers (and operations over them) just using abstraction and application.

For example, let us consider the following terms:

$$T \stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ x$$
$$F \stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ y$$

We can assume that T represents true and F represents false.

Under this convention, we can define the usual logical operations by letting:

$$\text{AND} \stackrel{\text{def}}{=} \lambda p.\ \lambda q.\ p\ q\ p$$
$$\text{OR} \stackrel{\text{def}}{=} \lambda p.\ \lambda q.\ p\ p\ q$$
$$\text{NOT} \stackrel{\text{def}}{=} \lambda p.\ \lambda x.\ \lambda y.\ p\ y\ x$$

Now suppose that P will reduce either to T or to F. The expression $P\ A\ B$ can be read as 'if P then A else B'.

For natural numbers, we can adopt the convention that the number n is represented by a function that takes a function f and an argument x and applies f to x for n times consecutively. For example:

$$
\begin{aligned}
0 &\stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ x \\
1 &\stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ f\ x \\
2 &\stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ f\ (f\ x) \\
&\dots
\end{aligned}
$$

Then, the operations for successor, sum, multiplication can be defined by letting:

$$
\begin{aligned}
\text{SUCC} &\stackrel{\text{def}}{=} \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x) \\
\text{SUM} &\stackrel{\text{def}}{=} \lambda n.\ \lambda m.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x) \\
\text{MUL} &\stackrel{\text{def}}{=} \lambda n.\ \lambda m.\ \lambda f.\ n\ (m\ f)
\end{aligned}
$$

6.1.2 Alpha-Conversion, Beta-Rule and Capture-Avoiding Substitution

The names of the formal parameters we choose for a given function should not matter. Therefore, any two expressions that differ just for the particular choice of λ-abstracted variables and have the same structure otherwise, should be considered as equal.

For example, we do not want to distinguish between the terms

$$\lambda x.\ (x^2 - 2x + 5) \qquad \lambda y.\ (y^2 - 2y + 5)$$

On the other hand, the expressions

$$x^2 - 2x + 5 \qquad y^2 - 2y + 5$$

must be distinguished, because depending on the context where they are used, the symbols x and y could have a different meaning.

We say that two terms are α-convertible if one is obtained from the other by renaming some λ-abstracted variables. We call *free* the variables x whose occurrences are not under the scope of a λ binder.

Definition 6.3 (Free variables). The set of free variables occurring in a term is defined by structural recursion:

$$
\begin{aligned}
\text{fv}(x) &\stackrel{\text{def}}{=} \{x\} \\
\text{fv}(\lambda x.\ t) &\stackrel{\text{def}}{=} \text{fv}(t) \setminus \{x\} \\
\text{fv}(t_0\ t_1) &\stackrel{\text{def}}{=} \text{fv}(t_0) \cup \text{fv}(t_1) \\
\text{fv}(t \to t_0, t_1) &\stackrel{\text{def}}{=} \text{fv}(t) \cup \text{fv}(t_0) \cup \text{fv}(t_1)
\end{aligned}
$$

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

Definition 6.4 (Alpha-conversion). We define *α-conversion* as the equivalence induced by letting

$$\lambda x.\, t = \lambda y.\, (t[^{y}/_{x}]) \quad \text{if } y \notin \mathrm{fv}(t)$$

where $t[^{y}/_{x}]$ denotes the substitution of x with y applied to the term t.

Note the side condition $y \notin \mathrm{fv}(t)$, which is needed to avoid 'capturing' other free variables appearing in t.

For example:

$$\lambda z.\, z^2 - 2y + 5 = \lambda x.\, x^2 - 2y + 5 \neq \lambda y.\, y^2 - 2y + 5$$

We have now all ingredients to define the basic computational rule, called β-rule, which explains how to apply a function to an argument:

Definition 6.5 (Beta-rule). Let t, t' be two lambda terms we define:

$$(\lambda x.\, t')\, t = t'[^{t}/_{x}]$$

this axiom is called β-rule.

In defining alpha-conversion and the beta-rule we have used substitutions like $[^{y}/_{x}]$ and $[^{t}/_{x}]$. Let us now try to formalise the notion of substitution by structural recursion. What is wrong with the following naive attempt?

$$y[^{t}/_{x}] \stackrel{\text{def}}{=} \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases}$$

$$(\lambda y.\, t')[^{t}/_{x}] \stackrel{\text{def}}{=} \begin{cases} \lambda y.\, t' & \text{if } y = x \\ \lambda y.\, (t'[^{t}/_{x}]) & \text{if } y \neq x \end{cases}$$

$$(t_0\ t_1)[^{t}/_{x}] \stackrel{\text{def}}{=} (t_0[^{t}/_{x}])\ (t_1[^{t}/_{x}])$$

$$(t' \to t_0, t_1)[^{t}/_{x}] \stackrel{\text{def}}{=} (t'[^{t}/_{x}]) \to (t_0[^{t}/_{x}]), (t_1[^{t}/_{x}])$$

Example 6.4 (Substitution, without alpha-renaming). Consider the terms

$$t \stackrel{\text{def}}{=} \lambda x.\, \lambda y.\, (x^2 - 2y + 5) \qquad t' \stackrel{\text{def}}{=} y.$$

and apply t to t':

$$\begin{aligned} t\ t' &= (\lambda x.\, \lambda y.\, (x^2 - 2y + 5))\ y \\ &= (\lambda y.\, (x^2 - 2y + 5))[^{y}/_{x}] \\ &= \lambda y.\, ((x^2 - 2y + 5)[^{y}/_{x}]) \\ &= \lambda y.\, (y^2 - 2y + 5) \end{aligned}$$

It happens that the free variable $y \in \mathrm{fv}(t\ t')$ has been 'captured' by the lambda-abstraction λy. Instead, free variables occurring in t should remain free during the application of the substitution $[^{t}/_{x}]$.

Thus we need to correct the above version of substitution for the case related to $(\lambda y.\ t')[{}^{t}/_{x}]$ by applying first the alpha-conversion to $\lambda y.\ t'$ (to make sure that if $y \in \mathrm{fv}(t)$, then the free occurrences of y in t will not be captured by λy when replacing x in t') and then the substitution $[{}^{t}/_{x}]$. Formally, we let:

Definition 6.6 (Capture-avoiding substitution). Let t, t', t_0 and t_1 be four lambda terms, we define:

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

Note that the matter of names is not so trivial. In the second equation we first rename y in t' with a fresh name z, then proceed with the substitution of x with t. As explained, 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 alpha-conversion equivalence, as previously introduced. 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 $\lambda x.\ x = \lambda y.\ y$.

Example 6.5 (Application with alpha-renaming). Consider the terms t, t' from Example 6.4:

$$
\begin{aligned}
t\ t' &= (\lambda x.\ \lambda y.\ (x^2 - 2y + 5))\ y \\
&= (\lambda y.\ (x^2 - 2y + 5))[{}^{y}/_{x}] \\
&= \lambda z.\ ((x^2 - 2y + 5)[{}^{z}/_{y}][{}^{y}/_{x}]) \\
&= \lambda z.\ ((x^2 - 2z + 5)[{}^{y}/_{x}]) \\
&= \lambda z.\ (y^2 - 2z + 5)
\end{aligned}
$$

Finally we introduce some notational conventions for omitting parentheses when defining the domains and codomains of functions:

$$
\begin{aligned}
A \to B \times C &= A \to (B \times C) & \qquad A \times B \times C &= (A \times B) \times C \\
A \times B \to C &= (A \times B) \to C & \qquad A \to B \to C &= A \to (B \to C)
\end{aligned}
$$

6.2 Denotational Semantics of IMP

As we said we will use lambda notation as meta-language; this means that we will express the semantics of IMP by translating IMP syntax to lambda terms.

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

Aexp: each arithmetic expression is mapped to a function from states to integers:

$$\mathscr{A} : Aexp \to (\Sigma \to \mathbb{Z})$$

Bexp: each boolean expression is mapped to a function from states to booleans:

$$\mathscr{B} : Bexp \to (\Sigma \to \mathbb{B})$$

Com: each command is mapped to a (partial) function from states to states:

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

6.2.1 Denotational Semantics of Arithmetic Expressions: The Function $\mathscr{A}$

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

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

We shall define $\mathscr{A}$ by structural recursion over the syntax of arithmetic expressions. Let us fix some notation: We will rely on definitions of the form

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

with the following meaning:

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

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

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

instead of

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

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

Definition 6.7 (Denotational semantics of arithmetic expressions). The denotational semantics of arithmetic expressions is defined by structural recursion as:

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

Let us briefly comment on the above definitions.

Constants:	The denotational semantics of any constant n is just the constant function that always returns n for any σ.
Variables:	The denotational semantics of any variable x is the function that takes a memory σ and returns the value of x in σ.
Binary expressions:	The denotational semantics of any binary expression evaluates the arguments (with the same given σ) and combines the results by exploiting the corresponding arithmetic operation.

Note that the symbols $+$, $-$ and $\times$ are overloaded: in the left hand side they represent elements of the syntax, while in the right hand side they represent operators of the metalanguage. Similarly for the symbol n in the first equation.

6.2.2 Denotational Semantics of Boolean Expressions: The Function $\mathscr{B}$

The denotational semantics of boolean expression is given by a function $\mathscr{B}$ defined in a very similar way to $\mathscr{A}$. The only difference is that the values to be returned are elements of $\mathbb{B}$ and not of $\mathbb{Z}$ and that $\mathscr{B}$ is not always defined in terms of itself: some defining equations exploit the function $\mathscr{A}$.

Definition 6.8 (Denotational semantics of boolean expressions). The denotational semantics of boolean expressions is defined by structural recursion as follows:

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

6.2.3 Denotational Semantics of Commands: The Function $\mathscr{C}$

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

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

Since commands can diverge, the codomain of $\mathscr{C}$ is the set of partial functions from memories to memories. As we have discussed in Example 5.14, for each partial function we can define an equivalent total function. So we define:

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

This will simplify the notation.

Instead of presenting the whole, structurally recursive, definition of $\mathscr{C}$ and then commenting the defining equations, we give each rule separately accompanied by the necessary explanations.

We start from the simplest commands: skip and assignments.

$$\mathscr{C}[\![\textbf{skip}]\!]\,\sigma \stackrel{\text{def}}{=} \sigma \tag{6.1}$$

We see that $\mathscr{C}[\![\textbf{skip}]\!]$ is the identity function: **skip** does not modify the memory.

$$\mathscr{C}[\![x := a]\!]\,\sigma \stackrel{\text{def}}{=} \sigma[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}] \tag{6.2}$$

The denotational semantics of the assignment evaluates the arithmetic expression a via $\mathscr{A}$ and then modifies the memory assigning the corresponding value to the location x.

Let us now consider the sequential composition of two commands. In interpreting $c_0; c_1$ we first interpret c_0 in the starting memory and then c_1 in the state produced by c_0. The problem is that from the first application of $\mathscr{C}[\![c_0]\!]$ we obtain a value in $\Sigma_\perp$, not necessarily in Σ, so we can not apply $\mathscr{C}[\![c_1]\!]$. To work this problem out we introduce a *lifting* operator $(\cdot)^*$: it takes a function in $\Sigma \to \Sigma_\perp$ and returns a function in $\Sigma_\perp \to \Sigma_\perp$, i.e., its type is $(\Sigma \to \Sigma_\perp) \to (\Sigma_\perp \to \Sigma_\perp)$.

Definition 6.9 (Lifting). Let $f : \Sigma \to \Sigma_\perp$, we define a function $f^* : \Sigma_\perp \to \Sigma_\perp$ as follows:

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

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

$$\mathscr{C}[\![c_0; c_1]\!]\,\sigma \stackrel{\text{def}}{=} \mathscr{C}[\![c_1]\!]^*\,(\mathscr{C}[\![c_0]\!]\,\sigma) \tag{6.3}$$

Note that we apply the lifted version $\mathscr{C}[\![c_1]\!]^*$ of $\mathscr{C}[\![c_1]\!]$ to the argument $\mathscr{C}[\![c_0]\!]\,\sigma$.

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

$$\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma \stackrel{\text{def}}{=} \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma \tag{6.4}$$

The definition of the denotational semantics of the while command is more intricate. We could think to define the interpretation simply as:

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

Obviously this definition is not a structural recursive, because the same expression $\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]$ whose meaning we want to define appears in the right-hand side of the defining equation. Indeed structural recursion allows only for the presence of subterms in the right-hand side, like $\mathscr{B}[\![b]\!]$ and $\mathscr{C}[\![c]\!]$. To solve this issue we will reduce the problem of defining the semantics of iteration to a fixpoint calculation. Let us define a function $\Gamma_{b,c} : (\Sigma \to \Sigma_\perp) \to \Sigma \to \Sigma_\perp$:

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

The function $\Gamma_{b,c}$ takes a function $\varphi : \Sigma \to \Sigma_\perp$, and returns the function

$$\lambda\sigma.\ \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![c]\!]\sigma), \sigma$$

of type $\Sigma \to \Sigma_\perp$, which given a memory σ evaluates $\mathscr{B}[\![b]\!]\sigma$ and depending on the outcome returns either $\varphi^*(\mathscr{C}[\![c]\!]\sigma)$ or σ. Note that the definition of $\Gamma_{b,c}$ refers only to subterms of the command **while** b **do** c. Clearly we require that $\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]$ is a fixpoint of $\Gamma_{b,c}$, i.e., that:

$$\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!] = \Gamma_{b,c}\,\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]$$

As there can be several fixpoints for $\Gamma_{b,c}$, we define $\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]$ as the least one. Next we show that $\Gamma_{b,c}$ is a monotone and continuous function, so that we can prove that $\Gamma_{b,c}$ has a least fixpoint and that by the fixpoint Theorem 5.6:

$$\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!] \stackrel{\text{def}}{=} \text{fix } \Gamma_{b,c} = \bigsqcup_{n\in\mathbb{N}} \Gamma^n_{b,c}(\perp_{\Sigma\to\Sigma_\perp}) \tag{6.5}$$

To prove continuity we will consider $\Gamma_{b,c}$ as operating on partial functions:

$$\Gamma_{b,c} : (\Sigma \rightharpoonup \Sigma) \longrightarrow (\Sigma \rightharpoonup \Sigma).$$

Partial functions in $\Sigma \rightharpoonup \Sigma$ can be represented as sets of pairs (σ, σ') that we write as formulas $\sigma \mapsto \sigma'$. Then the effect of $\Gamma_{b,c}$ can be represented by the immediate consequence operators for the following set of rules.

$$R_{\Gamma_{b,c}} \stackrel{\text{def}}{=} \left\{ \frac{\mathscr{B}[\![b]\!]\sigma \quad \mathscr{C}[\![c]\!]\sigma = \sigma'' \quad \sigma'' \mapsto \sigma'}{\sigma \mapsto \sigma'}, \qquad \frac{\neg\mathscr{B}[\![b]\!]\sigma}{\sigma \mapsto \sigma} \right\}$$

Note that there are infinitely many instance of the rules, but each rule has only a finite number of premises and that

$$\widehat{R}_{\Gamma_{b,c}} = \Gamma_{b,c}.$$

The only formulas appearing in the rules are $\sigma'' \mapsto \sigma'$ (as a premise of the first rule), $\sigma \mapsto \sigma'$ and $\sigma \mapsto \sigma$ (as conclusions); the other formulas express side-conditions: $\mathscr{B}[\![b]\!]\sigma \wedge \mathscr{C}[\![c]\!]\sigma = \sigma''$ for the first rule and $\neg\mathscr{B}[\![b]\!]\sigma$ for the second rule. An instance of the first rule schema is obtained by picking up two memories σ and σ'' such that $\mathscr{B}[\![b]\!]\sigma$ is true and $\mathscr{C}[\![c]\!]\sigma = \sigma''$. Then for every σ' such that $\sigma'' \mapsto \sigma'$ we can derive $\sigma \mapsto \sigma'$. The second rule schema is an axiom expressing that $\sigma \mapsto \sigma$ whenever $\neg\mathscr{B}[\![b]\!]\sigma$.

Since all the rules obtained in this way have a finite number of premises (actually one or none), we can apply Theorem 5.8, which ensures the continuity of $\widehat{R}_{\Gamma_{b,c}}$. Now by using Theorem 5.10 we have:

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

Let us conclude this section with three examples which explain how to use the definitions we have given.

Example 6.6. Let us consider the command:

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

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

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

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 $\Sigma \to \Sigma_\perp$. Then we have

$$\text{fix } \Gamma_{\textbf{true},\textbf{skip}} = \lambda\sigma.\, \perp_{\Sigma_\perp}.$$

In the following we will often write just Γ when the subscripts b and c are obvious from the context.

Example 6.7. Let us consider the commands

$$w \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } c$$
$$c' \stackrel{\text{def}}{=} \textbf{if } b \textbf{ then } (c\,;\,w) \textbf{ else skip}$$

Now we show the denotational equivalence between w and c' for any b and c. Since $\mathscr{C}[\![w]\!]$ is a fixpoint we have:

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

For c' we have:

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

Hence $\mathscr{C}[\![w]\!] = \mathscr{C}[\![c']\!]$.

Example 6.8. Let us consider the command:

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

we have $\mathscr{C}[\![c]\!] \stackrel{\text{def}}{=} \text{fix } \Gamma$ where:[1]

$$\begin{aligned}\Gamma\varphi &\stackrel{\text{def}}{=} \lambda\sigma.\, \mathscr{B}[\![x \neq 0]\!]\sigma \to \varphi^*(\mathscr{C}[\![x := x-1]\!]\sigma), \sigma \\ &= \lambda\sigma.\, \sigma x \neq 0 \to \varphi^*\left(\sigma[^{\sigma x - 1}/_x]\right), \sigma \\ &= \lambda\sigma.\, \sigma x \neq 0 \to \varphi\left(\sigma[^{\sigma x - 1}/_x]\right), \sigma\end{aligned}$$

Let us see some calculation for approximating the fixpoint:

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

Now we have the following possibilities for computing $\varphi_2\sigma$:

[1] Note that in the last step we can remove the lifting operation from φ^* because $\sigma[^{\sigma x-1}/_x] \neq \bot$.

$\sigma x < 0$) Then $\sigma x \neq 0$ and $\sigma[^{\sigma x-1}/_x]x \neq 0$ and thus $\varphi_2\sigma = \bot_{\Sigma_\bot}$.
$\sigma x = 0$) Then $\sigma x \neq 0$ is false and thus $\varphi_2\sigma = \sigma = \sigma[^0/_x]$
$\sigma x = 1$) Then $\sigma x \neq 0$ and $\sigma[^{\sigma x-1}/_x]x = 0$ and thus $\varphi_2\sigma = \sigma[^{\sigma x-1}/_x] = \sigma[^0/_x]$.
$\sigma x > 1$) Then $\sigma x \neq 0$ and $\sigma[^{\sigma x-1}/_x]x \neq 0$ and thus $\varphi_2\sigma = \bot_{\Sigma_\bot}$.

Summarising:

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

So we have:

$$\varphi_2 = \lambda\sigma.\ \sigma x < 0 \to \bot\ ,\ (\sigma x < 2 \to \sigma[^0/_x], \bot)$$

We can conjecture that $\forall n \in \mathbb{N}.\ P(n)$, where:

$$P(n) \stackrel{\text{def}}{=} (\ \varphi_n = \lambda\sigma.\ 0 \leq \sigma x < n \to \sigma[^0/_x]\ ,\ \bot)$$

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

Base case: The base case is trivial, indeed we know $\varphi_0 = \lambda\sigma.\ \bot$ and

$$\begin{aligned}\lambda\sigma.\ 0 \leq \sigma x < 0 \to \sigma[^0/_x]\ ,\ \bot &= \lambda\sigma.\mathbf{false}\ \to \sigma[^0/_x], \bot)\\ &= \lambda\sigma.\ \bot\end{aligned}$$

Inductive case: For the inductive case, let us assume

$$P(n) \stackrel{\text{def}}{=} (\ \varphi_n = \lambda\sigma.\ 0 \leq \sigma x < n \to \sigma[^0/_x]\ ,\ \bot).$$

We want to prove:

$$P(n+1) \stackrel{\text{def}}{=} (\ \varphi_{n+1} = \lambda\sigma.\ 0 \leq \sigma x < n+1 \to \sigma[^0/_x]\ ,\ \bot)$$

By definition:

$$\varphi_{n+1} = \Gamma\ \varphi_n = \lambda\sigma.\ \sigma x \neq 0 \to \varphi_n(\sigma[^{\sigma x-1}/_x]), \sigma$$

By the inductive hypothesis, we have:

$$\begin{aligned}\varphi_n(\sigma[^{\sigma x-1}/_x]) &= 0 \leq (\sigma[^{\sigma x-1}/_x])x < n \to (\sigma[^{\sigma x-1}/_x])[^0/_x]\ ,\ \bot\\ &= 0 \leq \sigma x - 1 < n \to \sigma[^0/_x]\ ,\ \bot\\ &= 1 \leq \sigma x < n+1 \to \sigma[^0/_x]\ ,\ \bot\end{aligned}$$

Thus:

$$\varphi_{n+1}\sigma = \sigma x \neq 0 \to (\ 1 \leq \sigma x < n+1 \to \sigma[^0/_x]\ ,\ \bot)\ ,\ \sigma$$

Now we have the following possibilities for computing $\varphi_{n+1}\sigma$:

$\sigma x < 0)$	Then $\sigma x \neq 0$ and $\sigma x < 1$, thus $\varphi_{n+1}\sigma = \bot$.
$\sigma x = 0)$	Then $\sigma x \neq 0$ is false and thus $\varphi_{n+1}\sigma = \sigma = \sigma[^0/_x]$
$1 \leq \sigma x < n+1)$	Then $\sigma x \neq 0$ and $1 \leq \sigma x < n+1$ are true, thus $\varphi_{n+1}\sigma = \sigma[^0/_x]$.
$\sigma x \geq n+1)$	Then $\sigma x \neq 0$ is true, but $1 \leq \sigma x < n+1$ is false, thus $\varphi_{n+1}\sigma = \bot$.

Summarising:

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

Then:

$$\varphi_{n+1} = \lambda\sigma.\ 0 \leq \sigma x < n+1 \to \sigma[^0/_x]\ ,\ \bot$$

which proves $P(n+1)$.

Finally we have:

$$\mathscr{C}[\![c]\!] = \text{fix}\ \Gamma = \bigsqcup_{n\in\mathbb{N}} \Gamma^n \bot = \bigsqcup_{n\in\mathbb{N}} \varphi_n = \lambda\sigma.\ 0 \leq \sigma x \to \sigma[^0/_x]\ ,\ \bot$$

6.3 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.

6.3.1 Equivalence Proofs For Expressions

We start by considering arithmetic expressions. We want to prove that the operational and denotational semantics coincide, 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.

Theorem 6.1. *For all arithmetic expressions $a \in Aexp$, the predicate $P(a)$ holds, where:*

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

Proof. The proof is by structural induction on arithmetic expressions.

Const: $P(n) \stackrel{\text{def}}{=} \forall \sigma.\ \langle n, \sigma \rangle \to \mathscr{A}[\![n]\!]\sigma$ holds because, given a generic σ, we have $\langle n, \sigma \rangle \to n$ and $\mathscr{A}[\![n]\!]\sigma = n$.

Vars: $P(x) \stackrel{\text{def}}{=} \forall \sigma.\ \langle x, \sigma \rangle \to \mathscr{A}[\![x]\!]\sigma$ holds because, given a generic σ, we have $\langle x, \sigma \rangle \to \sigma x$ and $\mathscr{A}[\![x]\!]\sigma = \sigma x$.

Ops: 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 $\odot \in \{+, -, \times\}$ of IMP, whose corresponding semantic operator is $\cdot$. We assume:

$$P(a_0) \stackrel{\text{def}}{=} \langle a_0, \sigma \rangle \to \mathscr{A}[\![a_0]\!]\sigma$$
$$P(a_1) \stackrel{\text{def}}{=} \langle a_1, \sigma \rangle \to \mathscr{A}[\![a_1]\!]\sigma$$

and we want to prove

$$P(a_0 \odot a_1) \stackrel{\text{def}}{=} \langle a_0 \odot a_1, \sigma \rangle \to \mathscr{A}[\![a_0 \odot a_1]\!]\sigma.$$

By using the inductive hypothesis we derive:

$$\langle a_0 \odot a_1, \sigma \rangle \to \mathscr{A}[\![a_0]\!]\sigma \cdot \mathscr{A}[\![a_1]\!]\sigma$$

Finally, by definition of $\mathscr{A}$

$$\mathscr{A}[\![a_0]\!]\sigma \cdot \mathscr{A}[\![a_1]\!]\sigma = \mathscr{A}[\![a_0 \odot a_1]\!]\sigma.$$

□

The case of boolean expressions is completely similar to that of arithmetic expressions, so we leave the proof as an exercise (see Problem 6.2).

Theorem 6.2. *For all boolean expressions $b \in Bexp$, the predicate $P(b)$ holds, where:*

$$P(b) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \langle b, \sigma \rangle \to \mathscr{B}[\![b]\!]\sigma$$

From now on we will assume the equivalence between denotational and operational semantics for boolean and arithmetic expressions.

6.3.2 Equivalence Proof for Commands

Central to the proof of equivalence between denotational and operational semantics is the case of commands. Operational and denotational semantics are defined in very different formalisms: on the one hand we have an inference rule system which allows to calculate the execution of each command, on the other hand we have a

function which associates to each command its functional meaning. So to show the equivalence between the two semantics we will prove the following property:

Theorem 6.3. $\forall c \in Com.\ \forall \sigma, \sigma' \in \Sigma.\ \langle c, \sigma\rangle \to \sigma' \Longleftrightarrow \mathscr{C}[\![c]\!]\,\sigma = \sigma'$.

As usual we divide the proof in two parts:

Completeness: $\forall c \in Com,\ \forall \sigma, \sigma' \in \Sigma$ we prove

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

Correctness: $\forall c \in Com$ we prove

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

Notice that in this way the non defined cases are also handled for the equivalence: for instance we have

$$\langle c, \sigma\rangle \not\to \quad \Rightarrow \quad \mathscr{C}[\![c]\!]\,\sigma = \perp_{\Sigma_\perp}$$

since otherwise, assuming $\mathscr{C}[\![c]\!]\,\sigma = \sigma'$ for some $\sigma' \in \Sigma$, it would follow that $\langle c, \sigma\rangle \to \sigma'$. Similarly in the opposite direction.

$$\mathscr{C}[\![c]\!]\,\sigma = \perp_{\Sigma_\perp} \quad \Rightarrow \quad \langle c, \sigma\rangle \not\to$$

6.3.2.1 Completeness of the Denotational Semantics

Let us prove the first part of Theorem 6.3. We let

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

and prove that $P\left(\langle c, \sigma\rangle \to \sigma'\right)$ holds for any $c \in Com$ and $\sigma, \sigma' \in \Sigma$.

We proceed by rule induction. So for each rule we will assume the property holds for the premises and we will prove the property holds for the conclusion.

skip: Let us consider the operational rule for the **skip**

$$\frac{}{\langle \mathbf{skip}, \sigma\rangle \to \sigma}$$

We want 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.

assign: Let us consider the rule for the assignment command:

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

We can assume $\langle a, \sigma \rangle \to m$ and therefore $\mathscr{A}[\![a]\!]\sigma = m$ by the correspondence between the operational and denotational semantics of arithmetic expressions.
We want to prove:

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

By the definition of denotational semantics:

$$\mathscr{C}[\![x := a]\!]\sigma = \sigma[^{\mathscr{A}[\![a]\!]\sigma}/_x] = \sigma[^m/_x]$$

seq: 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 want to prove:

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

By denotational semantics definition and inductive hypotheses:

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

Note that the lifting operator can be removed because $\sigma'' \neq \bot$ by inductive hypothesis.

iftt: Let us consider the rule:

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

We assume:

- $\langle b, \sigma \rangle \to \textbf{true}$ and therefore $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$ by the correspondence between operational and denotational semantics for boolean expressions;
- $P(\langle c_0, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c_0]\!]\sigma = \sigma'$.

We want to prove:

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

In fact, we have:

$$\begin{aligned}\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma &= \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma \\ &= \textbf{true} \to \sigma', \mathscr{C}[\![c_1]\!]\sigma \\ &= \sigma'\end{aligned}$$

ifff: The proof for the second rule of the conditional coommand is completely analogous to the previous one and thus omitted.

whff: Let us consider the rule:

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

We assume $\langle b, \sigma\rangle \to \textbf{false}$ and therefore $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$. We want to prove:

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

By the fixpoint property of the denotational semantics:

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

whtt: At last we consider the second rule of the while command

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

We assume:

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

We want to prove:

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

By the definition of the denotational semantics and inductive hypotheses:

$$\begin{aligned}\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma &= \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma \\ &= \textbf{true} \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*\sigma'', \sigma \\ &= \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*\sigma'' \\ &= \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma'' \\ &= \sigma'\end{aligned}$$

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

6.3.2.2 Correctness of the Denotational Semantics

Let us conclude the proof of Theorem 6.3 by showing the correctness of the denotational semantics. We need to prove, for all $c \in \mathit{Com}$:

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

Since the denotational semantics is given by structural recursion we will proceed by induction on the structure of commands.

skip: We need to prove:

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

By definition we have $\mathscr{C}[\![\mathbf{skip}]\!]\,\sigma = \sigma$ and $\langle \mathbf{skip}, \sigma\rangle \to \sigma$ is an axiom of the operational semantics.

assign: We need to prove:

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

By denotational semantics definition we have $\sigma' = \sigma[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}]$ and by the equivalence between operational and denotational semantics for expressions we have $\langle a, \sigma\rangle \to \mathscr{A}[\![a]\!]\,\sigma$, thus we can apply the rule (assign) to conclude

$$\langle x := a, \sigma\rangle \to \sigma[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}].$$

seq: We assume:

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

We want to prove:

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

We assume $\mathscr{C}[\![c_0; c_1]\!]\,\sigma = \sigma'$ and prove $\langle c_0; c_1, \sigma\rangle \to \sigma'$. We have

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

Since $\sigma' \neq \bot$, it must be $\mathscr{C}[\![c_0]\!]\,\sigma \neq \bot$, i.e., we can assume the termination of c_0 and can omit the lifting operator:

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

Let $\mathscr{C}[\![c_0]\!]\sigma = \sigma''$. We have $\mathscr{C}[\![c_1]\!]\sigma'' = \sigma'$. Then we can apply modus ponens to the inductive assumptions $P(c_0)$ and $P(c_1)$, to get $\langle c_0, \sigma\rangle \to \sigma''$ and $\langle c_1, \sigma''\rangle \to \sigma'$. Thus we can apply the inference rule:

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

to conclude $\langle c_0; c_1, \sigma\rangle \to \sigma'$.

if: We assume:

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

We need to prove:

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

Let us assume the premise $\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \sigma'$ and prove that $\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'$. By definition:

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

Now, either $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$ or $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$.
If $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$, we have also $\langle b, \sigma\rangle \to \textbf{false}$. Then:

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

By modus ponens on the inductive hypothesis $P(c_1)$ we have $\langle c_1, \sigma\rangle \to \sigma'$. Thus we can apply the rule:

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

to conclude $\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'$.
The case where $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$ is completely analogous and thus omitted.

while: We assume:

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

We need to prove:

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

By definition $\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \text{fix } \Gamma_{b,c}\,\sigma = \left(\bigsqcup_{n\in\mathbb{N}} \Gamma_{b,c}^n \bot\right)\sigma$ so:

$$\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$
$$\Leftrightarrow$$
$$\left(\bigsqcup_{n\in\mathbb{N}} \Gamma^n_{b,c}\bot\right)\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$
$$\Leftrightarrow$$
$$\left(\exists n \in \mathbb{N}.\ (\Gamma^n_{b,c}\bot)\sigma = \sigma'\right) \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$
$$\Leftrightarrow$$
$$\forall n \in \mathbb{N}.\ \left(\Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'\right)$$

Let

$$A(n) \stackrel{\text{def}}{=} \forall\sigma,\sigma'.\ \Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$

We prove that $\forall n \in \mathbb{N}.\ A(n)$ by mathematical induction.

Base case: We have to prove $A(0)$, namely:

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

Since $\Gamma^0_{b,c}\bot\sigma = \bot\sigma = \bot$ and $\sigma' \neq \bot$ the premise is false and hence the implication is true.

Ind. case: Let us assume

$$A(n) \stackrel{\text{def}}{=} \forall\sigma,\sigma'.\ \Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'.$$

We want to show that

$$A(n+1) \stackrel{\text{def}}{=} \forall\sigma,\sigma'.\ \Gamma^{n+1}_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'.$$

We assume $\Gamma^{n+1}_{b,c}\bot\sigma = \Gamma_{b,c}\left(\Gamma^n_{b,c}\bot\right)\sigma = \sigma'$, that is

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

Now either $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$ or $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$.

- If $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$, we have $\langle b, \sigma\rangle \to \textbf{false}$ and $\sigma' = \sigma$. Now by using the rule (whff):

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

  we conclude $\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma$.
- if $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$ we have $\langle b, \sigma\rangle \to \textbf{true}$ and

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

  Since $\sigma' \neq \bot$ there must exists some $\sigma'' \neq \bot$ with $\mathscr{C}[\![c]\!]\sigma = \sigma''$ and by structural induction $\langle c, \sigma\rangle \to \sigma''$.

Since $\left(\Gamma^n_{b,c}\bot\right)^*(\mathscr{C}[\![c]\!]\sigma) = \left(\Gamma^n_{b,c}\bot\right)\sigma'' = \sigma'$ we have by the mathematical induction hypothesis $A(n)$ that

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

Finally by using the rule (whtt):

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

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

6.4 Computational Induction

How are we able to prove properties about fixpoints? To fill this gap we introduce Scott's *computational induction*, which applies to a class of properties corresponding to inclusive sets.

Definition 6.10 (Inclusive property). Let $(D,\sqsubseteq)$ be a CPO, let $P \subseteq D$ be a set, we say that P is an *inclusive* set if and only if:

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

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

Intuitively, a set P is inclusive if whenever we form a chain out of elements in P, then the limit of the chain is also in P, i.e., P is inclusive if and only if it forms a CPO.

Example 6.9 (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 $x \sqsubseteq y$ iff x is a finite prefix of y. Let us now define the following property:

- $x \in \{a,b\}^* \cup \{a,b\}^\infty$ is fair iff $\nexists y \in \{a,b\}^*.\ x = ya^\infty \ \vee\ x = yb^\infty$

Fairness is the property of an arbiter which does not favor one of two competitors all the times from some point on. Fairness is not inclusive, indeed,

- the sequence a^n is finite and thus fair for any $n \in \mathbb{N}$;
- $\bigsqcup_{n\in\mathbb{N}} a^n = a^\infty$;
- a^∞ is obviously not fair.

Theorem 6.4 (Computational Induction). *Let P be a property, $(D,\sqsubseteq)$ a CPO$_\bot$ and f a monotone and continuous function on it. Then the inference rule:*

$$\frac{P \text{ inclusive} \quad \bot \in P \quad \forall d \in D.\ (d \in P \Rightarrow f(d) \in P)}{\operatorname{fix} f \in P}$$

is sound.

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

Example 6.10 (Computational induction). Let us consider the command

$$w \stackrel{\text{def}}{=} \textbf{while } x \neq 0 \textbf{ do } x := x - 1$$

from Example 6.8. We want to prove the property

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

By definition:

$$\mathscr{C}[\![w]\!] = \operatorname{fix} \Gamma \quad \text{where} \quad \Gamma \stackrel{\text{def}}{=} \lambda\varphi.\ \lambda\sigma.\ \sigma x \neq 0 \rightarrow \varphi\sigma[^{\sigma x - 1}/_x], \sigma$$

Let us define the property:

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

we will show that the property is inclusive, that is, taken a chain $\{\varphi_i\}_{i\in\mathbb{N}}$ we have:

$$(\forall i \in \mathbb{N}.\ P(\varphi_i)) \Rightarrow P(\bigsqcup_{i\in\mathbb{N}} \varphi_i)$$

Let us assume $\forall i \in \mathbb{N}.\ P(\varphi_i)$, namely that:

$$\forall i, \sigma, \sigma'.\ \left(\varphi_i\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]\right)$$

We want to prove that

$$\forall \sigma, \sigma'.\ \left((\bigsqcup_{i\in\mathbb{N}} \varphi_i)\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x] \right)$$

Suppose $(\bigsqcup_{i\in\mathbb{N}} \varphi_i)\sigma = \sigma'$. We are left to prove that $\sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]$. By $(\bigsqcup_{i\in\mathbb{N}} \varphi_i)\sigma = \sigma'$ we have that $\exists k \in \mathbb{N}.\ \varphi_k\sigma = \sigma'$. Then we can conclude the thesis by $P(\varphi_k)$.

We can now use the computational induction:

$$\frac{P \text{ inclusive} \quad P(\bot) \quad \forall \varphi.\ P(\varphi) \Rightarrow P(\Gamma\varphi)}{P(\operatorname{fix}\Gamma)}$$

as $P(\operatorname{fix}\Gamma) = P(\mathscr{C}[\![w]\!])$.

P inclusive: It has been proved above.

$P(\bot)$: It is obvious, since $\bot\sigma = \sigma'$ is always false.

$\forall\varphi.\, P(\varphi) \Rightarrow P(\Gamma\varphi)$: We assume

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

and we want to prove

$$P(\Gamma\varphi) = \forall\sigma,\sigma'.\, (\Gamma\varphi\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[{}^{0}/_{x}])$$

We assume the premise

$$\Gamma\varphi\sigma = \big(\sigma x \neq 0 \to \varphi\sigma[{}^{\sigma x-1}/_{x}], \sigma\big) = \sigma'$$

and need to prove that $\sigma x \geq 0 \wedge \sigma' = \sigma[{}^{0}/_{x}]$. There are two cases to consider:

- If $\sigma x = 0$, we have

  $$(\sigma x \neq 0 \to \varphi\sigma[{}^{\sigma x-1}/_{x}], \sigma) = \sigma$$

  therefore $\sigma' = \sigma$ and trivially

  $$\sigma x = 0 \geq 0 \qquad \sigma' = \sigma = \sigma[{}^{0}/_{x}].$$

- If $\sigma x \neq 0$, we have

  $$(\sigma x \neq 0 \to \varphi\sigma[{}^{\sigma x-1}/_{x}], \sigma) = \varphi\sigma[{}^{\sigma x-1}/_{x}].$$

  Let $\sigma'' = \sigma[{}^{\sigma x-1}/_{x}]$. We exploit $P(\varphi)$ over σ'', σ':

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

  we have:

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

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

Finally, we can conclude by computational induction that the property P holds for the fixpoint $\operatorname{fix}\Gamma$ and thus for the semantics of the command w as $\mathscr{C}[\![w]\!] = \operatorname{fix}\Gamma$.

Problems

6.1. The following problems serve to get acquainted with the use of variables in the lambda-notation.

1. Is $\lambda x.\ \lambda x.\ x$ α-convertible to one or more of the following expressions?

   a. $\lambda y.\ \lambda x.\ x$
   b. $\lambda y.\ \lambda x.\ y$
   c. $\lambda y.\ \lambda y.\ y$
   d. $\lambda x.\ \lambda y.\ x$
   e. $\lambda z.\ \lambda w.\ w$

2. Is $((\lambda x.\ \lambda y.\ x)\ y)$ equivalent to one or more of the following expressions?

   a. $\lambda y.\ \lambda y.\ y$
   b. $\lambda y.\ y$
   c. $\lambda y.\ z$
   d. $\lambda z.\ y$
   e. $\lambda x.\ y$

6.2. Prove Theorem 6.2.

6.3. Prove that the commands

$$c \stackrel{\text{def}}{=} x := 0;\ \textbf{if}\ x = 0\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \qquad\qquad c' \stackrel{\text{def}}{=} x := 0;\ c_1$$

are semantically equivalent for any commands c_1, c_2. Carry out the proof using both the operational semantics and the denotational semantics.

6.4. Prove that the two commands

$$w \stackrel{\text{def}}{=} \textbf{while}\ b\ \textbf{do}\ c \qquad\qquad w' \stackrel{\text{def}}{=} \textbf{while}\ b\ \textbf{do}\ (\textbf{if}\ b\ \textbf{then}\ c\ \textbf{else skip})$$

are equivalent for any b and c using the denotational semantics.

6.5. Prove that $\mathscr{C}[\![\textbf{while true do skip}]\!] = \mathscr{C}[\![\textbf{while true do}\ x := x+1]\!]$.

6.6. Prove that $\mathscr{C}[\![\textbf{while}\ x \neq 0\ \textbf{do}\ x := 0]\!] = \mathscr{C}[\![x := 0]\!]$.

6.7. Prove that

$$\mathscr{C}[\![\textbf{while}\ x = 0\ \textbf{do skip}]\!] = \mathscr{C}[\![\textbf{if}\ x = 0\ \textbf{then}\ (\textbf{while true do}\ x := 0)\ \textbf{else skip}]\!].$$

6.8. Introduce in **IMP** the command

$$\textbf{repeat}\ n\ \textbf{times}\ c$$

with n natural number, instead of the command **while**. Its denotational semantics is

$$\mathscr{C}[\![\textbf{repeat}\ n\ \textbf{times}\ c]\!]\,\sigma = (\mathscr{C}[\![c]\!])^n \sigma$$

1. Define the operational semantics of the new command.
2. Extend the proof of equivalence of the operational and denotational semantics of IMP to take into account the new command.
3. Prove that the execution of every command terminates.

6.9. Add to IMP the command

$$\textbf{reset } x \textbf{ in } c$$

with the following informal meaning: execute the command c in the state where x is reset to 0, then after the execution of c reassign to location x its original value.

1. Define the operational semantics of the new command.
2. Define the denotational semantics of the new command.
3. Extend the proof of equivalence of the operational and denotational semantics of IMP to take into account the new command.

6.10. Add to **IMP** the command

$$\textbf{do } c \textbf{ undoif } b$$

with the following informal meaning: execute c; if after the execution of c the boolean expression b is satisfied, then go back to the state before the execution of c.

1. Define the operational semantics of the new command.
2. Define the denotational semantics of the new command.
3. Extend the proof of equivalence of the operational and denotational semantics of IMP to take into account the new command.

6.11. Extend IMP with the command

$$\textbf{try } c_1 = c_2 \textbf{ else } c_3$$

that returns the store obtained by computing c_1 if it coincides with the one obtained by computing c_2; if they differ returns the store obtained by computing c_3; it diverges otherwise.

1. Define the operational semantics of the new command.
2. Define the denotational semantics of the new command.
3. Extend the proof of correspondence between the operational and the denotational semantics.

6.12. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } y > 0 \textbf{ do } (r := r \times x \,;\, y := y - 1)$$

Compute the denotational semantics $\mathscr{C}[\![w]\!] = \operatorname{fix} \Gamma$.

Hint: Prove that letting $\varphi_n \stackrel{\text{def}}{=} \Gamma^n \bot_{\Sigma \to \Sigma_\bot}$ it holds $\forall n \geq 1$

$$\varphi_n = \lambda\sigma.\ (\sigma y > 0) \to (\ (\sigma y \geq n) \to \bot_{\Sigma_\bot}\ ,\ \sigma[{}^{\sigma r \times (\sigma x)^{\sigma y}}/_r, {}^{0}/_y]\ )\ ,\ \sigma.$$

6.13. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } x \neq 0 \textbf{ do } (x := x-1 \ ; \ y := y+1)$$

Prove, using Scott computational induction, that for all σ, σ' we have:

$$\mathscr{C}[\![w]\!]\,\sigma = \sigma' \quad \Rightarrow \quad \sigma(x) \geq 0 \wedge \sigma' = \sigma[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x}]$$

Part III
HOFL: a higher-order functional language

This part focuses on models for sequential computations that are associated to HOFL, a higher-order declarative language that follows the functional style. Chapter 7 presents the syntax, typing and operational semantics of HOFL, while Chapter 9 defines its denotational semantics. The two are related in Chapter 10. Chapter 8 extends the theory presented in Chapter 5 to allow the definition of more complex domains, as needed by the type-constructors available in HOFL.

Chapter 7
Operational Semantics of HOFL

Typing is no substitute for thinking. (Richard Hamming)

Abstract In the previous part of the book we have introduced and studied an imperative language called IMP. In this chapter we move our attention to functional languages. In particular, we introduce HOFL, a simple higher-order functional language which allows for the explicit construction of infinitely many types. We overview Church and Curry type theories. Then, we present a *lazy* operational semantics, which corresponds to a *call-by-name* strategy, namely actual parameters are passed to functions without evaluating them. This view is contrasted with the *eager* evaluation semantics, which corresponds to a *call-by-value* strategy, where all actual parameters are evaluated before being passed to functions. The operational semantics evaluates (well-typed) terms to suitable canonical forms.

7.1 Syntax of HOFL

We start by introducing the plain syntax of HOFL. Then we discuss the type theory and define the well-formed terms. Finally we present the operational semantics of well-formed terms, which reduces terms to their canonical form (when they exist).

In IMP there are only three types: *Aexp* for arithmetic expressions, *Bexp* for boolean expressions and *Com* for commands. Since IMP does not allow to construct other types explicitly, these types are directly embedded in its syntax. HOFL, instead, allows one to define a variety of types, so we first present the grammar for *pre-terms*, then we introduce the concept of typed terms, namely the well-formed sentences of HOFL. 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. We assume a set *Var* of variables is given.

Definition 7.1 (HOFL: syntax). The following productions define the syntax of HOFL pre-terms:

$$t \quad ::= \quad x \mid n \mid t_0 + t_1 \mid t_0 - t_1 \mid t_0 \times t_1 \mid \textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \mid (t_0, t_1) \mid \textbf{fst}(t) \mid \textbf{snd}(t) \mid \lambda x.\, t \mid (t_0\ t_1) \mid \textbf{rec}\, x.\, t$$

where x is a variable and n an integer.

Besides usual variables x, constants n and arithmetic operators $+, -, \times$, we find: a conditional construct **if** t **then** t_0 **else** t_1 that reads as **if** $t = 0$ **then** t_0 **else** t_1; the constructs for pairing terms (t_0, t_1) and for projecting over the first and second component of a pair $\textbf{fst}(t)$ and $\textbf{snd}(t)$; function abstraction $\lambda x.\, t$ and application $(t_0\ t_1)$; and recursive definition $\textbf{rec}\, x.\, t$. Recursion allows to define recursive terms, namely $\textbf{rec}\, x.\, t$ defines a term t that can contain variable x, which in turn can be replaced by its recursive definition $\textbf{rec}\, x.\, t$.

We call *pre-terms* the terms generated by the syntax above, because it is evident that one could write ill-formed terms, like applying a projection to an integer instead of a pair ($\textbf{fst}(1)$) or summing an integer to a function ($1 + \lambda x.\, x$). To avoid these constructions we introduce the concepts of *type* and *typed term*.

7.1.1 Typed Terms

Definition 7.2 (HOFL types). A HOFL type is a term constructed by using the following grammar:

$$\tau \quad ::= \quad \mathit{int} \quad \mid \quad \tau_0 * \tau_1 \quad \mid \quad \tau_0 \to \tau_1$$

We let $\mathscr{T}$ denote the set of all types.

We allow constant type *int*, the pair type $\tau_0 * \tau_1$ and the function type $\tau_0 \to \tau_1$. Using these productions we can define infinitely many types, like $(\mathit{int} * \mathit{int}) \to \mathit{int}$ for functions that take as argument a pair of integers and return an integer, and $\mathit{int} \to (\mathit{int} * (\mathit{int} \to \mathit{int}))$ for functions that take an integer and return an integer in pair with a function from integers to integers.

Now we define the rule system which allows to say if a pre-term of HOFL is well-formed (i.e., if we can or not associate a type expressed in the above grammar to a given pre-term). The predicates we are interested in are of the form $t : \tau$, expressing that the pre-term t is well-formed and has type τ. We assume variables are typed, i.e., that a function $\widehat{(\cdot)} : \mathit{Var} \to \mathscr{T}$ is given, which assigns a unique type to each variable.

$$\frac{}{x : \widehat{x}}$$

The rule for variables assign to each variable x its type $\widehat{x}$.

$$\frac{}{n : \mathit{int}} \qquad \frac{t_0 : \mathit{int} \quad t_1 : \mathit{int}}{t_0 \text{ op } t_1 : \mathit{int}}\ \text{op} \in \{+, -, \times\} \qquad \frac{t : \mathit{int} \quad t_0 : \tau \quad t_1 : \tau}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 : \tau}$$

The rules for arithmetic expressions assign type *int* to each integer n and to each expression built using $+, -, \times$, whose arguments must be of type *int* too. The rule for conditional expressions **if** t **then** t_0 **else** $t_1 : \tau$ requires the condition t to be of type *int* and the the two branches t_0 and t_1 to have the same type τ, which is also the type of the conditional expression.

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

The rule for pairing says that the type of a term (t_0, t_1) is the pair type $\tau_0 * \tau_1$, where t_i has type τ_i for $i = 0, 1$. Vice versa, for projections it is required that the argument t has pair type $\tau_0 * \tau_1$ for some τ_0 and τ_1, and the result has type τ_0 when the first projection is used or τ_1 when the second projection is used.

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

The rule for function abstraction assigns to $\lambda x.\, t$ the functional type $\tau_0 \to \tau_1$, where τ_0 is the type of x and τ_1 is the type of t. In the case of function application $(t_1\ t_0)$, it is required that t_1 has functional type $\tau_0 \to \tau_1$ for some types τ_0 and τ_1, where τ_0 is also the type of t_0. Then, the result has type τ_1.

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

The last rule handles recursion: it check that the type τ of the defining expression t is the same as the type of the recursively defined name x; if so, then τ is also the type of the recursive expression **rec** $x.\, t$.

Definition 7.3 (Well-Formed Terms). Let t be a pre-term of HOFL, we say that t is *well-formed* (or *well-typed*, or *typable*) if there exists a type τ such that $t : \tau$.

We name T_τ the set of well-formed terms of type τ.

Note that our type system is very simple. Indeed it does not allow to construct useful types, such as recursive, parametric, dependent, polymorphic or abstract types. These limitations imply that we cannot construct many useful terms. For instance, while it is easy to express the types for lists of integer numbers of fixed length (using the type pairing operator $*$) and functions that manipulate them, in our type system lists of integer numbers of variable length are not typable, because some form of recursion should be allowed at the level of types to express them.

7.1.2 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.

7.1.2.1 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 inference rules in a bottom-up fashion).

In this case, we sometimes annotate directly the bounded variables with their type, like in $\lambda x : int.\ x + x$ or **rec** $f : int \to int.\ \lambda x : int.\ fx$.

Example 7.1 (Factorial with Church types). Let $x : int$ and $f : int \to int$ in the pre-term:

$$fact \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ (x \times (f(x-1)))$$

So we can type *fact* and all its subterms as below:

$$\dfrac{\dfrac{\widehat{f} = int \to int \quad \dfrac{\dfrac{\widehat{x} = int}{x : int} \quad \dfrac{}{1 : int}}{\dfrac{\widehat{x} = int}{x : int} \quad \dfrac{\dfrac{\widehat{f} = int \to int}{f : int \to int} \quad \dfrac{\dfrac{\widehat{x} = int}{x : int} \quad \dfrac{}{1 : int}}{x - 1 : int}}{f(x-1) : int}}{(x \times (f(x-1))) : int}}{\dfrac{\widehat{x} = int \quad \dfrac{\dfrac{\widehat{x} = int}{x : int} \quad \dfrac{}{1 : int}}{(x \times (f(x-1))) : int}}{\textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ (x \times (f(x-1))) : int}}}{\dfrac{f : int \to int \quad \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ (x \times (f(x-1))) : int \to int}{fact : int \to int}}$$

More concisely, we write:

$$fact \stackrel{\text{def}}{=} \textbf{rec}\ \underset{int \to int}{f}.\lambda\ \underset{int}{x}.\textbf{if}\ \underset{int}{x}\ \textbf{then}\ \underset{int}{1}\ \textbf{else}\ \underset{int}{x} \times (\underset{int \to int}{f}\ (\underbrace{\underset{int}{x} - \underset{int}{1}}_{int}))\ \underbrace{}_{int}\ \underbrace{}_{int}\ \underbrace{}_{int}\ \underbrace{}_{int \to int} : int \to int$$

7.1.2.2 Curry Type Theory

In Curry style, we do not need to explicitly declare the type of each variable. Instead we use the inference rules to calculate type equations (i.e., equations which have types as variables) whose solutions define all the possible type assignments for 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 instances of a single type term with variables, where one instance is obtained by freely 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, as we have done in Example 7.5.

Example 7.2 (Identity). Let us consider the identity function:

$$\lambda x.\ x$$

By using the type system we have:

$$\lambda x.\ x : \tau \quad \nwarrow_{\tau=\tau_1\to\tau_2,\ \widehat{x}=\tau_1} \quad x : \tau_2$$
$$\nwarrow_{\widehat{x}=\tau_2} \quad \square$$

So we have $\widehat{x} = \tau_1 = \tau_2$ and the principal type of $\lambda x.\ x$ is $\tau_1 \to \tau_1$. 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 $\tau = int \to int$, but if we set $\tau_1 = int * (int \to int)$ we have $\tau = (int * (int \to int)) \to (int * (int \to int))$.

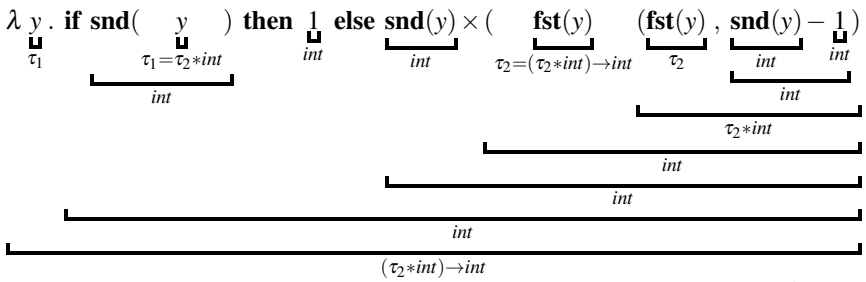
Example 7.3 (Non-typable term of HOFL). Let us consider the following function, which computes the factorial without using recursion.

begin
$$fact(f,x) \stackrel{\text{def}}{=} \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x \times f(f, x-1)$$
$$fact(fact, 3)$$
end

The first instruction defines *fact* as a function that takes two arguments (e.g., a function f and an integer x) and returns 1 if $x = 0$ and returns $x \times f(f, x-1)$ otherwise. The second instruction invokes *fact* by passing *fact* as a first argument and the number 3 as second argument. Since $3 \neq 0$, the invocation will trigger the calculation $3 \times fact(fact, 2)$ and so on. It can be translated to HOFL as follows:

$$fact \stackrel{\text{def}}{=} \lambda y.\ \textbf{if } \textbf{snd}(y) = 0 \textbf{ then } 1 \textbf{ else } \textbf{snd}(y) \times \textbf{fst}(y)(\textbf{fst}(y), \textbf{snd}(y) - 1)$$

We can try to infer the type of *fact* as follows:



We derive $\mathbf{fst}(y) : \tau_2$ and $\mathbf{fst}(y) : (\tau_2 * int) \to int$. Thus we have $\tau_2 = (\tau_2 * int) \to int$ which has no solution.

We recall the unification algorithm from Section 2.1.4 that can be used to solve general systems of type equations as well. We recall it here, in compact form, to address explicitly the unification of terms that denote types. The idea is that types are terms built over a suitable signature. In the case of HOFL, the signature just consists of the constant *int* and two binary operators $*$ and $\to$ and variables are usually denoted as τ's. We start from a system of type equations like:

$$\begin{cases} t_1 = t'_1 \\ t_2 = t'_2 \\ \dots \\ t_k = t'_k \end{cases}$$

and then we apply iteratively in any order the following steps:

1. We eliminate all the equations like $\tau = \tau$ for τ a type variable.
2. For each equation of the form $f(u_1, \dots, u_n) = f'(u'_1, \dots, u'_m)$:[1]

   if $f \neq f'$: then the system has no solutions and we stop.
   if $f = f'$: then $n = m$ so we must have:

   $$u_1 = u'_1, u_2 = u'_2, \dots, u_n = u'_n$$

   and thus we replace the original equation with these.

3. For each equation of the type $\tau = t$ with $t \neq \tau$:

   if τ appears in t: then the system has no solutions.
   if τ does not appear in t: we replace each occurrence of τ with t in all the other equations.

Eventually, either the system is recognised 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

[1] In our case f and f' can be taken from $\{int, *, \to\}$.

execution strategies yield a complexity linear or quasi linear with the size of the original system of equations.

Example 7.4. Let us now apply the algorithm to the Example 7.3: We have the type equation

$$\tau_2 = (\tau_2 * int) \to int$$

1. We cannot apply step 1 of the algorithm, because the equation does not express a trivial equality.
2. We cannot apply step 2 either, because the left-hand side of the equation consists of a variable and not of an operator applied to some subterms, as required.
3. Step 3 can be applied and it fails, because the type variable τ_2 appears in the right hand side.

Here we show another interesting term which is not typable.

Example 7.5 (Non-typable terms). Let us define a pre-term t which, when applied to the argument 0, should define the list of all even numbers:

$$t \stackrel{\text{def}}{=} \textbf{rec}\ p.\ \lambda x.\ (x, (p\ (x+2)))$$

Intuitively, the term $t\ 0$ takes the value 0 and place it in the first position of a pair, whose second component is the term t itself applied to $0+2=2$, so recursively $t\ 0$ should represent the infinite list of all even numbers:

$$t\ 0 \equiv (0, (t\ 2)) \equiv (0, (2, (t\ 4))) \equiv \cdots \equiv (0, (2, (4, \ldots)))$$

Let us show that this term is not typable:

$$\begin{array}{rl}
t = \textbf{rec}\ p.\ \lambda x.\ (x, (p\ (x+2))) : \tau & \nwarrow_{\widehat{p}=\tau}\ \lambda x.\ (x, (p\ (x+2))) : \tau \\
\nwarrow_{\tau=\tau_1\to\tau_2,\ \widehat{x}=\tau_1} & (x, (p\ (x+2))) : \tau_2 \\
\nwarrow_{\tau_2=\tau_3*\tau_4} & x : \tau_3,\ (p\ (x+2)) : \tau_4 \\
\nwarrow_{\widehat{x}=\tau_3} & (p\ (x+2)) : \tau_4 \\
\nwarrow & p : \tau_5 \to \tau_4,\ (x+2) : \tau_5 \\
\nwarrow_{\widehat{p}=\tau_5\to\tau_4} & (x+2) : \tau_5 \\
\nwarrow_{\tau_5=int} & x : int \\
\nwarrow_{\widehat{x}=int} & \square
\end{array}$$

So we have:

$$\widehat{x} = \tau_1 = \tau_3 = int \qquad \tau_2 = (\tau_3 * \tau_4) = (int * \tau_4) \qquad \tau = (\tau_1 \to \tau_2) = (int \to (int * \tau_4)) \qquad \tau_5 = int$$

From which:

$$\widehat{p} = \tau = (int \to (int * \tau_4)) \qquad \text{and} \qquad \widehat{p} = (\tau_5 \to \tau_4) = (int \to \tau_4)$$

Thus it must be the case that

$$int * \tau_4 = \tau_4$$

which is absurd, because it is not possible to unify τ_4 with a composed term containing an occurrence of τ_4. The above argument is represented more concisely below:

$$t = \textbf{rec}\ p.\quad \lambda\, \underset{int}{x}\,.\ (\underset{int}{x}, (\ \underset{int\to\tau_4}{p}\quad (\underbrace{\underset{int}{x}\ \underset{int}{2}}_{int})))$$
$$\underbrace{\qquad\qquad}_{\tau_4}$$
$$\underbrace{\qquad\qquad\qquad}_{int*\tau_4}$$
$$\underbrace{\qquad\qquad\qquad\qquad}_{(int\to(int*\tau_4)) = (int\to\tau_4) \Rightarrow \tau_4 = (int*\tau_4)}$$

So we have no solutions, and the term is not a well-formed term.

7.2 Operational Semantics of HOFL

In Section 6.1 we have defined the concepts of free variables and substitution for the λ-calculus. Now we define the same concepts in the case of HOFL, which will be necessary to define its operational semantics.

Definition 7.4 (Free variables). We define the set of free-variables of HOFL terms by structural recursion, as follows:

$$\begin{aligned}
\mathrm{fv}(n) &\stackrel{\text{def}}{=} \varnothing \\
\mathrm{fv}(x) &\stackrel{\text{def}}{=} \{x\} \\
\mathrm{fv}(t_0\ \mathrm{op}\ t_1) &\stackrel{\text{def}}{=} \mathrm{fv}(t_0) \cup \mathrm{fv}(t_1) \\
\mathrm{fv}(\textbf{if}\ t\ \textbf{then}\ t_0\ \textbf{else}\ t_1) &\stackrel{\text{def}}{=} \mathrm{fv}(t) \cup \mathrm{fv}(t_0) \cup \mathrm{fv}(t_1) \\
\mathrm{fv}((t_0, t_1)) &\stackrel{\text{def}}{=} \mathrm{fv}(t_0) \cup \mathrm{fv}(t_1) \\
\mathrm{fv}(\textbf{fst}(t)) &\stackrel{\text{def}}{=} \mathrm{fv}(t) \\
\mathrm{fv}(\textbf{snd}(t)) &\stackrel{\text{def}}{=} \mathrm{fv}(t) \\
\mathrm{fv}(\lambda x.\, t) &\stackrel{\text{def}}{=} \mathrm{fv}(t) \backslash \{x\} \\
\mathrm{fv}((t_0\ t_1)) &\stackrel{\text{def}}{=} \mathrm{fv}(t_0) \cup \mathrm{fv}(t_1) \\
\mathrm{fv}(\textbf{rec}\ x.\, t) &\stackrel{\text{def}}{=} \mathrm{fv}(t) \backslash \{x\}
\end{aligned}$$

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

Definition 7.5 (Capture-avoiding substitution). Capture avoiding substitution $[{}^{t}/_{x}]$ of x with t is defined by structural recursion over HOFL terms as follows:

$$
\begin{aligned}
n[^t/_x] &= n \\
y[^t/_x] &\stackrel{\text{def}}{=} \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \\
(t_0 \text{ op } t_1)[^t/_x] &\stackrel{\text{def}}{=} t_0[^t/_x] \text{ op } t_1[^t/_x] \quad \text{with op} \in \{+,-,\times\} \\
(\textbf{if } t' \textbf{ then } t_0 \textbf{ else } t_1)[^t/_x] &\stackrel{\text{def}}{=} \textbf{if } t'[^t/_x] \textbf{ then } t_0[^t/_x] \textbf{ else } t_1[^t/_x] \\
(t_0,t_1)[^t/_x] &\stackrel{\text{def}}{=} (t_0[^t/_x], t_1[^t/_x]) \\
\textbf{fst}(t')[^t/_x] &\stackrel{\text{def}}{=} \textbf{fst}(t'[^t/_x]) \\
\textbf{snd}(t')[^t/_x] &\stackrel{\text{def}}{=} \textbf{snd}(t'[^t/_x]) \\
(t_0\ t_1)[^t/_x] &\stackrel{\text{def}}{=} (t_0[^t/_x]\ t_1[^t/_x]) \\
(\lambda y.\ t')[^t/_x] &\stackrel{\text{def}}{=} \lambda z.\ (\ t'[^z/_y][^t/_x]\ ) \quad \text{with } z \notin \text{fv}(\lambda y.\ t') \cup \text{fv}(t) \cup \{x\} \\
(\textbf{rec } y.\ t')[^t/_x] &\stackrel{\text{def}}{=} \textbf{rec } z.\ (t'[^z/_y][^t/_x]) \quad \text{with } z \notin \text{fv}(\textbf{rec } y.\ t') \cup \text{fv}(t) \cup \{x\}
\end{aligned}
$$

Note that in the last two rules we perform α-conversion $[^z/_y]$ of the bound variable y with a fresh identifier z before the substitution. This ensures that the free occurrences of y in t, if any, are not bound accidentally after the substitution. As discussed in Section 6.1, the substitution is well-defined if we consider the terms up to α-conversion (i.e., up to the renaming of bound variables). 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 7.1 (Substitution Respects Types). *Let $x,t : \tau$ and $t' : \tau'$. Then, we have*

$$t'[^t/_x] : \tau'$$

Proof. The proof is in two steps. First we prove by rule induction the stronger predicate (for any term t' and type τ')

$$P(t' : \tau') \stackrel{\text{def}}{=} \forall x,t : \tau.\ \forall n \in \mathbb{N}.\ \forall x_1,z_1 : \tau_1,\ldots,x_n,z_n : \tau_n.\ t'[^{z_1}/_{x_1},\cdots,^{z_n}/_{x_n},^t/_x] : \tau'$$

Second, the main statement of the theorem follows as the special case where $n = 0$. The stronger assertion is needed for handling the cases of functions (i.e., $t' = \lambda y.\ t''$ for some y and t'') and recursive expressions (i.e., $t' = \textbf{rec } y.\ t''$ for some y and t''), which are the only non trivial cases (because of the way in which capture-avoiding substitution is defined). The reader is left to fill in the missing details of the proof as an exercise. □

We are now ready to present the 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 term of HOFL. In particular, we focus on *closed terms* only, i.e., terms t with no free variables ($\text{fv}(t) = \varnothing$). Canonical forms are particular closed terms,

which we will assume to be the results of calculations (i.e., as ordinary values). For each type we fix the set of terms in canonical form by taking a subset of terms which reasonably represent the notion of values for that type.

As shown in the previous section, HOFL has three type constructors: the constant *int*, and the binary operators $*$ for pairs and $\to$ for functions. Terms which represent the integers provide the obvious canonical forms for the integer type. For pair types we take any pair of terms as canonical form: note that this choice is arbitrary; for example we could have taken instead pairs of terms that are themselves in canonical form. We will explain later the rationale of our choice. 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 7.6 (Canonical forms). Let us define a set C_τ of canonical forms for each type τ as follows:

$$\frac{}{n \in C_{int}} \qquad \frac{t_0 : \tau_0 \quad t_1 : \tau_1 \quad t_0, t_1 \text{ closed}}{(t_0, t_1) \in C_{\tau_0 * \tau_1}} \qquad \frac{\lambda x.\, t : \tau_0 \to \tau_1 \quad \lambda x.\, t \text{ closed}}{\lambda x.\, t \in C_{\tau_0 \to \tau_1}}$$

We now define the rules of the operational semantics; these rules define an evaluation relation:

$$t \to c$$

where t is a well-formed closed term of HOFL and c is its canonical form.

For terms that are already in canonical form according to Definition 7.6 we let:

$$\frac{}{c \to c}$$

For clarity, the above rule offers a concise representation to the otherwise verbose rules:

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

Next, we give the rules for arithmetic expressions.

$$\frac{t_0 \to n_0 \quad t_1 \to n_1}{t_0 \text{ op } t_1 \to n_0 \underline{\text{op}}\, n_1} \qquad \frac{t \to 0 \quad t_0 \to c_0}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_0} \qquad \frac{t \to n \quad n \neq 0 \quad t_1 \to c_1}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_1}$$

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 HOFL syntactic operators and meta-operators by underlying the latter. For instance, we have $1 + 2 \to 3$, since $1 \to 1$, $2 \to 2$ and $1 \underline{+} 2 = 3$.

We recall that for the conditional statement, since we have no boolean values, we use the convention that **if** t **then** t_0 **else** t_1 stands for **if** $t = 0$ **then** t_0 **else** t_1, so the premise $t \to n \neq 0$ means the test is false and $t \to 0$ means the test is true.

Let us now consider the pairing. Obviously, since we consider pairs as canonical values, we do not have to add further rules for simple pairs. We have instead two rules for projections:

$$\frac{t \to (t_0, t_1) \quad t_0 \to c_0}{\mathbf{fst}(t) \to c_0} \qquad \frac{t \to (t_0, t_1) \quad t_1 \to c_1}{\mathbf{snd}(t) \to c_1}$$

The rules are obviously similar: the canonical form of t is computed, which must be of the form (t_0, t_1), because t must have pair type for the projection to be applicable and $\mathbf{fst}(t)$ typable. Note however that t_0 and t_1 need not be in canonical form. So only the canonical form of the component indicated by the projection operator is computed, with the other component discarded.

Function abstraction is handled by the axiom for terms already in canonical form, as in the case of pairing. For function application, we show two rules, according to two different evaluation strategies, called *lazy* and *eager*. In the lazy operational semantics, we do not evaluate the canonical forms of the parameters when passing them to the function body. The lazy semantics will be our primary focus in the rest of this part of the book concerned with HOFL.

$$\frac{t_1 \to \lambda x.\, t_1' \quad t_1'[{}^{t_0}/_x] \to c}{(t_1\ t_0) \to c} \quad \text{(lazy)}$$

We remark that the in the second premise of the rule, we replace with t_0 each occurrence of x in t_1', i.e., we replace each instance of x with a copy of the (non evaluated) parameter t_0 and not with its canonical form.

For the sake of discussion let us consider the *eager* alternative to this rule.

$$\frac{t_1 \to \lambda x.\, t_1' \quad t_0 \to c_0 \quad t_1'[{}^{c_0}/_x] \to c}{(t_1\ t_0) \to c} \quad \text{(eager)}$$

Unlike the lazy semantics, the eager semantics evaluates the parameters only once and before the substitution. 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. Vice versa, according to the lazy semantics, if the argument is actually needed it may be later evaluated several times (every times it is used).

Finally, we have a last rule for recursive terms:

$$\frac{t[{}^{\mathbf{rec}\ x.\ t}/_x] \to c}{\mathbf{rec}\ x.\ t \to c}$$

To evaluate the canonical form of $\mathbf{rec}\ x.\ t$ we first plug in t the recursive definition itself in place of every occurrence of x and then compute the canonical form.

Example 7.6. Let us consider the term $t \stackrel{\text{def}}{=} \lambda x.\ 0+x$. Clearly the term t is closed and typable, with $t : \mathit{int} \to \mathit{int}$. It is already in canonical form and we have in fact:

$$t \to c \ \nwarrow_{c=\lambda x.\ 0+x} \Box$$

Example 7.7. Let us consider the term $t \stackrel{\text{def}}{=} \mathbf{rec}\ x.\ 0+x$. Clearly the term t is closed and typable, with $t : \mathit{int}$. We show that the term has no canonical form, in fact:

$$\begin{array}{rl} t \to c & \nwarrow\ (0+x)[{}^{t}/_{x}] \to c \\ & =\ 0+t \to c \\ & \nwarrow_{c=c_1+c_2}\ 0 \to c_1,\ t \to c_2 \\ & \nwarrow_{c_1=0}\ t \to c_2 \\ & \nwarrow\ \ldots \end{array}$$

Let us see an example which illustrates how rules are used to evaluate a function application.

Example 7.8 (Factorial). Let us consider the well-formed factorial function seen in the Example 7.1:

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

It is immediate to see that *fact* is closed and we know it has type $\mathit{int} \to \mathit{int}$. So we can calculate its canonical form by using the last rule seen and the axiom for terms in canonical form:

$$\frac{\lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (\mathit{fact}(x-1)) \to \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (\mathit{fact}(x-1))}{\mathit{fact} \to \lambda x.\ \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times (\mathit{fact}(x-1))}$$

We can apply this function to a specific value and calculate the canonical form of the result. For example, we see what is the canonical form c of the (closed and typable) term $(\mathit{fact}\ 2) : \mathit{int}$

$$
\begin{array}{rl}
(\mathit{fact}\ 2) \to c & \nwarrow\ \mathit{fact} \to \lambda x'.\, t',\quad t'[^2/_{x'}] \to c \\
& \nwarrow\ \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x \times (\mathit{fact}(x-1)) \to \lambda x'.\, t', \\
& \quad t'[^2/_{x'}] \to c \\
\nwarrow_{x'=x,\ t'=\textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x\times \mathit{fact}(x-1)} & \textbf{if}\ 2\ \textbf{then}\ 1\ \textbf{else}\ 2 \times (\mathit{fact}(2-1)) \to c \\
\nwarrow^* & 2 \times (\mathit{fact}(2-1)) \to c \\
\nwarrow_{c=c_1 \underline{\times} c_2} & 2 \to c_1,\quad (\mathit{fact}(2-1)) \to c_2 \\
\nwarrow^*_{c_1=2} & \mathit{fact} \to \lambda x''.\, t'',\quad \underbrace{t''[^{2-1}/_{x''}] \to c_2}_{\text{note that } 2-1 \text{ is not evaluated}} \\
\nwarrow_{x''=x,\ t''=\textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ x\times \mathit{fact}(x-1)} & \textbf{if}\ (2-1)\ \textbf{then}\ 1 \\
& \textbf{else}\ (2-1) \times (\mathit{fact}((2-1)-1)) \to c_2 \\
\nwarrow & 2-1 \to n,\quad n \neq 0, \\
& (2-1) \times \mathit{fact}((2-1)-1) \to c_2 \\
\nwarrow_{n=n_1 \underline{-} n_2} & 2 \to n_1,\quad 1 \to n_2,\quad n_1 \underline{-} n_2 \neq 0, \\
& \quad (2-1) \times \mathit{fact}((2-1)-1) \to c_2 \\
\nwarrow^*_{n_1=2,\ n_2=1} & (2-1) \times \mathit{fact}((2-1)-1) \to c_2 \\
\nwarrow^*_{c_2=c_3 \underline{\times} c_4} & 2-1 \to c_3,\quad \mathit{fact}((2-1)-1) \to c_4 \\
\nwarrow^*_{c_3=1} & \mathit{fact}((2-1)-1) \to c_4 \\
\nwarrow^* & \textbf{if}\ (2-1)-1\ \textbf{then}\ 1 \\
& \textbf{else}((2-1)-1) \times (\mathit{fact}(((2-1)-1)-1)) \to c_4 \\
\nwarrow & (2-1)-1 \to 0,\quad 1 \to c_4 \\
\nwarrow^*_{c_4=1} & \square
\end{array}
$$

So we have

$$c = c_1 \underline{\times} c_2 = 2 \underline{\times} (c_3 \underline{\times} c_4) = 2 \underline{\times} (1 \underline{\times} 1) = 2$$

Example 7.9 (Lazy vs eager evaluation). The aim of this example is to illustrate the difference between lazy and eager semantics. Let us consider the term

$$t \stackrel{\text{def}}{=} ((\lambda x : \mathit{int}.\ 3)(\textbf{rec}\ y : \mathit{int}.\ y)) : \mathit{int}$$

also written more concisely as

$$t \stackrel{\text{def}}{=} (\lambda x.\ 3)\, \textbf{rec}\ y.\ y$$

assuming $\widehat{x} = \widehat{y} = \mathit{int}$. It consists of the constant function $\lambda x.\ 3$ applied to a diverging term **rec** $y.\ y$ (i.e., a term with no canonical form).

- **Lazy evaluation**
  Lazy evaluation evaluates a parameter 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 our example:

$$\begin{array}{lll} ((\lambda x.\, 3)\,\mathbf{rec}\ y.\, y) \to c & \nwarrow & \lambda x.\, 3 \to \lambda x.\, t, \quad t[^{\mathbf{rec}\ y.\, y}/_x] \to c \\ & \nwarrow_{t=3} & 3[^{\mathbf{rec}\ y.\, y}/_x] \to c \\ & \nwarrow_{c=3} & \square \end{array}$$

So although the argument **rec** $y.\, y$ has 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 must be evaluated before the application, leading to a diverging computation:

$$\begin{array}{lll} ((\lambda x.\, 3)\,\mathbf{rec}\ y.\, y) \to c & \nwarrow & \lambda x.\, 3 \to \lambda x.\, t, \quad \mathbf{rec}\ y.\, y \to c_1, \quad t[^{c_1}/_x] \to c \\ & \nwarrow_{t=3} & \mathbf{rec}\ y.\, y \to c_1, \quad 3[^{c_1}/_x] \to c \\ & \nwarrow & \mathbf{rec}\ y.\, y \to c_1 \quad 3[^{c_1}/_x] \to c \\ & \nwarrow & \ldots \end{array}$$

  So the evaluation does not terminate.

However if the parameter of a function is used n times, the parameter would be evaluated n times (at most) in the lazy semantics and only once in the eager case.

We conclude this chapter by presenting a theorem that guarantees that

1. if a term can be reduced to a canonical form then it is unique (determinacy);
2. the evaluation of the canonical form preserves the type assignments (type preservation).

Theorem 7.2. *Let t be a closed and typable term.*

1. *For any canonical form c, c', if $t \to c$ and $t \to c'$ then $c = c'$*
2. *For any canonical form c and type τ, if $t \to c$ and $t : \tau$ then $c : \tau$*

Proof. Property 1 is proved by rule induction, taking the predicate

$$P(t \to c) \stackrel{\text{def}}{=} \forall c'.\, t \to c' \Rightarrow c = c'$$

We show only the case of the application rule, the remainder of the proof of the theorem, including the proof of Property 2, is left as an exercise (see Problem 7.11). We have the rule:

$$\frac{t_1 \to \lambda x.\, t_1' \quad t_1'[^{t_0}/_x] \to c}{(t_1\ t_0) \to c}$$

We assume the inductive hypotheses:

- $P(t_1 \to \lambda x.\, t_1') \stackrel{\text{def}}{=} \forall c'.\, t_1 \to c' \Rightarrow \lambda x.\, t_1' = c'$
- $P(t_1'[^{t_0}/_x] \to c) \stackrel{\text{def}}{=} \forall c'.\, t_1'[^{t_0}/_x] \to c' \Rightarrow c = c'$

We want to prove:

$$P((t_1\ t_0) \to c) \stackrel{\text{def}}{=} \forall c'.\ (t_1\ t_0) \to c' \Rightarrow c = c'$$

As usual, we assume the premise of the implication:

$$(t_1\ t_0) \to c'$$

From it, by goal reduction:

$$(t_1\ t_0) \to c' \quad \nwarrow \quad t_1 \to \lambda x'.\ t_1'', \quad t_1''[{}^{t_0}/_{x'}] \to c'$$

Then we have by the first inductive hypothesis:

$$\lambda x.\ t_1' = \lambda x'.\ t_1''$$

i.e., $x = x'$ and $t_1' = t_1''$. Then $t_1''[{}^{t_0}/_{x'}] = t_1'[{}^{t_0}/_{x}]$ and by the second inductive hypothesis we have $c = c'$. □

Problems

7.1. Let $x, y, w : \mathit{int}$, and $f : \mathit{int} \to (\mathit{int} \to \mathit{int})$. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ x\ \textbf{then}\ (\lambda y.\ (y + w))\ \textbf{else}\ (f\ w)$$

1. Compute the term $t[{}^{((f\ x)\ y)}/_{w}]$.
2. Compute the term $t[{}^{((f\ x)\ y)}/_{x}]$.

Hint: The exercise is about making practice with capture-avoiding substitutions. You are allowed to introduce additional (typed) variables if needed.

7.2. Is it possible to assign a type to the HOFL pre-term below? If yes, compute its principal type.

$$\textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ \textbf{snd}(x)\ \textbf{then}\ 1\ \textbf{else}\ f(\ \textbf{fst}(x)\ ,\ (\textbf{fst}(x)\ \textbf{snd}(x)))$$

7.3. A *list of positive numbers* is defined by the following syntax, where $n \in \mathbb{N}, n > 0$:

$$L\ ::=\ (n, 0)\ \mid\ (n, L)$$

For instance the list with 3 followed by 5 is represented by the term $(3, (5, 0))$.

1. Define a HOFL term t (closed and typable) such that the application $(t\,L)$ to a list L of 3 elements returns the last element of the list.
2. Is it possible to find a closed and typable HOFL term which returns the last element of a generic list?

7.4. Given the two HOFL terms

$$t_1 \stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ x+3$$
$$t_2 \stackrel{\text{def}}{=} \lambda z.\ \mathbf{fst}(z)+3$$

1. Compute their types.
2. Prove that, given the canonical form $c : \tau$, the two terms

$$((t_1\ 1)\ c) \qquad \text{and} \qquad (t_2\ (1,c))$$

yield the same canonical form.

7.5. Let us consider the HOFL term

$$map \stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ ((f\ \mathbf{fst}(x)),(f\ \mathbf{snd}(x)))$$

Show that *map* is a typable term and give its principal type. Then, compute the canonical form of the term

$$((map\ (\lambda x.\ 2\times x))\ (1,2))$$

7.6. Determine the type of the HOFL term

$$t \stackrel{\text{def}}{=} \mathbf{rec}\ x.\ ((\lambda y.\ \mathbf{if}\ y\ \mathbf{then}\ 0\ \mathbf{else}\ 0)\ x).$$

Then compute its operational semantics.

7.7. Recall the definition of *binomial coefficients* $\binom{n}{k}$ from Problem 4.13:

$$\binom{n}{0} \stackrel{\text{def}}{=} 1 \qquad \binom{n}{n} \stackrel{\text{def}}{=} 1 \qquad \binom{n+1}{k+1} \stackrel{\text{def}}{=} \binom{n}{k} + \binom{n}{k+1}.$$

where $n,k \in \mathbb{N}$ and $0 \le k \le n$. Consider the corresponding HOFL program:

$$\begin{aligned} t \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda n.\ \lambda k.\ \ & \mathbf{if}\ k\ \mathbf{then}\ 1 \\ & \mathbf{else\ if}\ n-k\ \mathbf{then}\ 1 \\ & \qquad \mathbf{else}\ ((f\ (n-1))\ k)+((f\ (n-1))\ (k-1)). \end{aligned}$$

Compute its type and evaluate the canonical form of the term $((t\ 2)\ 1)$.

7.8. Consider the Fibonacci sequence already found in Problem 4.14

$$F(0) \stackrel{\text{def}}{=} 1 \qquad F(1) \stackrel{\text{def}}{=} 1 \qquad F(n+2) \stackrel{\text{def}}{=} F(n+1)+F(n)$$

where $n \in \mathbb{N}$.

1. Write a well-formed, closed HOFL term $t : int \to int$ to compute F.

2. Compute the operational semantics of $(t\ 2)$

7.9. Check if the HOFL pre-term

$$\lambda x.\ \lambda y.\ \lambda z.\ \textbf{if}\ z\ \textbf{then}\ (y\ x)\ \textbf{else}\ (x\ y).$$

is typable, in which case give its type.

7.10. Let us consider the HOFL pre-term $t = \lambda x.\ (x\ x)$. Prove that it is not typable. Try to compute anyway the canonical form of the application $(t\ t)$. Given that any well-typed term without recursion has a canonical form, argue why the given term is not typable.

7.11. Complete the proof of Theorem 7.2.

7.12. Suppose we extend HOFL with the inference rule:

$$\frac{t_1 \to 0}{t_1 \times t_2 \to 0}$$

Prove that the property of determinacy

$$\forall t, c_1, c_2.\ t \to c_1 \ \wedge\ t \to c_2 \Rightarrow c_1 = c_2$$

is still valid. What if also the inference rule below is added?

$$\frac{t_2 \to 0}{t_1 \times t_2 \to 0}$$

7.13. Prove that typable terms are uniquely typed, i.e., that for any pre-term t and types τ, τ', if $t : \tau$ and $t : \tau'$ then $\tau = \tau'$.

Chapter 8
Domain Theory

Order, unity and continuity are human inventions just as truly as catalogues and encyclopedias. (Bertrand Russell)

Abstract As done for IMP, we would like to introduce the denotational semantics of HOFL, for which we need to develop a proper domain theory that is more sophisticated than the one presented in Chapter 5. 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, using structural induction, that the type constructors of HOFL correspond to domains which are equipped with adequate $CPO_\perp$ structures and that we can define useful continuous functions between them.

8.1 The Flat Domain of Integer Numbers $\mathbb{Z}_\perp$

The first domain we introduce is very simple: it consists of all the integers numbers together with a distinguished bottom element. It relies on a flat order in the sense of Example 5.5.

Definition 8.1 ($\mathbb{Z}_\perp$). We define the CPO with bottom $\mathbb{Z}_\perp = (\mathbb{Z} \cup \{\perp\}, \sqsubseteq)$ as follows:

- $\mathbb{Z}$ is the set of integer numbers;
- $\perp$ is a distinguished bottom element that we add to the purpose;
- $\forall x \in \mathbb{Z} \cup \{\perp\}.\ \perp \sqsubseteq x$ and $x \sqsubseteq x$

It is immediate to check that $\mathbb{Z}_\perp$ is a CPO with bottom, where $\perp$ is the bottom element and each chain has a lub because chains are all finite: they either contain 1 or 2 different elements.

Remark 8.1. Since in this chapter we present several different domains, each coming with its proper order relation and bottom element, we find it useful to annotate them with the name of the domain as a subscript to avoid ambiguities. For example, we can write $\perp_{\mathbb{Z}_\perp}$ to make explicit that we are referring to the bottom element of the

domain $\mathbb{Z}_\perp$. Also note that the subscript $\perp$ we attach to the name of the domain $\mathbb{Z}$ is just a tag and it should not be confused with the name of the bottom element itself: it is the standard way to indicate that the domain $\mathbb{Z}$ is enriched with a bottom element (e.g., we could have used a different notation like $\underline{\mathbb{Z}}$ to the same purpose).

8.2 Cartesian Product of Two Domains

Given two CPO$_\perp$s we can combine them to obtain another CPO$_\perp$ whose elements are pairs formed with one element from each CPO$_\perp$.

Definition 8.2. Let:

$$\mathscr{D} = (D, \sqsubseteq_D) \qquad \mathscr{E} = (E, \sqsubseteq_E)$$

be two CPO$_\perp$s. Now we define their Cartesian product domain

$$\mathscr{D} \times \mathscr{E} = (D \times E, \sqsubseteq_{D\times E})$$

1. whose elements are the pairs of elements from D and E; and
2. whose order $\sqsubseteq_{D\times E}$ is defined as follows:[1]

$$\forall d_0, d_1 \in D,\ \forall e_0, e_1 \in E.\ (d_0, e_0) \sqsubseteq_{D\times E} (d_1, e_1) \Leftrightarrow d_0 \sqsubseteq_D d_1 \wedge e_0 \sqsubseteq_E e_1$$

Proposition 8.1. *$(D \times E, \sqsubseteq_{D\times E})$ is a partial order with bottom.*

Proof. We need to show that the relation $\sqsubseteq_{D\times E}$ is reflexive, antisymmetric and transitive:

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 $(d_0, e_0) \sqsubseteq_{D\times E} (d_1, e_1)$ and $(d_1, e_1) \sqsubseteq_{D\times E} (d_0, e_0)$ so by definition of $\sqsubseteq_{D\times E}$ we have $d_0 \sqsubseteq_D d_1$ (using the first relation) and $d_1 \sqsubseteq_D d_0$ (by using the second relation) so it must be $d_0 = d_1$ and similarly $e_0 = e_1$, hence $(d_0, e_0) = (d_1, e_1)$.

transitivity: let us assume $(d_0, e_0) \sqsubseteq_{D\times E} (d_1, e_1)$ and $(d_1, e_1) \sqsubseteq_{D\times E} (d_2, e_2)$. By definition of $\sqsubseteq_{D\times E}$ we have $d_0 \sqsubseteq_D d_1$, $d_1 \sqsubseteq_D d_2$, $e_0 \sqsubseteq_E e_1$ and $e_1 \sqsubseteq_E e_2$. By transitivity of $\sqsubseteq_D$ and $\sqsubseteq_E$ we have $d_0 \sqsubseteq_D d_2$ and $e_0 \sqsubseteq_E e_2$. By definition of $\sqsubseteq_{D\times E}$ we get $(d_0, e_0) \sqsubseteq_{D\times E} (d_2, e_2)$.

Finally, we show that the there is a bottom element. Let $\perp_{D\times E} = (\perp_D, \perp_E)$. 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 $\mathscr{D} \times \mathscr{E}$.

[1] Note that the order is different from the lexicographic one considered in Example 4.9.

Theorem 8.1 (Completeness of $\mathscr{D} \times \mathscr{E}$). *The PO $\mathscr{D} \times \mathscr{E}$ defined above is complete.*

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

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

Obviously $(\bigsqcup_{i \in \mathbb{N}} d_i, \bigsqcup_{i \in \mathbb{N}} e_i)$ is an upper bound, indeed for each $j \in \mathbb{N}$ we have $d_j \sqsubseteq_D \bigsqcup_{i \in \mathbb{N}} d_i$ and $e_j \sqsubseteq_E \bigsqcup_{i \in \mathbb{N}} e_i$ so by definition of $\sqsubseteq_{D \times E}$ it holds $(d_j, e_j) \sqsubseteq_{D \times E} (\bigsqcup_{i \in \mathbb{N}} d_i, \bigsqcup_{i \in \mathbb{N}} e_i)$.
Moreover $(\bigsqcup_{i \in \mathbb{N}} d_i, \bigsqcup_{i \in \mathbb{N}} e_i)$ is also the least upper bound. Indeed, let (d, e) be an upper bound of $\{(d_i, e_i)\}_{i \in \mathbb{N}}$, since $\bigsqcup_{i \in \mathbb{N}} d_i$ is the lub of $\{d_i\}_{i \in \mathbb{N}}$ we have $\bigsqcup_{i \in \mathbb{N}} d_i \sqsubseteq_D d$, furthermore we have that $\bigsqcup_{i \in \mathbb{N}} e_i$ is the lub of $\{e_i\}_{i \in \mathbb{N}}$ then $\bigsqcup_{i \in \mathbb{N}} e_i \sqsubseteq_E e$. So by definition of $\sqsubseteq_{D \times E}$ we have $(\bigsqcup_{i \in \mathbb{N}} d_i, \bigsqcup_{i \in \mathbb{N}} e_i) \sqsubseteq_{D \times E} (d, e)$. Thus $(\bigsqcup_{i \in \mathbb{N}} d_i, \bigsqcup_{i \in \mathbb{N}} e_i)$ is the least upper bound. □

We can now define suitable projection operators over $\mathscr{D} \times \mathscr{E}$.

Definition 8.3 (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.

$$\pi_1((d, e)) \stackrel{\text{def}}{=} d \qquad \text{and} \qquad \pi_2((d, e)) \stackrel{\text{def}}{=} e.$$

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 preserves the order and limits) and so we can calculate its fixpoints to solve recursive equations. So we have to prove that each function which we use on $\mathscr{D} \times \mathscr{E}$ is continuous. The proof that projections are monotone is immediate and left as an exercise (see Problem 8.1).

Theorem 8.2 (Continuity of π_1 and π_2). *Let π_1 and π_2 be the projection functions in Definition 8.3 and let $\{(d_i, e_i)\}_{i \in \mathbb{N}}$ be a chain of elements in $\mathscr{D} \times \mathscr{E}$, then:*

$$\pi_1 \left(\bigsqcup_{i \in \mathbb{N}} (d_i, e_i) \right) = \bigsqcup_{i \in \mathbb{N}} \pi_1((d_i, e_i)) \qquad \pi_2 \left(\bigsqcup_{i \in \mathbb{N}} (d_i, e_i) \right) = \bigsqcup_{i \in \mathbb{N}} \pi_2((d_i, e_i))$$

Proof. Let us prove the first statement:

$$\begin{aligned} \pi_1 \left(\bigsqcup_{i \in \mathbb{N}} (d_i, e_i) \right) &= \pi_1 \left(\left(\bigsqcup_{i \in \mathbb{N}} d_i, \bigsqcup_{i \in \mathbb{N}} e_i \right) \right) && \text{(by definition of limit in } D \times E) \\ &= \bigsqcup_{i \in \mathbb{N}} d_i && \text{(by definition of projection)} \\ &= \bigsqcup_{i \in \mathbb{N}} \pi_1((d_i, e_i)) && \text{(by definition of projection).} \end{aligned}$$

For the second statement the proof is completely analogous. □

8.3 Functional Domains

Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be two CPOs. In the following we denote by $D \to E \stackrel{\text{def}}{=} \{f \mid f : D \to E\}$ the set of all functions from D to E (where the order relations are not important), while we denote by $[D \to E] \subseteq D \to E$ the set of all continuous functions from D to E (i.e., $[D \to E]$ contains just the functions that preserve order and limits). As for Cartesian product, we can define a suitable order on the set $[D \to E]$ to get a $\text{CPO}_\perp$. Note that as usual we require the continuity of the functions to preserve the applicability of fixpoint theory.

Definition 8.4. Let us consider the $\text{CPO}_\perp$s:

$$\mathscr{D} = (D, \sqsubseteq_D) \qquad \mathscr{E} = (E, \sqsubseteq_E)$$

We define an order on the set of continuous functions from D to E as follows:

$$[\mathscr{D} \to \mathscr{E}] = \left([D \to E], \sqsubseteq_{[D \to E]}\right)$$

where:

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

We leave as an exercise the proof that $[\mathscr{D} \to \mathscr{E}]$ is a PO with bottom, namely that the relation $\sqsubseteq_{[D \to E]}$ is reflexive, antisymmetric, transitive and that the function $\perp_{[D \to E]} : D \to E$ defined by letting, for any $d \in D$:

$$\perp_{[D \to E]}(d) \stackrel{\text{def}}{=} \perp_E$$

is continuous and that it is also the bottom element of $[\mathscr{D} \to \mathscr{E}]$ (see Problem 8.2).

We show that the PO $[\mathscr{D} \to \mathscr{E}]$ is complete. In order to simplify the proof we introduce first the following lemmas.

Lemma 8.1 (Switch Lemma). *Let $(E, \sqsubseteq_E)$ be a CPO whose elements are of the form $e_{n,m}$ with $n, m \in \mathbb{N}$. 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:

$$\bigsqcup_{n,m \in \mathbb{N}} e_{n,m} = \bigsqcup_{n \in \mathbb{N}} \left(\bigsqcup_{m \in \mathbb{N}} e_{n,m} \right) = \bigsqcup_{m \in \mathbb{N}} \left(\bigsqcup_{n \in \mathbb{N}} e_{n,m} \right) = \bigsqcup_{k \in \mathbb{N}} e_{k,k}$$

Proof. The relation between the elements of E can be summarized as follows:

$$
\begin{array}{ccccccccc}
\vdots & & \vdots & & \vdots & & \vdots & & \iddots \\
\sqcup & & \sqcup & & \sqcup & & \sqcup & & \\
e_{n,0} & \sqsubseteq & e_{n,1} & \sqsubseteq & e_{n,2} & \sqsubseteq \cdots \sqsubseteq & e_{n,m} & \sqsubseteq & \cdots \\
\sqcup & & \sqcup & & \sqcup & & \sqcup & & \\
\vdots & & \vdots & & \vdots & \iddots & \vdots & & \\
\sqcup & & \sqcup & & \sqcup & & \sqcup & & \\
e_{2,0} & \sqsubseteq & e_{2,1} & \sqsubseteq & e_{2,2} & \sqsubseteq \cdots \sqsubseteq & e_{2,m} & \sqsubseteq & \cdots \\
\sqcup & & \sqcup & & \sqcup & & \sqcup & & \\
e_{1,0} & \sqsubseteq & e_{1,1} & \sqsubseteq & e_{1,2} & \sqsubseteq \cdots \sqsubseteq & e_{1,m} & \sqsubseteq & \cdots \\
\sqcup & & \sqcup & & \sqcup & & \sqcup & & \\
e_{0,0} & \sqsubseteq & e_{0,1} & \sqsubseteq & e_{0,2} & \sqsubseteq \cdots \sqsubseteq & e_{0,m} & \sqsubseteq & \cdots
\end{array}
$$

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

$$
\{e_{n,m}\}_{n,m\in\mathbb{N}} \qquad \left\{\bigsqcup_{m\in\mathbb{N}} e_{n,m}\right\}_{n\in\mathbb{N}} \qquad \left\{\bigsqcup_{n\in\mathbb{N}} e_{n,m}\right\}_{m\in\mathbb{N}} \qquad \{e_{k,k}\}_{k\in\mathbb{N}}
$$

- Let us consider the first two sets. For any $n \in \mathbb{N}$, let $e_n = \bigsqcup_{j\in\mathbb{N}} e_{n,j}$. This amounts to consider each row of the above diagram and compute the least upper bound for the elements in the same row. Clearly, $e_{n_1} \sqsubseteq e_{n_2}$ when $n_1 \leq n_2$ because for any $j \in \mathbb{N}$ an upper bound of $e_{n_2,j}$ is also an upper bound of $e_{n_1,j}$.

$$
\begin{array}{ccccccccc}
\vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
\sqcup & & \sqcup & & \sqcup & & \sqcup & & \sqcup \\
e_{n,0} & \sqsubseteq & e_{n,1} & \sqsubseteq & e_{n,2} & \sqsubseteq \cdots \sqsubseteq & e_{n,m} & \sqsubseteq \cdots \sqsubseteq & e_n = \bigsqcup_{m\in\mathbb{N}} e_{n,m} \\
\sqcup & & \sqcup & & \sqcup & & \sqcup & & \sqcup \\
\vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
\sqcup & & \sqcup & & \sqcup & & \sqcup & & \sqcup \\
e_{2,0} & \sqsubseteq & e_{2,1} & \sqsubseteq & e_{2,2} & \sqsubseteq \cdots \sqsubseteq & e_{2,m} & \sqsubseteq \cdots \sqsubseteq & e_2 = \bigsqcup_{m\in\mathbb{N}} e_{2,m} \\
\sqcup & & \sqcup & & \sqcup & & \sqcup & & \sqcup \\
e_{1,0} & \sqsubseteq & e_{1,1} & \sqsubseteq & e_{1,2} & \sqsubseteq \cdots \sqsubseteq & e_{2,m} & \sqsubseteq \cdots \sqsubseteq & e_1 = \bigsqcup_{m\in\mathbb{N}} e_{1,m} \\
\sqcup & & \sqcup & & \sqcup & & \sqcup & & \sqcup \\
e_{0,0} & \sqsubseteq & e_{0,1} & \sqsubseteq & e_{0,2} & \sqsubseteq \cdots \sqsubseteq & e_{2,m} & \sqsubseteq \cdots \sqsubseteq & e_0 = \bigsqcup_{m\in\mathbb{N}} e_{0,m}
\end{array}
$$

Let e be an upper bound of $\{e_i\}_{i\in\mathbb{N}}$, we want to show that e is an upper bound for $\{e_{n,m}\}_{n,m\in\mathbb{N}}$. Take any $n,m \in \mathbb{N}$. Then

$$
e_{n,m} \sqsubseteq \bigsqcup_{j\in\mathbb{N}} e_{n,j} = e_n \sqsubseteq e
$$

since $e_{n,m}$ is an element of the chain $\{e_{n,j}\}_{j\in\mathbb{N}}$ whose limit is $e_n = \bigsqcup_{j\in\mathbb{N}} e_{n,j}$. Thus e is an upper bound for $\{e_{n,m}\}_{n,m\in\mathbb{N}}$.

Vice versa, let e be an upper bound of $\{e_{i,j}\}_{i,j\in\mathbb{N}}$ and consider $e_n = \bigsqcup_{m\in\mathbb{N}} e_{n,m}$ for some n. Since $\{e_{n,m}\}_{m\in\mathbb{N}} \subseteq \{e_{i,j}\}_{i,j\in\mathbb{N}}$, obviously e is an upper bound for $\{e_{n,m}\}_{m\in\mathbb{N}}$ and therefore $e_n \sqsubseteq e$, because e_n is the lub of $\{e_{n,m}\}_{m\in\mathbb{N}}$.

- The correspondence between the sets of upper bounds of $\{e_{n,m}\}_{n,m\in\mathbb{N}}$ and $\{\bigsqcup_{n\in\mathbb{N}} e_{n,m}\}_{m\in\mathbb{N}}$ can be proved analogously.
- Finally, let us consider the sets $\{e_{n,m}\}_{n,m\in\mathbb{N}}$ and $\{e_{k,k}\}_{k\in\mathbb{N}}$ and show that they have the same set of upper bounds.
  Taken any $n,m \in \mathbb{N}$ the element, let $k = \max\{n,m\}$. We have

$$e_{n,m} \sqsubseteq e_{n,k} \sqsubseteq e_{k,k}$$

  thus any upper bound of $\{e_{k,k}\}_{k\in\mathbb{N}}$ is also an upper bound of $\{e_{n,m}\}_{n,m\in\mathbb{N}}$.
  Vice versa, it is immediate to check that $\{e_{k,k}\}_{k\in\mathbb{N}}$ is a subset of $\{e_{n,m}\}_{n,m\in\mathbb{N}}$ so any upper bound of $\{e_{n,m}\}_{n,m\in\mathbb{N}}$ is also an upper bound of $\{e_{k,k}\}_{k\in\mathbb{N}}$.

We conclude by noting that the set of upper bounds $\{e_{n,m}\}_{n,m\in\mathbb{N}}$ has a least element. In fact, $\{\bigsqcup_{m\in\mathbb{N}} e_{n,m}\}_{n\in\mathbb{N}}$ is a chain, and it has a lub because E is a CPO. □

Lemma 8.2. *Let $\{f_n\}_{n\in\mathbb{N}}$ be a chain of functions[2] in $\mathscr{D} \to \mathscr{E}$. Then the lub $\bigsqcup_{n\in\mathbb{N}} f_n$ exists and it is defined as:*

$$\left(\bigsqcup_{n\in\mathbb{N}} f_n\right)(d) = \bigsqcup_{n\in\mathbb{N}} (f_n(d))$$

Proof. The function

$$h \stackrel{\text{def}}{=} \lambda d.\, \bigsqcup_{n\in\mathbb{N}} (f_n(d))$$

is clearly an upper bound for $\{f_n\}_{n\in\mathbb{N}}$ since for every $k \in \mathbb{N}$ and $d \in D$ we have $f_k(d) \sqsubseteq_E \bigsqcup_{n\in\mathbb{N}} f_n(d)$.

The function h is also the lub of $\{f_n\}_{n\in\mathbb{N}}$. In fact, given any other upper bound g, i.e., such that $f_n \sqsubseteq_{D\to E} g$ for any $n \in \mathbb{N}$, we have that for any $d \in D$ the element $g(d)$ is an upper bound of the chain $\{f_n(d)\}_{n\in\mathbb{N}}$ and therefore $\bigsqcup_{n\in\mathbb{N}}(f_n(d)) \sqsubseteq_E g(d)$. □

Lemma 8.3. *Let $\{f_n\}_{n\in\mathbb{N}}$ be a chain of continuous functions in $[\mathscr{D} \to \mathscr{E}]$ and let $\{d_n\}_{n\in\mathbb{N}}$ be a chain of $\mathscr{D}$. Then, the function*

$$h \stackrel{\text{def}}{=} \lambda d.\, \bigsqcup_{n\in\mathbb{N}} (f_n(d))$$

is continuous, namely

$$h\left(\bigsqcup_{m\in\mathbb{N}} d_m\right) = \bigsqcup_{m\in\mathbb{N}} h(d_m)$$

Furthermore, h is the lub of $\{f_n\}_{n\in\mathbb{N}}$ not only in $\mathscr{D} \to \mathscr{E}$ as stated by Lemma 8.2, but also in $[\mathscr{D} \to \mathscr{E}]$.

[2] Note that the f_n are not necessarily continuous, because we select $\mathscr{D} \to \mathscr{E}$ and not $[\mathscr{D} \to \mathscr{E}]$.

Proof.

$$
\begin{aligned}
h\left(\bigsqcup_{m\in\mathbb{N}} d_m\right) &= \bigsqcup_{n\in\mathbb{N}}\left(f_n\left(\bigsqcup_{m\in\mathbb{N}} d_m\right)\right) && \text{(by definition of } h\text{)}\\
&= \bigsqcup_{n\in\mathbb{N}}\left(\bigsqcup_{m\in\mathbb{N}} (f_n(d_m))\right) && \text{(by continuity of } f_n\text{)}\\
&= \bigsqcup_{m\in\mathbb{N}}\left(\bigsqcup_{n\in\mathbb{N}} (f_n(d_m))\right) && \text{(by Switch Lemma 8.1)}\\
&= \bigsqcup_{m\in\mathbb{N}} h(d_m) && \text{(by definition of } h\text{)}
\end{aligned}
$$

Note that, in the previous passages, the premises for applying the Switch Lemma 8.1 because the elements $\{f_n(d_m)\}_{n,m\in\mathbb{N}}$ are in the CPO E and they satisfy $f_n(d_m) \sqsubseteq_E f_{n'}(d_{m'})$ whenever $n \leq n'$ and $m \leq m'$ as $f_n(d_m) \sqsubseteq_E f_n(d_{m'})$ by monotonicity of f_n (because $d_m \sqsubseteq_D d_{m'}$) and $f_n(d_{m'}) \sqsubseteq_E f_{n'}(d_{m'})$ because $f_n \sqsubseteq_{[D\to E]} f_{n'}$. The upper bounds of $\{f_n\}_{n\in\mathbb{N}}$ in the PO $\mathscr{D} \to \mathscr{E}$ are a larger set then those in $[\mathscr{D} \to \mathscr{E}]$, thus if h is the lub in $\mathscr{D} \to \mathscr{E}$, it is also the lub in $[\mathscr{D} \to \mathscr{E}]$. □

Theorem 8.3 ($[\mathscr{D} \to \mathscr{E}]$ is a CPO$_\perp$). *The PO $[\mathscr{D} \to \mathscr{E}]$ is a CPO$_\perp$*

Proof. The statement follows immediately from the previous lemmas. □

8.4 Lifting

In IMP we introduced a lifting operator (see Definition 6.9) on functions $f : \Sigma \to \Sigma_\perp$ to derive a function $f^* : \Sigma_\perp \to \Sigma_\perp$ defined over the lifted domain $\Sigma_\perp$, and thus able to handle the argument $\perp_{\Sigma_\perp}$. In the semantics of HOFL we need the same operator in a more general fashion: we need to apply the lifting operator to any domain, not just Σ.

Definition 8.5 (Lifted domain). Let $\mathscr{D} = (D, \sqsubseteq_D)$ be a CPO and let $\perp$ be an element not in D. We define the lifted domain $\mathscr{D}_\perp = (D_\perp, \sqsubseteq_{D_\perp})$ as follows:

- $D_\perp \stackrel{\text{def}}{=} \{\perp\} \uplus D = \{(0,\perp)\} \cup \{1\} \times D$
- $\perp_{D_\perp} \stackrel{\text{def}}{=} (0,\perp)$
- $\forall x \in D_\perp.\ \perp_{D_\perp} \sqsubseteq_{D_\perp} x$
- $\forall d_1, d_2 \in D.\ d_1 \sqsubseteq_D d_2 \Rightarrow (1,d_1) \sqsubseteq_{D_\perp} (1,d_2)$

We leave it as an exercise to show that $\mathscr{D}_\perp$ is a CPO$_\perp$ (see Problem 8.3).
We define a lifting function $\lfloor\cdot\rfloor : D \to D_\perp$ by letting, for any $d \in D$:

$$\lfloor d \rfloor \stackrel{\text{def}}{=} (1,d)$$

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 $[D \to E]$ to continuous functions in $[D_\perp \to E]$. The function defining the extension should itself be continuous.

Definition 8.6 (Lifting). Let $\mathscr{D}$ be a CPO and let $\mathscr{E}$ be a $CPO_\perp$. We define the lifting operator $(\cdot)^* : [D \to E] \to [D_\perp \to E]$ as follows:

$$\forall f \in [D \to E] \,.\, f^*(x) \stackrel{\text{def}}{=} \begin{cases} \perp_E & \text{if } x = \perp_{D_\perp} \\ f(d) & \text{if } x = \lfloor d \rfloor \text{ for some } d \in D \end{cases}$$

We need to prove that the definition is well-given and that the lifting operator is continuous.

Theorem 8.4. *Let $\mathscr{D}, \mathscr{E}$ be two CPOs.*

1. *If $f : D \to E$ is continuous, then f^* is continuous.*
2. *The operator $(\cdot)^*$ is continuous.*

Proof. We prove the two statements separately.

1. We need to prove that if $f \in [D \to E]$, then $f^* \in [D_\perp \to E]$. Let $\{x_n\}_{n\in\mathbb{N}}$ be a chain in $\mathscr{D}_\perp$. We have to prove $f^*(\bigsqcup_{n\in\mathbb{N}} x_n) = \bigsqcup_{n\in\mathbb{N}} f^*(x_n)$.
   If $\forall n \in \mathbb{N}.\, x_n = \perp_{D_\perp}$, then this is obvious.
   Otherwise, for some $k \in \mathbb{N}$ there must exist a set of elements $\{d_{n+k}\}_{n\in\mathbb{N}}$ in D such that for all $m \geq k$ we have $x_m = \lfloor d_m \rfloor$ and also $\bigsqcup_{n\in\mathbb{N}} x_n = \bigsqcup_{n\in\mathbb{N}} x_{n+k} = \lfloor \bigsqcup_{n\in\mathbb{N}} d_{n+k} \rfloor$ (by prefix independence of the limit, Lemma 5.1). Then:

$$\begin{aligned}
f^*\left(\bigsqcup_{n\in\mathbb{N}} x_n\right) &= f^*\left(\left\lfloor \bigsqcup_{n\in\mathbb{N}} d_{n+k} \right\rfloor\right) && \text{by the above argument} \\
&= f\left(\bigsqcup_{n\in\mathbb{N}} d_{n+k}\right) && \text{by definition of lifting} \\
&= \bigsqcup_{n\in\mathbb{N}} f(d_{n+k}) && \text{by continuity of } f \\
&= \bigsqcup_{n\in\mathbb{N}} f^*(\lfloor d_{n+k} \rfloor) && \text{by definition of lifting} \\
&= \bigsqcup_{n\in\mathbb{N}} f^*(x_{n+k}) && \text{by definition of } x_{n+k} \\
&= \bigsqcup_{n\in\mathbb{N}} f^*(x_n) && \text{by Lemma 5.1}
\end{aligned}$$

2. We leave the proof that $(\cdot)^*$ is monotone as an exercise (see Problem 8.4).
   Let $\{f_i\}_{i\in\mathbb{N}}$ be a chain of functions in $[\mathscr{D} \to \mathscr{E}]$. We will prove that for all $x \in D_\perp$:

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

if $x = \bot_{D_\bot}$ both sides of the equation simplify to $\bot_E$. So let us assume $x = \lfloor d \rfloor$ for some $d \in D$ we have:

$$\begin{aligned}
\left(\bigsqcup_{i\in\mathbb{N}} f_i\right)^* (\lfloor d \rfloor) &= \left(\bigsqcup_{i\in\mathbb{N}} f_i\right)(d) && \text{by definition of lifting} \\
&= \bigsqcup_{i\in\mathbb{N}} (f_i(d)) && \text{by def. of lub in a functional domain} \\
&= \bigsqcup_{i\in\mathbb{N}} (f_i^*(\lfloor d \rfloor)) && \text{by definition of lifting} \\
&= \left(\bigsqcup_{i\in\mathbb{N}} f_i^*\right)(\lfloor d \rfloor) && \text{by def. of lub in a functional domain}
\end{aligned}$$

□

8.5 Function's Continuity Theorems

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

Theorem 8.5 (Continuity of composition). *Let $f \in [D \to E]$ and $g \in [E \to F]$. Their composition*

$$f;g = g \circ f \stackrel{\text{def}}{=} \lambda d.\ g(f(d)) : D \to F$$

is a continuous function, i.e., $g \circ f \in [D \to F]$.

Proof. The statement is just a rephrasing of Theorem 5.5. □

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

$$f : D \to E_1 \times E_2$$

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 $g_1 : D \to E_1$ and $g_2 : D \to E_2$, such that $f(d) = (g_1(d), g_2(d))$ for any $d \in D$. Then we can prove the continuity of f from the continuity of g_1 and g_2 (and vice versa).

Theorem 8.6. *Let $f : D \to E_1 \times E_2$ be a function over CPOs and let*

$$g_1 \stackrel{\text{def}}{=} f;\pi_1 : D \to E_1 \qquad g_2 \stackrel{\text{def}}{=} f;\pi_2 : D \to E_2$$

where $f;\pi_i = \lambda x.\ \pi_i(f(x))$ is the composition of f and π_i for $i = 1,2$. Then: f is continuous if and only if g_1 and g_2 are continuous.

Proof. Notice that we have

$$\forall d \in D.\ f(d) = (g_1(d), g_2(d))$$

We prove the two implications separately.

⇒) Since f is continuous, by Theorem 8.5 (continuity of composition) and Theorem 8.2 (continuity of projections), also g_1 and g_2 are continuous, because they are obtained as the composition of continuous functions.

⇐) We assume the continuity of g_1 and g_2 and we want to prove that f is continuous. Let $\{d_i\}_{i\in\mathbb{N}}$ be a chain in D. We want to prove:

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

So we have:

$$\begin{aligned}
f\left(\bigsqcup_{i\in\mathbb{N}} d_i\right) &= \left(g_1\left(\bigsqcup_{i\in\mathbb{N}} d_i\right), g_2\left(\bigsqcup_{i\in\mathbb{N}} d_i\right)\right) && \text{(by definition of } g_1, g_2\text{)}\\
&= \left(\bigsqcup_{i\in\mathbb{N}} g_1(d_i), \bigsqcup_{i\in\mathbb{N}} g_2(d_i)\right) && \text{(by continuity of } g_1 \text{ and } g_2\text{)}\\
&= \bigsqcup_{i\in\mathbb{N}} (g_1(d_i), g_2(d_i)) && \text{(by definition of lub of pairs)}\\
&= \bigsqcup_{i\in\mathbb{N}} f(d_i) && \text{(by definition of } g_1, g_2\text{)}
\end{aligned}$$

□

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

Theorem 8.7. *Let $f : D_1 \times D_2 \to E$ be a function over CPOs. Then f is continuous if and only if all the functions in the following two classes are continuous:*

1. $\forall d' \in D_1.\ f_{d'} : D_2 \to E$ is defined as $f_{d'} \stackrel{\text{def}}{=} \lambda y.f(d', y)$;
2. $\forall d'' \in D_2.\ f_{d''} : D_1 \to E$ is defined as $f_{d''} \stackrel{\text{def}}{=} \lambda x.f(x, d'')$.

Proof. We prove the two implications separately:

⇒) If f is continuous then for all $d' \in D_1, d'' \in D_2$ the functions $f_{d'}$ and $f_{d''}$ are continuous, since we are considering only certain chains (where one element of the pair is fixed). For example, let us fix $d' \in D_1$ and consider a chain $\{d''_i\}_{i\in\mathbb{N}}$ in D_2. Then we prove that $f_{d'}$ is continuous as follows:

$$
\begin{aligned}
f_{d'}\left(\bigsqcup_{i\in\mathbb{N}} d_i''\right) &= f\left(d', \bigsqcup_{i\in\mathbb{N}} d_i''\right) && \text{(by definition of } f_{d'}\text{)}\\
&= f\left(\bigsqcup_{i\in\mathbb{N}} (d', d_i'')\right) && \text{(by definition of lub)}\\
&= \bigsqcup_{i\in\mathbb{N}} f(d', d_i'') && \text{(by continuity of } f\text{)}\\
&= \bigsqcup_{i\in\mathbb{N}} f_{d'}(d_i'') && \text{(by definition of } f_{d'}\text{)}
\end{aligned}
$$

Similarly, if we fix $d'' \in D_2$ and take a chain $\{d_i'\}_{i\in\mathbb{N}}$ in D_1 we have $f_{d''}(\bigsqcup_{i\in\mathbb{N}} d_i') = \bigsqcup_{i\in\mathbb{N}} f_{d''}(d_i')$.

$\Leftarrow$) In the opposite direction, assume that $f_{d'}$ and $f_{d''}$ are continuous for all elements $d' \in D_1$ and $d'' \in D_2$. We want to prove that f is continuous. Take a chain $\{(d_k', d_k'')\}_{k\in\mathbb{N}}$. By definition of lub on pairs, we have

$$
\bigsqcup_{k\in\mathbb{N}} (d_k', d_k'') = \left(\bigsqcup_{i\in\mathbb{N}} d_i', \bigsqcup_{j\in\mathbb{N}} d_j''\right)
$$

Let $d'' \stackrel{\text{def}}{=} \bigsqcup_{j\in\mathbb{N}} d_j''$. It follows:

$$
\begin{aligned}
f\left(\bigsqcup_{k\in\mathbb{N}}(d'_k,d''_k)\right) &= f\left(\bigsqcup_{i\in\mathbb{N}}d'_i,\bigsqcup_{j\in\mathbb{N}}d''_j\right) && \text{(by definition of lub on pairs)}\\
&= f\left(\bigsqcup_{i\in\mathbb{N}}d'_i,d''\right) && \text{(by definition of } d''\text{)}\\
&= f_{d''}\left(\bigsqcup_{i\in\mathbb{N}}d'_i\right) && \text{(by definition of } f_{d''}\text{)}\\
&= \bigsqcup_{i\in\mathbb{N}} f_{d''}(d'_i) && \text{(by continuity of } f_{d''}\text{)}\\
&= \bigsqcup_{i\in\mathbb{N}} f(d'_i,d'') && \text{(by definition of } f_{d''}\text{)}\\
&= \bigsqcup_{i\in\mathbb{N}} f_{d'_i}(d'') && \text{(by definition of } f_{d'_i}\text{)}\\
&= \bigsqcup_{i\in\mathbb{N}} f_{d'_i}\left(\bigsqcup_{j\in\mathbb{N}}d''_j\right) && \text{(by definition of } d''\text{)}\\
&= \bigsqcup_{i\in\mathbb{N}}\bigsqcup_{j\in\mathbb{N}} f_{d'_i}(d''_j) && \text{(by continuity of } f_{d'_i}\text{)}\\
&= \bigsqcup_{i\in\mathbb{N}}\bigsqcup_{j\in\mathbb{N}} f(d'_i,d''_j) && \text{(by definition of } f_{d'_i}\text{)}\\
&= \bigsqcup_{k\in\mathbb{N}} f(d'_k,d''_k) && \text{(by Lemma 8.1 (switch lemma))}
\end{aligned}
$$

□

8.6 Apply, Curry and Fix

As done for IMP we will use the λ-notation as meta-language for the denotational semantics of HOFL. In Section 8.2 we have already defined two new continuous functions for our meta-language (π_1 and π_2). In this section we introduce some additional functions that will form the kernel of our meta-language.

Definition 8.7 (Apply). Let D and E be two CPOs. We define a function apply : $[D \to E] \times D \to E$ as follows:

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

The function apply represents the application of a function in our meta-language: it takes a continuous function $f : D \to E$ and an element $d \in D$ and then returns $f(d)$ as

a result. We leave it as an exercise to prove that apply is monotone (see Problem 8.5). We prove that it is also continuous.

Theorem 8.8 (Continuity of apply). *Let* apply : $[D \to E] \times D \to E$ *be the function defined above and let* $\{(f_n, d_n)\}_{n\in\mathbb{N}}$ *be a chain in the CPO*$_\perp$ $[\mathscr{D} \to \mathscr{E}] \times \mathscr{D}$ *then:*

$$\text{apply}\left(\bigsqcup_{n\in\mathbb{N}} (f_n, d_n)\right) = \bigsqcup_{n\in\mathbb{N}} \text{apply}(f_n, d_n)$$

Proof. By Theorem 8.7 we can prove the continuity on each parameter separately.

- Let us fix $d \in D$ and take a chain $\{f_n\}_{n\in\mathbb{N}}$ in $[D \to E]$. We have:

$$\begin{aligned}
\text{apply}\left(\left(\bigsqcup_{n\in\mathbb{N}} f_n\right), d\right) &= \left(\bigsqcup_{n\in\mathbb{N}} f_n\right)(d) && \text{(by definition)} \\
&= \bigsqcup_{n\in\mathbb{N}} (f_n(d)) && \text{(by definition of lub of functions)} \\
&= \bigsqcup_{n\in\mathbb{N}} \text{apply}(f_n, d) && \text{(by definition)}
\end{aligned}$$

- Now we fix $f \in [D \to E]$ and take a chain $\{d_n\}_{n\in\mathbb{N}}$ in D. We have:

$$\begin{aligned}
\text{apply}\left(f, \bigsqcup_{n\in\mathbb{N}} d_n\right) &= f\left(\bigsqcup_{n\in\mathbb{N}} d_n\right) && \text{(by definition)} \\
&= \bigsqcup_{n\in\mathbb{N}} f(d_n) && \text{(by continuity of } f\text{)} \\
&= \bigsqcup_{n\in\mathbb{N}} \text{apply}(f, d_n) && \text{(by definition)}
\end{aligned}$$

So apply is a continuous function. □

Currying is the name of a technique for transforming a function that takes a pair (or, more generally, a tuple) of arguments into a function that takes each argument separately but computes the same result.

Definition 8.8 (Curry and un-curry). We define the function

$$\text{curry} : (D \times E \to F) \to (D \to E \to F)$$

by letting, for any $g : D \times E \to F$, $d \in D$ and $e \in E$:

$$\text{curry}\ g\ d\ e \stackrel{\text{def}}{=} g(d, e)$$

And we define the function

$$\text{un-curry} : (D \to E \to F) \to (D \times E \to F)$$

by letting, for any $h : D \to E \to F$, $d \in D$ and $e \in E$:

$$\text{un-curry } h\ (d,e) \stackrel{\text{def}}{=} h\ d\ e$$

Theorem 8.9 (Continuity of curry). *Let D,E,F be CPOs and $g : D \times E \to F$ be a continuous function. Then* $(\text{curry}\, g) : D \to (E \to F)$ *is a continuous function, namely given any chain $\{d_i\}_{i\in\mathbb{N}}$ in D:*

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

Proof. Let us note that since g is continuous, by Theorem 8.7, g is continuous separately on each argument. Then let us take $e \in E$ we have:

$$\begin{aligned}
(\text{curry } g)\left(\bigsqcup_{i\in\mathbb{N}} d_i\right)(e) &= g\left(\left(\bigsqcup_{i\in\mathbb{N}} d_i\right), e\right) && \text{(by definition of curry } g\text{)}\\
&= \bigsqcup_{i\in\mathbb{N}} g(d_i, e) && \text{(by continuity of } g\text{)}\\
&= \bigsqcup_{i\in\mathbb{N}} ((\text{curry } g)(d_i)(e)) && \text{(by definition of curry } g\text{)}
\end{aligned}$$

□

To define the denotational semantics of recursive definitions we need to provide a fixpoint operator. So it seems useful to introduce fix in our meta-language.

Definition 8.9 (Fix). Let D be a $CPO_\perp$. We define $\text{fix} : [D \to D] \to D$ as:

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

Note that, since $\{\lambda f.\ f^i(\perp_D)\}_{i\in\mathbb{N}}$ is a chain of functions and $[D \to D] \to D$ is complete, we are guaranteed that the lub $\bigsqcup_{i\in\mathbb{N}} \lambda f.f^i(\perp_D)$ exists.

Theorem 8.10 (Continuity of fix). *The function* $\text{fix} : [D \to D] \to D$ *is continuous, namely* $\text{fix} \in [[D \to D] \to D]$.

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

Base case: $\lambda f.\ f^0(\perp_D) = \lambda f.\ \perp_D$ is a constant, and thus continuous, function.

Inductive case: Let us assume that $g \stackrel{\text{def}}{=} \lambda f.\ f^i(\perp_D)$ is continuous, i.e., that given a chain $\{f_n\}_{n\in\mathbb{N}}$ in $[D \to D]$ we have $g\left(\bigsqcup_{n\in\mathbb{N}} f_n\right) = \bigsqcup_{n\in\mathbb{N}} g(f_n)$, and let us prove that $h \stackrel{\text{def}}{=} \lambda f.\ f^{i+1}(\perp_D)$ is continuous, namely that $h\left(\bigsqcup_{n\in\mathbb{N}} f_n\right) = \bigsqcup_{n\in\mathbb{N}} h(f_n)$. In fact we have:

$$
\begin{aligned}
h\left(\bigsqcup_{n\in\mathbb{N}} f_n\right) &= \left(\bigsqcup_{n\in\mathbb{N}} f_n\right)^{i+1}(\bot_D) && \text{(by def. of } h\text{)}\\
&= \left(\bigsqcup_{n\in\mathbb{N}} f_n\right)\left(\left(\bigsqcup_{n\in\mathbb{N}} f_n\right)^{i}(\bot_D)\right) && \text{(by def. of } (\cdot)^{i+1}\text{)}\\
&= \left(\bigsqcup_{n\in\mathbb{N}} f_n\right)\left(g\left(\bigsqcup_{n\in\mathbb{N}} f_n\right)\right) && \text{(by def. of } g\text{)}\\
&= \left(\bigsqcup_{n\in\mathbb{N}} f_n\right)\left(\bigsqcup_{n\in\mathbb{N}} g(f_n)\right) && \text{(by ind. hyp.)}\\
&= \left(\bigsqcup_{n\in\mathbb{N}} f_n\right)\left(\bigsqcup_{n\in\mathbb{N}} f_n^i(\bot_D)\right) && \text{(by def of } g\text{)}\\
&= \bigsqcup_{n\in\mathbb{N}}\left(f_n\left(\bigsqcup_{m\in\mathbb{N}} f_m^i(\bot_D)\right)\right) && \text{(by def. of lub)}\\
&= \bigsqcup_{n\in\mathbb{N}}\bigsqcup_{m\in\mathbb{N}} f_n\left(f_m^i(\bot_D)\right) && \text{(by cont. of } f_n\text{)}\\
&= \bigsqcup_{k\in\mathbb{N}} f_k\left(f_k^i(\bot_D)\right) && \text{(by Lemma 8.1)}\\
&= \bigsqcup_{k\in\mathbb{N}} f_k^{i+1}(\bot_D) && \text{(by def. of } (\cdot)^{i+1}\text{)}\\
&= \bigsqcup_{n\in\mathbb{N}} h(f_n) && \text{(by def. of } h\text{)}
\end{aligned}
$$

□

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

Definition 8.10 (Let operator). Let $\mathscr{E}$ be a $CPO_\bot$ and $\lambda x.\ e$ a function in $[D \to E]$. We define the *let* operator as follows, where $d' \in D_\bot$:

$$
\textbf{let } x \Leftarrow d'.\ e \stackrel{\text{def}}{=} \underbrace{\underbrace{\underbrace{(\lambda x.\ e)}_{D\to E}}_{D_\bot \to E}\ (\underbrace{d'}_{D_\bot})}_{E} = \begin{cases} \bot_E & \text{if } d' = \bot_{D_\bot} \\ e\left[{}^{d}/_{x}\right] & \text{if } d' = \lfloor d \rfloor \text{ for some } d \in D \end{cases}
$$

Intuitively, taken $d' \in D_\bot$, if $d' = \bot$ then **let** $x \Leftarrow d'.\ e$ returns $\bot_E$, otherwise it means that $d' = \lfloor d \rfloor$ for some $d \in D$ and thus it returns $e\left[{}^{d}/_{x}\right]$, as if $\lambda x.\ e$ was applied to d, i.e., $d' = \lfloor d \rfloor$ is de-lifted so that $\lambda x.\ e$ can be used.

Problems

8.1. Prove that the projection functions in Definition 8.3 are monotone.

8.2. Prove that the domain $[\mathscr{D} \to \mathscr{E}]$ from Definition 8.4 is a $\text{CPO}_\perp$.

8.3. Prove that the lifted domain $\mathscr{D}_\perp$ from Definition 8.5 is a $\text{CPO}_\perp$.

8.4. Complete the proof of Theorem 8.4 for what is concerned with the monotonicity of the lifting function $(\cdot)^*$.

8.5. Prove that the function apply : $[D \to E] \times D \to E$ introduced in Definition 8.7 is monotone.

8.6. Let D be a CPO and $f : D \to D$ be a continuous function. Prove that the set of fixpoints of f is itself a CPO (under the order inherited from D).

8.7. Let D and E be two $\text{CPO}_\perp$s. A function $f : D \to E$ is called *strict* if $f(\perp_D) = \perp_E$. Prove that the set of strict functions from D to E is a $\text{CPO}_\perp$ under the usual order.

8.8. Let D and E be two CPOs. Prove that the following two definitions of the order between continuous functions $f, g : D \to E$ are equivalent.

1. $f \sqsubseteq g \;\Leftrightarrow\; \forall d \in D.\ f(d) \sqsubseteq_E g(d)$.
2. $f \preceq g \;\Leftrightarrow\; \forall d_1, d_2 \in D.\ (d_1 \sqsubseteq_D d_2 \Rightarrow f(d_1) \sqsubseteq_E g(d_2))$

8.9. Let $\mathscr{D} = (D, \sqsubseteq_D)$ and $\mathscr{E} = (E, \sqsubseteq_E)$ be two CPOs. Their sum $\mathscr{D} + \mathscr{E}$ has:

1. The set of elements

$$\{\perp\} \uplus D \uplus E = \{(0, \perp)\} \cup \{1\} \times ((\{0\} \times D) \cup (\{1\} \times E))$$

2. The order relation $\sqsubseteq_{D+E}$ defined by letting:
   - $(1, (0, d_1)) \sqsubseteq_{D+E} (1, (0, d_2))$ if $d_1 \sqsubseteq_D d_2$;
   - $(1, (1, e_1)) \sqsubseteq_{D+E} (1, (1, e_2))$ if $e_1 \sqsubseteq_E e_2$;
   - $\forall x \in \{\perp\} \uplus D \uplus E.\ (0, \perp) \sqsubseteq_{D+E} x$.

Prove that $\mathscr{D} + \mathscr{E}$ is a $\text{CPO}_\perp$.

8.10. Prove that un-curry is continuous and inverse to curry (see Definition 8.8).

Chapter 9
Denotational Semantics of HOFL

Work out what you want to say before you decide how you want to say it. (Christopher Strachey's first law of logical design)

Abstract In this chapter we exploit the domain theory from Chapter 8 to define the (lazy) denotational semantics of HOFL. For each type τ we introduce a corresponding domain $(V_\tau)_\perp$ which is defined inductively over the structure of τ and such that we can assign an element of the domain $(V_\tau)_\perp$ to each (closed and typable) term t with type τ. Moreover, we introduce the notion of environment, which assigns meanings to variables, and that can be exploited to define the denotational semantics of (typable) terms with variables. Interestingly, all constructions we use are continuous, so that we are able to assign meaning also to any (typable) term that is recursively defined. We conclude the chapter by showing some important properties of the denotational semantics; in particular, that it is compositional.

9.1 HOFL Semantic Domains

In order to specify the denotational semantics of a programming language, we have to define, by structural recursion, an interpretation function from each syntactic domain to a semantic domain. In IMP there are three syntactic domains, *Aexp* for arithmetic expressions, *Bexp* for boolean expressions and *Com* for commands. Correspondingly, we have defined three semantics domains and three interpretation functions ($\mathscr{A}[\![\cdot]\!]$, $\mathscr{B}[\![\cdot]\!]$ and $\mathscr{C}[\![\cdot]\!]$). HOFL has a sole syntactic domain (i.e., the set of well-formed terms t) and thus we have only one interpretation function, written $[\![\cdot]\!]$. However, since HOFL terms are typed, the interpretation function is parametric w.r.t. the type τ of t and we have one semantic domain V_τ for each type τ. Actually, 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 element $\perp_{(V_\tau)_\perp}$ assigns a meaning to all the terms of type τ without a canonical form. Moreover, we will need to handle terms with free variables, as, e.g., when defining the denotational semantics of $\lambda x.\, t$ in terms of the denotational semantics of t (with x possibly in $\mathrm{fv}(t)$). This was not the case for the operational semantics of HOFL, where only closed terms are considered. As terms may contain free variables, we pass to the interpretation function an *environment*

$$\rho \in Env \stackrel{\text{def}}{=} Var \to \bigcup_{\tau} (V_\tau)_\perp$$

which assigns meaning to variables. For consistency reasons, any environment ρ that we consider must satisfy the condition $\rho(x) \in (V_\tau)_\perp$ whenever $x : \tau$. Thus, we have

$$[\![t : \tau]\!] : Env \to (V_\tau)_\perp.$$

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

$$\begin{array}{rcl@{\qquad}rcl} V_{int} & \stackrel{\text{def}}{=} & \mathbb{Z} & (V_{int})_\perp & \stackrel{\text{def}}{=} & \mathbb{Z}_\perp \\ V_{\tau_1 * \tau_2} & \stackrel{\text{def}}{=} & (V_{\tau_1})_\perp \times (V_{\tau_2})_\perp & (V_{\tau_1 * \tau_2})_\perp & \stackrel{\text{def}}{=} & ((V_{\tau_1})_\perp \times (V_{\tau_2})_\perp)_\perp \\ V_{\tau_1 \to \tau_2} & \stackrel{\text{def}}{=} & [(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp] & (V_{\tau_1 \to \tau_2})_\perp & \stackrel{\text{def}}{=} & [(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp]_\perp \end{array}$$

Notice that the recursive definition above takes advantage of the domain constructors we have defined in Chapter 8. While the lifiting $\mathbb{Z}_\perp$ of the integer numbers $\mathbb{Z}$ 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 by the end of Chapter 10.

9.2 HOFL Interpretation Function

Now we are ready to define the interpretation function, by structural recursion. We briefly comment on each definition and show that the clauses of the structural recursion are typed correctly.

9.2.1 Constants

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

$$[\![n]\!]\rho \stackrel{\text{def}}{=} \lfloor n \rfloor$$

At the level of types we have:

$$\underbrace{[\![\underbrace{n}_{int}]\!]\rho}_{(V_{int})_\perp = \mathbb{Z}_\perp} = \lfloor \underbrace{n}_{\mathbb{Z}} \rfloor_{\mathbb{Z}_\perp}$$

9.2.2 Variables

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

$$[\![x]\!]\,\rho \stackrel{\text{def}}{=} \rho(x)$$

It is obvious that the typing is respected (under the assumption that $\rho(x) \in (V_\tau)_\perp$ whenever $x : \tau$):

$$\underbrace{[\![\underbrace{x}_{\tau}]\!]\rho}_{(V_\tau)_\perp} = \underbrace{\rho(\underbrace{x}_{\tau})}_{(V_\tau)_\perp}$$

9.2.3 Arithmetic Operators

We give the generic semantics of a binary operator $\mathrm{op} \in \{+,-,\times\}$ as:

$$[\![t_0 \ \mathrm{op}\ t_1]\!]\,\rho = [\![t_0]\!]\,\rho\ \underline{\mathrm{op}}_\perp\ [\![t_1]\!]\,\rho$$

where for any operator $\mathrm{op} \in \{+,-,\times\}$ in the syntax we have the corresponding function $\underline{\mathrm{op}} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ on the integers $\mathbb{Z}$ and also the binary function $\underline{\mathrm{op}}_\perp$ on $\mathbb{Z}_\perp$ defined as

$$\underline{\mathrm{op}}_\perp : (\mathbb{Z}_\perp \times \mathbb{Z}_\perp) \to \mathbb{Z}_\perp$$

$$x_1\ \underline{\mathrm{op}}_\perp\ x_2 = \begin{cases} \lfloor n_1\ \underline{op}\ n_2 \rfloor & \text{if } x_1 = \lfloor n_1 \rfloor \text{ and } x_2 = \lfloor n_2 \rfloor \text{ for some } n_1, n_2 \in \mathbb{Z} \\ \perp_{\mathbb{Z}_\perp} & \text{otherwise} \end{cases}$$

We remark that $\underline{\mathrm{op}}_\perp$ yields $\perp_{\mathbb{Z}_\perp}$ when at least one of the two arguments is $\perp_{\mathbb{Z}_\perp}$.

At the level of types, we have:

$$\underbrace{[\![(\underbrace{t_0}_{int}\ \mathrm{op}\ \underbrace{t_1}_{int})]\!]\rho}_{(V_{int})_\perp = \mathbb{Z}_\perp} = \underbrace{[\![\underbrace{t_0}_{int}]\!]\rho}_{(V_{int})_\perp}\ \underbrace{\underline{\mathrm{op}}_\perp}_{(\mathbb{Z}_\perp \times \mathbb{Z}_\perp) \to \mathbb{Z}_\perp}\ \underbrace{[\![\underbrace{t_1}_{int}]\!]\rho}_{(V_{int})_\perp}$$

9.2.4 Conditional

In order to define the semantics of the conditional expression, we exploit the conditional operator of the meta-language

$$Cond_\tau : \mathbb{Z}_\perp \times (V_\tau)_\perp \times (V_\tau)_\perp \to (V_\tau)_\perp$$

defined as:

$$Cond_\tau(v,d_0,d_1) \stackrel{\text{def}}{=} \begin{cases} d_0 & \text{if } v = \lfloor 0 \rfloor \\ d_1 & \text{if } \exists n \in \mathbb{Z}.\ v = \lfloor n \rfloor \wedge n \neq 0 \\ \bot_{(V_\tau)_\bot} & \text{if } v = \bot_{\mathbb{Z}_\bot} \end{cases}$$

Note that $Cond_\tau$ is parametric on the type τ. In the following, when τ can be inferred, we write just *Cond*. The conditional operator is *strict* on its first argument (i.e., it returns $\bot$ when the first argument is $\bot$) but not on the second and third arguments.

We can now define the denotational semantics of the conditional operator by letting:

$$[\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho \stackrel{\text{def}}{=} Cond\,([\![t]\!]\rho, [\![t_0]\!]\rho, [\![t_1]\!]\rho)$$

At the level of types we have:

$$\underbrace{[\![\textbf{if } \underbrace{t_0}_{int} \textbf{ then } \underbrace{t_1}_{\tau} \textbf{ else } \underbrace{t_2}_{\tau}]\!]\rho}_{(V_\tau)_\bot} = \underbrace{\underbrace{Cond_\tau}_{\mathbb{Z}_\bot\times(V_\tau)_\bot\times(V_\tau)_\bot\to(V_\tau)_\bot}\ \underbrace{([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho)}_{(V_{int})_\bot\times(V_\tau)_\bot\times(V_\tau)_\bot}}_{(V_\tau)_\bot}$$

9.2.5 Pairing

For the pairing operator we simply let:

$$[\![(t_0,t_1)]\!]\rho \stackrel{\text{def}}{=} \lfloor ([\![t_0]\!]\rho, [\![t_1]\!]\rho) \rfloor$$

Note that, for $t_0 : \tau_0$ and $t_1 : \tau_1$, the pair $([\![t_0]\!]\rho, [\![t_1]\!]\rho)$ is in $(V_{\tau_0})_\bot \times (V_{\tau_1})_\bot$ and not in $((V_{\tau_0})_\bot \times (V_{\tau_1})_\bot)_\bot$, thus we apply the lifting. In fact, at the level of type consistency we have:

$$\underbrace{[\![\underbrace{(\underbrace{t_0}_{\tau_0}, \underbrace{t_1}_{\tau_1})}_{\tau_0*\tau_1}]\!]\rho}_{(V_{\tau_0*\tau_1})_\bot} = \underbrace{\lfloor ([\![\underbrace{t_0}_{\tau_0}]\!]\rho, [\![\underbrace{t_1}_{\tau_1}]\!]\rho) \rfloor}_{((V_{\tau_0})_\bot\times(V_{\tau_1})_\bot)_\bot}$$

9.2.6 Projections

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

$$\begin{aligned}[\![\mathbf{fst}(t)]\!]\rho &\stackrel{\text{def}}{=} \mathbf{let}\ d \Leftarrow [\![t]\!]\rho.\ \pi_1\ d \\ &= \pi_1^*([\![t]\!]\rho) \\ [\![\mathbf{snd}(t)]\!]\rho &\stackrel{\text{def}}{=} \mathbf{let}\ d \Leftarrow [\![t]\!]\rho.\ \pi_2\ d \\ &= \pi_2^*([\![t]\!]\rho)\end{aligned}$$

The **let** operator (see Definition 8.10) allows to *de-lift* $[\![t]\!]\rho$ in order to apply projections π_1 and π_2. Instead, if $[\![t]\!]\rho = \bot$ the result is also $\bot$.

Again, we check that the type constraints are respected by the definition:

$$[\![\mathbf{fst}(\underbrace{\underbrace{t}_{\tau_0*\tau_1}}_{\tau_0})]\!]\rho = \mathbf{let}\ \underbrace{d}_{V_{\tau_0*\tau_1}} \Leftarrow \underbrace{[\![\underbrace{t}_{\tau_0*\tau_1}]\!]\rho}_{(V_{\tau_0*\tau_1})_\bot}.\ \underbrace{\underbrace{\pi_1}_{(V_{\tau_0})_\bot\times(V_{\tau_1})_\bot\to(V_{\tau_0})_\bot}\ \underbrace{d}_{V_{\tau_0*\tau_1}}}_{(V_{\tau_0})_\bot}$$

The case of $\mathbf{snd}(t : \tau_0 * \tau_1)$ is completely analogous and thus omitted.

9.2.7 Lambda Abstraction

For lambda-abstraction we use, of course, the lambda operator of the meta-language:

$$[\![\lambda x.\ t]\!]\rho \stackrel{\text{def}}{=} \left\lfloor \lambda d.\ [\![t]\!]\rho[^d/_x] \right\rfloor$$

where we bind x to d for evaluating t.

Note that, as in the case of pairing, we need to apply the lifting, because $\lambda d.\ [\![t]\!]\rho[^d/_x]$ is an element of $V_{\tau_0\to\tau_1} = [(V_{\tau_0})_\bot \to (V_{\tau_1})_\bot]$ and not of $(V_{\tau_0\to\tau_1})_\bot = [(V_{\tau_0})_\bot \to (V_{\tau_1})_\bot]_\bot$.

$$[\![\lambda\underbrace{\underbrace{x}_{\tau_0}.\ \underbrace{t}_{\tau_1}}_{\tau_0\to\tau_1}]\!]\rho = \lfloor\underbrace{\underbrace{\lambda\ \underbrace{d}_{(V_{\tau_0})_\bot}.\ \underbrace{[\![\underbrace{t}_{\tau_1}]\!]\rho[^d/_x]}_{(V_{\tau_1})_\bot}}_{[(V_{\tau_0})_\bot\to(V_{\tau_1})_\bot]}}_{[(V_{\tau_0})_\bot\to(V_{\tau_1})_\bot]_\bot}\rfloor$$

9.2.8 Function Application

Similarly to the case of projections, we apply the de-lifted version of the function to its argument:

$$\begin{aligned}[\![(t_1\ t_0)]\!]\rho &\stackrel{\text{def}}{=} \textbf{let}\ \varphi \Leftarrow [\![t_1]\!]\rho.\ \varphi([\![t_0]\!]\rho)\\ &= (\lambda\varphi.\ \varphi([\![t_0]\!]\rho))^*([\![t_1]\!]\rho)\end{aligned}$$

At the level of types, we have:

$$[\![(\underbrace{\underbrace{t_1}_{\tau_0\to\tau_1}\ \underbrace{t_0}_{\tau_0}}_{\tau_1})]\!]\rho = \underbrace{\textbf{let}\ \underbrace{\varphi}_{V_{\tau_0\to\tau_1}} \Leftarrow \underbrace{[\![t_1]\!]\rho}_{(V_{\tau_0\to\tau_1})_\perp}.\ \underbrace{\underbrace{\varphi}_{V_{\tau_0\to\tau_1}}\ \underbrace{([\![t_0]\!]\rho)}_{(V_{\tau_0})_\perp}}_{(V_{\tau_1})_\perp}}_{(V_{\tau_1})_\perp}$$

9.2.9 Recursion

For handling recursion we would like to find a solution (in the domain $(V_\tau)_\perp$, for $t : \tau$) to the recursive equation

$$[\![\textbf{rec}\ x.\ t]\!]\rho = [\![t]\!]\rho[{}^{[\![\textbf{rec}\ x.\ t]\!]\rho}/_x]$$

The least solution can be computed simply by applying the fix operator of the meta-language:

$$[\![\textbf{rec}\ x.\ t]\!]\rho \stackrel{\text{def}}{=} \text{fix}\ \lambda d.\ [\![t]\!]\rho[{}^{d}/_x]$$

Finally, we check that also this last definition is consistent with the typing:

$$[\![\textbf{rec}\ \underbrace{x}_{\tau}.\ \underbrace{t}_{\tau}]\!]\rho = \underbrace{\underbrace{\text{fix}}_{[(V_\tau)_\perp\to(V_\tau)_\perp]\to(V_\tau)_\perp}\ \underbrace{\lambda\ \underbrace{d}_{(V_\tau)_\perp}.\ \underbrace{[\![t]\!]\rho[{}^{d}/_x]}_{(V_\tau)_\perp}}_{[(V_\tau)_\perp\to(V_\tau)_\perp]}}_{(V_\tau)_\perp}$$

9.2.10 Eager semantics

The denotational semantics we have defined is *lazy*, in the sense that the evaluation of the argument is not enforced by the interpretation of application. The corresponding *eager* variant could be defined simply by letting:

$$[\![(t_1\ t_0)]\!]\rho \stackrel{\text{def}}{=} \textbf{let}\ \varphi \Leftarrow [\![t_1]\!]\rho.\ \textbf{let}\ d \Leftarrow [\![t_0]\!]\rho.\ \varphi(d)$$

The difference is that, according to the eager semantics, $[\![(t_1\ t_0)]\!]\rho$ evaluates to $\perp$ when $[\![t_0]\!]\rho$ evaluates to $\perp$, while this is not necessarily the case in the lazy semantics.

9.2.11 Examples

Example 9.1. Let us see some simple 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 not used.

1. $f \stackrel{\text{def}}{=} \lambda x : int.\ 3$
2. $g \stackrel{\text{def}}{=} \lambda x : int.\ \textbf{if}\ x\ \textbf{then}\ 3\ \textbf{else}\ 3$
3. $h \stackrel{\text{def}}{=} \textbf{rec}\, y : int \to int.\ \lambda x : int.\ 3$

Note that $f, g, h : int \to int$. For the term f we have:

$$[\![f]\!]\rho = [\![\lambda x.\ 3]\!]\rho = \lfloor \lambda d.\ [\![3]\!]\rho[^{d}/_{x}] \rfloor = \lfloor \lambda d.\ \lfloor 3 \rfloor \rfloor$$

When considering g, instead:

$$\begin{aligned} [\![g]\!]\rho &= [\![\lambda x.\ \textbf{if}\ x\ \textbf{then}\ 3\ \textbf{else}\ 3]\!]\rho \\ &= \lfloor \lambda d.\ [\![\textbf{if}\ x\ \textbf{then}\ 3\ \textbf{else}\ 3]\!]\rho[^{d}/_{x}] \rfloor \\ &= \lfloor \lambda d.\ \mathit{Cond}(d, \lfloor 3 \rfloor, \lfloor 3 \rfloor) \rfloor \\ &= \lfloor \lambda d.\ \textbf{let}\ x \Leftarrow d.\ \lfloor 3 \rfloor \rfloor \end{aligned}$$

where the last equality follows from the fact that both expressions $\mathit{Cond}(d, \lfloor 3 \rfloor, \lfloor 3 \rfloor)$ and $\textbf{let}\ x \Leftarrow d.\ \lfloor 3 \rfloor$ evaluate to $\bot_{\mathbb{Z}_\bot}$ when $d = \bot_{\mathbb{Z}_\bot}$ and to $\lfloor 3 \rfloor$ if d is a lifted value. Thus we can conclude that $[\![f]\!]\rho \neq [\![g]\!]\rho$.

Finally, for h we get:

$$\begin{aligned} [\![h]\!]\rho &= [\![\textbf{rec}\ y.\ \lambda x.\ 3]\!]\rho \\ &= \mathrm{fix}\ \lambda d_y.\ [\![\lambda x.\ 3]\!]\rho[^{d_y}/_{y}] \\ &= \mathrm{fix}\ \lambda d_y.\ \lfloor \lambda d_x.\ [\![3]\!]\rho[^{d_y}/_{y}, ^{d_x}/_{x}] \rfloor \\ &= \mathrm{fix}\ \lambda d_y.\ \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor \end{aligned}$$

Let $\Gamma_h = \lambda d_y.\ \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor$. We can compute the fixpoint by exploiting the fixpoint theorem to compute successive approximations:

$$\begin{aligned} d_0 &= \Gamma_h^0(\bot_{[\mathbb{Z}_\bot \to \mathbb{Z}_\bot]_\bot}) = \bot_{[\mathbb{Z}_\bot \to \mathbb{Z}_\bot]_\bot} \\ d_1 &= \Gamma_h(d_0) = (\lambda d_y.\ \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor)\bot = \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor \\ d_2 &= \Gamma_h(d_1) = (\lambda d_y.\ \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor)\lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor = \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor = d_1 \end{aligned}$$

Since $d_2 = d_1$ we have reached the fixpoint and thus

$$[\![h]\!]\rho = \lfloor \lambda d_x.\ \lfloor 3 \rfloor \rfloor = [\![f]\!]\rho.$$

Note that we could have avoided the calculation of d_2, because d_1 is already a maximal element in $[\mathbb{Z}_\bot \to \mathbb{Z}_\bot]_\bot$ and therefore it must be $\Gamma_h(d_1) = d_1$.

9.3 Continuity of Meta-language's Functions

In order to show that the semantics is always well defined we have to show that all the functions we employ in the definition are continuous, so that the fixpoint theory is applicable.

Theorem 9.1. *The following functions are monotone and continuous:*

1. $\underline{\mathrm{op}}_{\perp} : (\mathbb{Z}_{\perp} \times \mathbb{Z}_{\perp}) \to \mathbb{Z}_{\perp}$;
2. $Cond_{\tau} : \mathbb{Z}_{\perp} \times (V_{\tau})_{\perp} \times (V_{\tau})_{\perp} \to (V_{\tau})_{\perp}$;
3. $(\_,\_) : (V_{\tau_0})_{\perp} \times (V_{\tau_1})_{\perp} \to V_{\tau_0 * \tau_1}$;
4. $\pi_1 : V_{\tau_0 * \tau_1} \to (V_{\tau_0})_{\perp}$;
5. $\pi_2 : V_{\tau_0 * \tau_1} \to (V_{\tau_1})_{\perp}$;
6. **let**
7. apply
8. $\mathrm{fix} : [[(V_{\tau})_{\perp} \to (V_{\tau})_{\perp}] \to (V_{\tau})_{\perp}]$.

Proof. Monotonocity is obvious in most cases. We focus on the continuity of the various functions

1. Since $\underline{\mathrm{op}}_{\perp}$ is monotone over a domain with only finite chains then it is also continuous.
2. By using the Theorem 8.7, we can prove the continuity of *Cond* on each parameter separately.
   Let us show the continuity on the first parameter. Since chains in $\mathbb{Z}_{\perp}$ are finite, it is enough to prove monotonicity. We fix $d_1, d_2 \in (V_{\tau})_{\perp}$ and we prove the monotonicity of $\lambda x.\, Cond_{\tau}(x, d_1, d_2) : \mathbb{Z}_{\perp} \to (V_{\tau})_{\perp}$. Let $n, m \in \mathbb{Z}$.

   - the cases $\perp_{\mathbb{Z}_{\perp}} \sqsubseteq_{\mathbb{Z}_{\perp}} \perp_{\mathbb{Z}_{\perp}}$ or $\lfloor n \rfloor \sqsubseteq_{\mathbb{Z}_{\perp}} \lfloor n \rfloor$ are trivial;
   - for the case $\perp_{\mathbb{Z}_{\perp}} \sqsubseteq_{\mathbb{Z}_{\perp}} \lfloor n \rfloor$ then obviously

     $$Cond_{\tau}(\perp_{\mathbb{Z}_{\perp}}, d_1, d_2) = \perp_{(V_{\tau})_{\perp}} \sqsubseteq_{(V_{\tau})_{\perp}} Cond_{\tau}(\lfloor n \rfloor, d_1, d_2)$$

     because $\perp_{(V_{\tau})_{\perp}}$ is the bottom element of $(V_{\tau})_{\perp}$.
   - for the case $\lfloor n \rfloor \sqsubseteq_{\mathbb{Z}_{\perp}} \lfloor m \rfloor$, since $\mathbb{Z}_{\perp}$ is a flat domain we have $n = m$ and trivially $Cond_{\tau}(\lfloor n \rfloor, d_1, d_2) \sqsubseteq_{(V_{\tau})_{\perp}} Cond_{\tau}(\lfloor m \rfloor, d_1, d_2)$

   Now let us show the continuity on the second parameter, namely we fix $v \in \mathbb{Z}_{\perp}$ and $d \in (V_{\tau})_{\perp}$ and for any chain $\{d_i\}_{i \in \mathbb{N}}$ in $(V_{\tau})_{\perp}$ we prove that

   $$Cond_{\tau}\left(v, \bigsqcup_{i \in \mathbb{N}} d_i, d\right) = \bigsqcup_{i \in \mathbb{N}} Cond_{\tau}(v, d_i, d)$$

   - if $v = \perp_{\mathbb{Z}_{\perp}}$, then

     $$Cond_{\tau}\left(\perp_{\mathbb{Z}_{\perp}}, \bigsqcup_{i \in \mathbb{N}} d_i, d\right) = \perp_{\mathbb{Z}_{\perp}} = \bigsqcup_{i \in \mathbb{N}} \perp_{\mathbb{Z}_{\perp}} = \bigsqcup_{i \in \mathbb{N}} Cond_{\tau}(\perp_{\mathbb{Z}_{\perp}}, d_i, d)$$

- if $v = \lfloor 0 \rfloor$, then $\lambda x.\ Cond_\tau(\lfloor 0 \rfloor, x, d)$ is the identity function $\lambda x.\ x$ and we have

$$Cond_\tau\left(\lfloor 0 \rfloor, \bigsqcup_{i\in\mathbb{N}} d_i, d\right) = \bigsqcup_{i\in\mathbb{N}} d_i = \bigsqcup_{i\in\mathbb{N}} Cond_\tau(\lfloor 0 \rfloor, d_i, d)$$

- if $v = \lfloor n \rfloor$ with $n \neq 0$, then $\lambda x.\ Cond_\tau(\lfloor n \rfloor, x, d)$ is the constant function $\lambda x.\ d$ and we have

$$Cond_\tau\left(\lfloor n \rfloor, \bigsqcup_{i\in\mathbb{N}} d_i, d\right) = d = \bigsqcup_{i\in\mathbb{N}} d = \bigsqcup_{i\in\mathbb{N}} Cond_\tau(\lfloor n \rfloor, d_i, d)$$

In all cases $Cond_\tau$ is continuous.
Continuity on the third parameter is analogous.

3. For pairing $(\_,\_)$ we can use again the Theorem 8.7, which allows to show separately the continuity on each parameter. If we fix the first element we have

$$\left(d, \bigsqcup_{i\in\mathbb{N}} d_i\right) = \left(\bigsqcup_{i\in\mathbb{N}} d, \bigsqcup_{i\in\mathbb{N}} d_i\right) = \bigsqcup_{i\in\mathbb{N}} (d, d_i)$$

by definition of lub of a chain of pairs (see Theorem 8.1). The same holds for the second parameter.
4. Projections π_1 and π_2 are continuous by Theorem 8.2.
5. The **let** function is continuous since $(\cdot)^*$ is continuous by Theorem 8.4.
6. apply is continuous by Theorem 8.8
7. fix is continuous by Theorem 8.10. □

In the previous theorem we have not mentioned the continuity proofs for lambda abstraction and recursion. The next theorem fills these gaps.

Theorem 9.2. *Let $t : \tau$ be a well typed term of HOFL; then the following holds:*

1. *$(\lambda d.\ [\![t]\!]\rho[^d/_x])$ is a continuous function.*
2. *if $\tau = \tau_0 \to \tau_1$ is a functional type, then* fix *$\lambda d.\ [\![t]\!]\rho[^d/_x]$ is a continuous function.*

Proof. Let us prove the two properties:

1. We prove the stronger property that, for any $n \in \mathbb{N}$:

$$\lambda(d_1, ..., d_n).\ [\![t]\!]\rho[^{d_1}/_{x_1}, \cdots, ^{d_n}/_{x_n}]$$

is a continuous function. The proof is by structural induction on t. Below, for brevity, we write $\widetilde{d}$ instead of $d_1, ..., d_n$ and ρ' instead of $\rho[^{d_1}/_{x_1}, \cdots, ^{d_n}/_{x_n}]$:

$t = y$: Then $\lambda\widetilde{d}.\ [\![y]\!]\rho'$ is either a projection function (if $y = x_i$ for some $i \in [1, n]$) or the constant function $\lambda\widetilde{d}.\ \rho(y)$ (if $y \notin \{x_1, ..., x_n\}$), which are continuous.

$t = t_1 \text{ op } t_2$: By inductive hypothesis $f_1 \stackrel{\text{def}}{=} \lambda\widetilde{d}.\, [\![t_1]\!]\rho'$ and $f_2 \stackrel{\text{def}}{=} \lambda\widetilde{d}.\, [\![t_2]\!]\rho'$ are continuous. Then $f \stackrel{\text{def}}{=} \lambda\widetilde{d}.\, ((f_1\ \widetilde{d}),(f_2\ \widetilde{d}))$ is continuous, and

$$\begin{aligned}\lambda\widetilde{d}.\, [\![t_1 \text{ op } t_2]\!]\rho' &= \lambda\widetilde{d}.\, ([\![t_1]\!]\rho' \ \underline{\text{op}}_\perp\ [\![t_2]\!]\rho') \\ &= \lambda\widetilde{d}.\, (f_1\ \widetilde{d})\ \underline{\text{op}}_\perp\ (f_2\ \widetilde{d}) \\ &= \underline{\text{op}}_\perp \circ f\end{aligned}$$

is continuous because $\underline{\text{op}}_\perp$ is continuous and the composition of continuous functions yields a continuous function by Theorem 8.5.

$t = \lambda y.\, t'$: By inductive hypothesis we can assume that $\lambda(\widetilde{d},d).[\![t']\!]\rho'[^d/_y]$ is continuous. Then $\text{curry}(\lambda(\widetilde{d},d).[\![t']\!]\rho'[^d/_y])$ is continuous since curry is continuous, and we conclude by noting that

$$\begin{aligned}\text{curry}(\lambda(\widetilde{d},d).[\![t']\!]\rho'[^d/_y]) &= \lambda\widetilde{d}.\, \lambda d.\, [\![t']\!]\rho'[^d/_y] \\ &= \lambda\widetilde{d}.[\![\lambda y.\, t']\!]\rho'.\end{aligned}$$

We leave the remaining cases as an exercise.

2. To prove the second proposition we note that

$$\text{fix}\ \ \lambda d.[\![t]\!]\rho[^d/_x]$$

is the application of a continuous function (i.e., the function fix, by Theorem 8.10) to a continuous argument (i.e., $\lambda d.[\![t]\!]\rho[^d/_x]$, continuous by the first part of this theorem) so it is continuous by Theorem 8.8. □

We conclude this section by recalling that the definition of denotational semantics is consistent with the typing.

Theorem 9.3 (Type Consistency). *If $t : \tau$ then $\forall \rho \in Env.\ [\![t]\!]\,\rho \in (V_\tau)_\perp$.*

Proof. The proof is by structural induction on t and it has been outlined when giving the structurally recursive definition of the denotational semantics (where we have also relied on the previous continuity theorems). □

9.4 Substitution Lemma and Other Properties

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 9.4 (Substitution Lemma). *Let $x,t : \tau$ and $t' : \tau'$. We have*

$$[\![t'[^t/_x]]\!]\,\rho = [\![t']\!]\,\rho[^{[\![t]\!]\rho}/_x]$$

Proof. By Theorem 7.1 we know that $t'[^{t}/_{x}] : \tau'$. The proof is by structural induction on t' and left as an exercise (see Problem 9.13). □

In words, replacing a variable x with a term t in a term t' returns a term $t'[^{t}/_{x}]$ whose denotational semantics $[\![t'[^{t}/_{x}]]\!]\rho = [\![t']\!]\rho[^{[\![t]\!]\rho}/_{x}]$ depends only on the denotational semantics $[\![t]\!]\rho$ of t.

Remark 9.1 (Compositionality). The substitution lemma is an important result, as it implies the compositionality of denotational semantics, namely for all terms t_1, t_2 and environment ρ we have:

$$[\![t_1]\!]\rho = [\![t_2]\!]\rho \quad \Rightarrow \quad [\![t[^{t_1}/_{x}]]\!]\rho = [\![t[^{t_2}/_{x}]]\!]\rho$$

Theorem 9.5. *Let t be a well-defined term of HOFL. Let $\rho, \rho' \in Env$ such that $\forall\, x \in \mathrm{fv}(t).\ \rho(x) = \rho'(x)$ then:*

$$[\![t]\!]\rho = [\![t]\!]\rho'$$

Proof. The proof is by structural induction on t and left as an exercise (see Problem 9.16). □

Theorem 9.6. *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. □

Problems

9.1. Consider the HOFL term:

$$t \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ (f(x) \times f(x))$$

Derive the type, the canonical form and the denotational semantics of t.

9.2. Consider the HOFL term:

$$t \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ \lambda y.\ \mathbf{if}\ x \times y\ \mathbf{then}\ x\ \mathbf{else}\ (fx)((fx)y)$$

Derive the type, the canonical form and the denotational semantics of t.

9.3. Consider the HOFL term:

$$t \stackrel{\text{def}}{=} \mathbf{fst}(\ (\lambda x.\ x)\ (\ 1\ ,\ ((\mathbf{rec}\, f.\ \lambda y.\ (f\ y))\ 2)\ )\ ).$$

Derive the type, the canonical form and the denotational semantics of t.

9.4. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ (g\ (f\ (x-1)))$$

1. Derive the type of t and the denotational semantics of $[\![t]\!]\rho$ by assuming that $\rho g = \lfloor h \rfloor$ for some suitable h.
2. Compute the canonical form of the term $(((\lambda g.\ t)\ \lambda x.\ x)\ 1)$. Would it be possible to compute the canonical form of t?

9.5. Let us consider the following recursive definition:

$$f(x) \stackrel{\text{def}}{=} \textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ 2 \times f(x-1).$$

1. Define a well-formed, closed HOFL term t that corresponds to the above definition and determine its type.
2. Compute its denotational semantics $[\![t]\!]\rho$ and prove that

$$n \geq 0 \quad \Rightarrow \quad \textbf{let}\ \varphi \Leftarrow [\![t]\!]\rho.\ \varphi\lfloor n \rfloor = \lfloor 2^n \rfloor.$$

Hint: Prove that the n-th fixpoint approximation is

$$d_n\ =\ \lfloor\ \lambda d.\ Cond(\lfloor 0 \rfloor \underline{\leq}_{\perp} d \underline{\leq}_{\perp} \lfloor n \rfloor\ ,\ \lfloor 2^d \rfloor\ ,\ \perp)\ \rfloor.$$

9.6. Let us consider the following recursive definition:

$$f(x) \stackrel{\text{def}}{=} \textbf{if}\ x = 0\ \textbf{then}\ 0\ \textbf{else}\ f(f(x-1))$$

1. Define a well-formed, closed HOFL term t that corresponds to the above definition and determine its type, its canonical form and its denotational semantics.
2. Define the set of fixpoints that satisfy the recursive definition.

9.7. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 0\ \textbf{else}\ f\ (x-x)$$

1. Determine the type of t and its denotational semantics $[\![t]\!]\rho = \text{fix}\ \Gamma$.
2. Is fix Γ the unique fixpoint of Γ?
   Hint: Consider the elements greater than fix Γ in the order and check if they are fixpoints for Γ.

9.8. Consider the Fibonacci sequence already found in Problem 4.14 and the corresponding term t from Problem 7.8:

$$F(0) \stackrel{\text{def}}{=} 1 \qquad F(1) \stackrel{\text{def}}{=} 1 \qquad F(n+2) \stackrel{\text{def}}{=} F(n+1) + F(n)s.$$

where $n \in \mathbb{N}$.

1. Compute the suitable transformation Γ such that $[\![t]\!]\rho = \text{fix}\ \Gamma$.

2. Prove that the denotational semantics $[\![t]\!]\rho$ satisfies the above equations, to conclude that the given implementation of Fibonacci numbers is correct.
Hint: Compute $[\![(t\ 0)]\!]\rho$, $[\![(t\ 1)]\!]\rho$ and $[\![(t\ n+2)]\!]\rho$ exploiting the equality $[\![t]\!] = \Gamma\,[\![t]\!]$.

9.9. Assuming that t_1 has type τ_1, let us consider the term $t_2 \stackrel{\text{def}}{=} \lambda x.\ (t_1\ x)$.

1. Do both terms have the same type?
2. Do both terms have the same lazy denotational semantics?

9.10. Let us consider the terms

$$t_1 \stackrel{\text{def}}{=} \lambda x.\ \mathbf{rec}\ y.\ y+1$$
$$t_2 \stackrel{\text{def}}{=} \mathbf{rec}\ y.\ \lambda x.\ (y\ x)+2$$

1. Do both terms have the same type?
2. Do both terms have the same lazy denotational semantics?

9.11. Given a monotone function $f : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$, prove that $f\perp_{\mathbb{Z}_\perp} = f(f\perp_{\mathbb{Z}_\perp})$. Then, let $t : \mathit{int} \to \mathit{int}$ be a closed term of HOFL and consider the term

$$t_1 \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ (t\ (f\ x))$$

1. Determine the most general type of t_1.
2. Exploit the above result to prove that $[\![t_1]\!]\rho = [\![t_2]\!]\rho$, where

$$t_2 \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ (t\ \mathbf{rec}\ y.\ y)$$

9.12. Let us extend the syntax of (lazy) HOFL by adding the construct for sequential composition $t_1;t_2$ that, informally, represents the function obtained by applying the function t_1 to the argument and then the function t_2 to the result. Define, for the new construct ;:

1. the typing rule;
2. the (big-step) operational semantics;
3. the denotational semantics.

Then prove that for every closed term t, both terms $(t_1;t_2\ t)$ and $(t_2\ (t_1\ t))$ have the same type and are equivalent according to the denotational semantics.

9.13. Complete the proof of the Substitution Lemma (Theorem 9.4).

9.14. Let t_1, t_2 be well-formed HOFL terms and ρ an environment.

1. Prove that

$$[\![t_1]\!]\rho\ =\ [\![t_2]\!]\rho \quad \Rightarrow \quad [\![(t_1\ x)]\!]\rho\ =\ [\![(t_2\ x)]\!]\rho \tag{9.1}$$

2. Prove that the reversed implication is generally not valid by giving a counterexample. Then, find the conditions under which also the reversed implication holds.
3. Exploit the Substitution Lemma (Theorem 9.4) to prove that for all t and $x \notin \mathrm{fv}(t_1) \cup \mathrm{fv}(t_2)$:

$$[\![t_1]\!]\rho \;=\; [\![t_2]\!]\rho \quad \Rightarrow \quad [\![t[t_1/x]]\!]\rho \;=\; [\![t[t_2/x]]\!]\rho \tag{9.2}$$

4. Observe that the implication 9.1 is just a special case of the latter equality 9.2 and explain how.

9.15. Is it possible to modify the denotational semantics of HOFL assigning to the construct

$$\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1$$

- the semantics of t_1 if the semantics of t is $\bot_{N_\bot}$, and
- the semantics of t_0 otherwise? (If not, why?)

9.16. Complete the proof of Theorem 9.5.

Chapter 10
Equivalence between HOFL denotational and operational semantics

Honest disagreement is often a good sign of progress. (Mahatma Gandhi)

Abstract In this chapter we address the correspondence between the operational semantics of HOFL from Chapter 7 and its denotational semantics from Chapter 9. The situation is not as straightforward as for IMP. A first discrepancy between the two semantics is that the operational one evaluates only closed (and typable) terms, while the denotational one can handle terms with variables, thanks to environments. Apart from this minor issue, the key fact is that the canonical forms arising from the operational semantics for the terms of type τ are more concrete than the mathematical elements of the corresponding domain $(V_\tau)_\perp$. Thus, it is inevitable that terms with different canonical forms can be assigned the same denotation. Vice versa, we show that terms with the same canonical form are always assigned the same denotation. Only for terms of type *int* we have a full agreement between the two semantics. On the positive side, a term converges operationally if and only if it converges denotationally. We conclude the chapter by discussing the equivalences over terms induced by the two semantics and by presenting an alternative denotational semantics, called *unlifted*, which is simpler but less expressive than the one studied in Chapter 9.

10.1 HOFL: Operational Semantics vs Denotational Semantics

As we have done for IMP, now we address the relation between the denotational and operational semantics of HOFL. One might expect to prove a complete equivalence, as in the case of IMP:

$$\forall t, c. \quad t \to c \quad \stackrel{?}{\Leftrightarrow} \quad \forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho$$

But, as we are going to show, the situation in the case of HOFL is more complex and the implication is valid in one direction only, i.e., the operational semantics is correct but not complete:

$$t \to c \Rightarrow \forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho \quad \text{but} \quad (\forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho) \not\Rightarrow t \to c$$

Let us consider a very simple example that shows the difference between the denotational and the operational semantics.

Example 10.1. Let $c_0 = \lambda x.\ x+0$ and $c_1 = \lambda x.\ x$ be two HOFL terms, where $x : \mathit{int}$. Clearly:

$$[\![c_0]\!]\rho = [\![c_1]\!]\rho \qquad \text{but} \qquad c_0 \not\to c_1$$

In fact, from the denotational semantics we get:

$$[\![c_0]\!]\rho = [\![\lambda x.\ x+0]\!]\rho = \lfloor \lambda d.\ d \underline{+}_\perp \lfloor 0 \rfloor \rfloor = \lfloor \lambda d.\ d \rfloor = [\![\lambda x.\ x]\!]\rho = [\![c_1]\!]\rho$$

but for the operational semantics we have that both $\lambda x.\ x$ and $\lambda x.\ x+0$ are already in canonical form and $c_0 \neq c_1$.

The counterexample shows that, at least for the functional type $\mathit{int} \to \mathit{int}$, there are different canonical forms with the same denotational semantics, namely terms which compute the same function in $[\mathbb{Z}_\perp \to \mathbb{Z}_\perp]_\perp$. 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 is not recursively enumerable, while the set of theorems of every (finite) inference system is recursively enumerable.

Even if we cannot have a strong correspondence result between the operational and denotational semantics as it was the case for IMP, we can establish a full agreement between the two semantics w.r.t. the notion of termination. In particular, by letting the predicate $t\downarrow$ denote the fact that the term t can be reduced to some canonical form (called *operational convergence*) and $t \Downarrow$ denote the fact that the term $t : \tau$ is assigned a denotation other than $\perp_{(V_\tau)_\perp}$ (called *denotational convergence*), we have the perfect match:

$$t\downarrow \quad \Leftrightarrow \quad t \Downarrow$$

10.2 Correctness

We are ready to show the correctness of the operational semantics of HOFL w.r.t. the denotational one. Note that since the operational semantics is defined for closed terms only, the environment is inessential in the following theorem.

Theorem 10.1 (Correctness). *Let $t : \tau$ be a HOFL closed term and let $c : \tau$ be a canonical form. Then we have:*

$$t \to c \quad \Rightarrow \quad \forall \rho \in \mathit{Env}.\ [\![t]\!]\rho = [\![c]\!]\rho$$

Proof. We proceed by rule induction. So we prove

$$P(t \to c) \stackrel{\text{def}}{=} \forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho$$

for the conclusion $t \to c$ of each rule, when the predicate holds for the premises.

C_τ: The rule for terms in canonical forms (integers, pairs, abstraction) is

$$\frac{}{c \to c}$$

We have to prove $P(c \to c) \stackrel{\text{def}}{=} \forall \rho.\ [\![c]\!]\rho = [\![c]\!]\rho$, which is obviously true.

Arit.: Let us consider the rules for arithmetic operators $\text{op} \in \{+, -, \times\}$:

$$\frac{t_1 \to n_1 \quad t_2 \to n_2}{t_1 \text{ op } t_2 \to n_1 \underline{\text{op}}\, n_2}$$

We assume the inductive hypotheses:

$$P(t_1 \to n_1) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_1]\!]\rho = [\![n_1]\!]\rho = \lfloor n_1 \rfloor$$
$$P(t_2 \to n_2) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_2]\!]\rho = [\![n_2]\!]\rho = \lfloor n_2 \rfloor$$

and we want to prove

$$P(t_1 \text{ op } t_2 \to n_1 \underline{\text{op}}\, n_2) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_1 \text{ op } t_2]\!]\rho = \left[\!\!\left[n_1 \underline{\text{op}}\, n_2 \right]\!\!\right] \rho.$$

We have:

$$\begin{aligned}
[\![t_1 \text{ op } t_2]\!]\rho &= [\![t_1]\!]\rho \ \underline{\text{op}}_\perp\ [\![t_2]\!]\rho && \text{(by definition of } [\![\cdot]\!]\text{)} \\
&= \lfloor n_1 \rfloor \ \underline{\text{op}}_\perp\ \lfloor n_2 \rfloor && \text{(by inductive hypotheses)} \\
&= \lfloor n_1 \ \underline{\text{op}}\ n_2 \rfloor && \text{(by definition of } \underline{\text{op}}_\perp\text{)} \\
&= \left[\!\!\left[n_1 \ \underline{\text{op}}\ n_2 \right]\!\!\right] \rho && \text{(by definition of } [\![\cdot]\!]\text{).}
\end{aligned}$$

Cond.: In the case of the conditional construct we have two rules to consider. For

$$\frac{t \to 0 \quad t_0 \to c_0}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_0}$$

we can assume

$$P(t \to 0) \stackrel{\text{def}}{=} \forall \rho.\ [\![t]\!]\rho = [\![0]\!]\rho = \lfloor 0 \rfloor$$
$$P(t_0 \to c_0) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_0]\!]\rho = [\![c_0]\!]\rho$$

and we want to prove:

$$P(\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_0) \stackrel{\text{def}}{=} \forall \rho.\ [\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho = [\![c_0]\!]\rho$$

We have:

$$\begin{aligned}
[\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho &= \mathit{Cond}([\![t]\!]\rho, [\![t_0]\!]\rho, [\![t_1]\!]\rho) && \text{(by def. of } [\![\cdot]\!]\text{)}\\
&= \mathit{Cond}(\lfloor 0 \rfloor, [\![t_0]\!]\rho, [\![t_1]\!]\rho) && \text{(by ind. hyp.)}\\
&= [\![t_0]\!]\rho && \text{(by def. of } \mathit{Cond}\text{)}\\
&= [\![c_0]\!]\rho && \text{(by ind. hyp.).}
\end{aligned}$$

The same procedure holds for the second rule of the conditional operator.

Proj.: Let us consider the rule for the first projection:

$$\frac{t \to (t_0, t_1) \quad t_0 \to c_0}{\textbf{fst}(t) \to c_0}$$

We can assume

$$\begin{aligned}
P(t \to (t_0, t_1)) &\stackrel{\text{def}}{=} \forall \rho.\ [\![t]\!]\rho = [\![(t_0, t_1)]\!]\rho\\
P(t_0 \to c_0) &\stackrel{\text{def}}{=} \forall \rho.\ [\![t_0]\!]\rho = [\![c_0]\!]\rho
\end{aligned}$$

and we want to prove

$$P(\textbf{fst}(t) \to c_0) \stackrel{\text{def}}{=} \forall \rho.\ [\![\textbf{fst}(t)]\!]\rho = [\![c_0]\!]\rho.$$

We have:

$$\begin{aligned}
[\![\textbf{fst}(t)]\!]\rho &= \pi_1^*([\![t]\!]\rho) && \text{(by def. of } [\![\cdot]\!]\text{)}\\
&= \pi_1^*([\![(t_0, t_1)]\!]\rho) && \text{(by ind. hyp.)}\\
&= \pi_1^*(\lfloor([\![t_0]\!]\rho, [\![t_1]\!]\rho)\rfloor) && \text{(by def. of } [\![\cdot]\!]\text{)}\\
&= \pi_1([\![t_0]\!]\rho, [\![t_1]\!]\rho) && \text{(by def. of lifting)}\\
&= [\![t_0]\!]\rho && \text{(by def. of } \pi_1\text{)}\\
&= [\![c_0]\!]\rho && \text{(by ind. hyp.).}
\end{aligned}$$

The same procedure holds for the **snd** operator.

App.: The rule for application is:

$$\frac{t_1 \to \lambda x.\, t_1' \quad t_1'[^{t_0}/_x] \to c}{(t_1\ t_0) \to c}$$

We can assume:

$$\begin{aligned}
P(t_1 \to \lambda x.\, t_1') &\stackrel{\text{def}}{=} \forall \rho.\ [\![t_1]\!]\rho = [\![\lambda x.\, t_1']\!]\rho\\
P(t_1'[^{t_0}/_x] \to c) &\stackrel{\text{def}}{=} \forall \rho.\ [\![t_1'[^{t_0}/_x]]\!]\rho = [\![c]\!]\rho
\end{aligned}$$

and we want to prove

$$P((t_1\ t_0) \to c) \stackrel{\text{def}}{=} \forall \rho.\ [\![(t_1\ t_0)]\!]\rho = [\![c]\!]\rho.$$

We have:

$$\begin{aligned}
[\![(t_1\ t_0)]\!]\rho &= \textbf{let}\ \varphi \Leftarrow [\![t_1]\!]\rho.\ \varphi([\![t_0]\!]\rho) && \text{(by definition of } [\![\cdot]\!]) \\
&= \textbf{let}\ \varphi \Leftarrow [\![\lambda x.\ t_1']\!]\rho.\ \varphi([\![t_0]\!]\rho) && \text{(by ind. hypothesis)} \\
&= \textbf{let}\ \varphi \Leftarrow \left\lfloor \lambda d.\ [\![t_1']\!]\rho[^d/_x] \right\rfloor.\ \varphi([\![t_0]\!]\rho) && \text{(by definition of } [\![\cdot]\!]) \\
&= (\lambda d.\ [\![t_1']\!]\rho[^d/_x])\ ([\![t_0]\!]\rho) && \text{(by de-lifting)} \\
&= [\![t_1']\!]\rho[^{[\![t_0]\!]\rho}/_x] && \text{(by application)} \\
&= [\![t_1'[^{t_0}/_x]]\!]\rho && \text{(by Subst. Lemma)} \\
&= [\![c]\!]\rho && \text{(by ind. hypothesis).}
\end{aligned}$$

Rec.: Finally, we consider the rule for recursion:

$$\frac{t[^{\textbf{rec}\ x.\ t}/_x] \to c}{\textbf{rec}\ x.\ t \to c}$$

We can assume:

$$P(t[^{\textbf{rec}\ x.\ t}/_x] \to c) \stackrel{\text{def}}{=} \forall\rho.\ [\![t[^{\textbf{rec}\ x.\ t}/_x]]\!]\rho = [\![c]\!]\rho$$

and we want to prove

$$P(\textbf{rec}\ x.\ t \to c) \stackrel{\text{def}}{=} \forall\rho.\ [\![\textbf{rec}\ x.\ t]\!]\rho = [\![c]\!]\rho.$$

We have:

$$\begin{aligned}
[\![\textbf{rec}\ x.\ t]\!]\rho &= [\![t]\!]\rho[^{[\![\textbf{rec}\ x.\ t]\!]\rho}/_x] && \text{(by definition)} \\
&= [\![t[^{\textbf{rec}\ x.\ t}/_x]]\!]\rho && \text{(by the Substitution Lemma)} \\
&= [\![c]\!]\rho && \text{(by inductive hypothesis)}
\end{aligned}$$

Since there are no more rules to consider, we conclude the thesis holds. □

10.3 Agreement on Convergence

Now we define the concept of convergence (i.e., termination) for the operational and the denotational semantics.

Definition 10.1 (Operational convergence). Let $t : \tau$ be a closed term of HOFL, we define the following predicate:

$$t\downarrow \quad \Leftrightarrow \quad \exists c \in C_\tau.\ t \longrightarrow c.$$

If $t\downarrow$, then we say that t *converges operationally*. We say that t *diverges*, written $t\uparrow$, if t does not converge operationally.

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 $\bot$.

Definition 10.2 (Denotational convergence). Let t be a closed term of HOFL with type τ, we define the following predicate:

$$t \Downarrow \quad \Leftrightarrow \quad \forall \rho \in Env, \exists v \in V_\tau.\ [\![t]\!]\rho = \lfloor v \rfloor.$$

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. The implication $t \downarrow \Rightarrow\ t \Downarrow$ can be readily proved.

Theorem 10.2. *Let $t : \tau$ be a closed typable term of HOFL. Then we have:*

$$t \downarrow \quad \Rightarrow \quad t \Downarrow$$

Proof. If $t \to c$, then $\forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho$ by Theorem 10.1. But $[\![c]\!]\rho$ is a lifted value, (see Theorem 9.6) and thus it is different than $\bot_{(V_\tau)_\bot}$. □

Also the opposite implication $t \Downarrow \Rightarrow\ t \downarrow$ holds (for any closed and typable term t, see Theorem 10.3) but the proof is not straightforward: We cannot simply rely on structural induction; instead it is necessary to introduce a particular logical relation between elements of the interpretation domains and HOFL terms. We will only sketch the proof, but first we show that the standard structural induction does not help in proving the agreement of semantics about convergence.

Remark 10.1 (On the reason why structural induction fails for proving $t \Downarrow \Rightarrow\ t \downarrow$). The property $P(t) \stackrel{\text{def}}{=} t \Downarrow \Rightarrow\ t \downarrow$ cannot be proved by structural induction on t. Here we give some insights on the reason why it is so. Let us focus on the case of function application $(t_1\ t_0)$. By structural induction, we assume

$$P(t_1) \stackrel{\text{def}}{=} t_1 \Downarrow \Rightarrow\ t_1 \downarrow \qquad \text{and} \qquad P(t_0) \stackrel{\text{def}}{=} t_0 \Downarrow \Rightarrow\ t_0 \downarrow$$

and we want to prove $P(t_1\ t_0) \stackrel{\text{def}}{=} (t_1\ t_0) \Downarrow \Rightarrow\ (t_1\ t_0) \downarrow$.

Let us assume the premise $(t_1\ t_0) \Downarrow$ (i.e., $[\![(t_1\ t_0)]\!]\rho \neq \bot$) of the implication. We would like to prove that $(t_1\ t_0) \downarrow$, i.e., that $\exists c.\ (t_1\ t_0) \to c$. By definition of denotational semantics we have $t_1 \Downarrow$. In fact

$$[\![(t_1\ t_0)]\!]\rho \stackrel{\text{def}}{=} \textbf{let}\ \varphi \Leftarrow [\![t_1]\!]\rho.\ \varphi([\![t_0]\!]\rho)$$

and therefore $[\![(t_1\ t_0)]\!]\rho \neq \bot$ requires $[\![t_1]\!]\rho \neq \bot$. By the first inductive hypothesis we then have $t_1 \downarrow$ and by definition of the operational semantics it must be the case that $t_1 \to \lambda x.\ t_1'$ for some x and t_1'. By correctness (Theorem 10.1), we then have

$$[\![t_1]\!]\rho = [\![\lambda x.\ t_1']\!]\rho = \left\lfloor \lambda d.\ [\![t_1']\!]\rho[^d/_x] \right\rfloor.$$

Therefore:

$$\begin{aligned}
[\![(t_1\ t_0)]\!]\rho &= \textbf{let}\ \varphi \Leftarrow \lfloor \lambda d.\ [\![t_1']\!]\rho[^d/_x] \rfloor .\ \varphi([\![t_0]\!]\rho) && \text{(see above)}\\
&= (\lambda d.\ [\![t_1']\!]\rho[^d/_x])\ ([\![t_0]\!]\rho) && \text{(by de-lifting)}\\
&= [\![t_1']\!]\rho[^{[\![t_0]\!]\rho}/_x] && \text{(by functional application)}\\
&= [\![t_1'[^{t_0}/_x]]\!]\rho && \text{(by the Substitution Lemma)}
\end{aligned}$$

So $(t_1\ t_0) \Downarrow$ if and only if $t_1'[^{t_0}/_x] \Downarrow$. We would like to conclude by structural induction that $t_1'[^{t_0}/_x] \downarrow$ and then prove the theorem by using the rule:

$$\frac{t_1 \to \lambda x.\, t_1' \quad t_1'[^{t_0}/_x] \to c}{(t_1\ t_0) \to c}$$

but this is incorrect since $t_1'[^{t_0}/_x]$ is not a sub-term of $(t_1\ t_0)$ and we are not allowed to assume that $P(t_1'[^{t_0}/_x])$ holds.

Theorem 10.3. *For any closed typable term $t : \tau$ we have:*

$$t \Downarrow \quad \Rightarrow \quad t \downarrow$$

Proof. The proof exploits two suitable *logical relations*, indexed by HOFL types:

- $\lesssim^c_\tau \subseteq V_\tau \times C_\tau$ that relates canonical forms to corresponding values in V_τ and that is defined by structural induction over types τ;
- $\lesssim_\tau \subseteq (V_\tau)_\perp \times T_\tau$ that relates well-formed (closed) terms to values in $(V_\tau)_\perp$ and that is defined by letting:

$$d \lesssim_\tau t \quad \stackrel{\text{def}}{=} \quad \forall v \in V_\tau.\ d = \lfloor v \rfloor \Rightarrow \exists c.\ t \to c \wedge v \lesssim^c_\tau c$$

  In particular, note that, by definition, we have $\perp_{(V_\tau)_\perp} \lesssim_\tau t$ for any term $t : \tau$.

The logical relation on canonical forms is defined as follows

ground type: we simply let $n \lesssim^c_{int} n$;
product type: we let $(d_0, d_1) \lesssim^c_{\tau_0 * \tau_1} (t_0, t_1)$ iff $d_0 \lesssim_{\tau_0} t_0$ and $d_1 \lesssim_{\tau_1} t_1$;
function type: we let $\varphi \lesssim^c_{\tau_0 \to \tau_1} \lambda x.\, t$ iff $\forall d_0 \in (V_{\tau_0})_\perp$ and $\forall t_0 : \tau_0$ closed, $d_0 \lesssim_{\tau_0} t_0$ implies $\varphi(d_0) \lesssim_{\tau_1} t[^{t_0}/_x]$.

Then one can show, by structural induction on $t : \tau$ that:

1. $\forall d, d' \in (V_\tau)_\perp.\ (d \sqsubseteq_{(V_\tau)_\perp} d' \wedge d' \lesssim_\tau t) \Rightarrow d \lesssim_\tau t$;
2. if $\{d_i\}_{i \in \mathbb{N}}$ is a chain in $(V_\tau)_\perp$ such that $\forall i \in \mathbb{N}.\ d_i \lesssim_\tau t$, then $\bigsqcup_{i \in \mathbb{N}} d_i \lesssim_\tau t$ (i.e., the predicate $\cdot \lesssim_\tau t$ is inclusive).

Finally, by structural induction on terms, one can prove that $\forall t : \tau$ with $\mathrm{fv}(t) \subseteq \{x_1 : \tau_1, \ldots, x_k : \tau_k\}$, if $\forall i \in [1,k].\ d_i \lesssim_{\tau_i} t_i$ then $[\![t]\!]\rho[^{d_1}/_{x_1}, \ldots, ^{d_k}/_{x_k}] \lesssim_\tau t[^{t_1}/_{x_1}, \ldots, ^{t_k}/_{x_k}]$. In fact, taking $t : \tau$ closed, it follows from the definition of $\lesssim_\tau$ that if $t \Downarrow$, i.e., $[\![t]\!]\rho = \lfloor v \rfloor$ for some $v \in V_\tau$, then $t \to c$ for some canonical form c, i.e., $t \downarrow$. □

10.4 Operational and Denotational Equivalences of Terms

In this chapter introduction we have shown that the denotational semantics is more abstract than the operational. In order to study the relationship between the operational and denotational semantics of HOFL we now introduce two equivalence relations between terms. Operationally two closed terms are equivalent if they both diverge or have the same canonical form.

Definition 10.3 (Operational equivalence). Let t_0 and t_1 be two well-typed terms of HOFL then we define a binary relation $\equiv_{op}$ as follows:

$$t_0 \equiv_{op} t_1 \quad \Leftrightarrow \quad (t_0 \uparrow \wedge\, t_1 \uparrow) \vee (\exists c.\, t_0 \to c \wedge t_1 \to c)$$

And we say that t_0 is operationally equivalent to t_1 if $t_0 \equiv_{op} t_1$.

We have also the denotational counterpart of the definition of equivalence.

Definition 10.4 (Denotational equivalence). Let t_0 and t_1 be two well-typed terms of HOFL then we define a binary relation $\equiv_{den}$ as follows:

$$t_0 \equiv_{den} t_1 \quad \Leftrightarrow \quad \forall \rho.\, [\![t_0]\!]\,\rho = [\![t_1]\!]\,\rho$$

And we say that t_0 is denotationally equivalent to t_1 if $t_0 \equiv_{den} t_1$.

Remark 10.2. Note that the definition of denotational equivalence terms applies also to non closed terms. Operational equivalence of non closed terms t and t' could also be defined by taking the closure of the equivalence w.r.t. the embedding of t and t' in any context $C[\cdot]$ such that $C[t]$ and $C[t']$ are also closed, i.e., by requiring that $C[t]$ and $C[t']$ are operationally equivalent for any context $C[\cdot]$.

From Theorem 10.1 it follows that: $\equiv_{op} \Rightarrow \equiv_{den}$.
As pointed out in Example 10.1: $\equiv_{den} \not\Rightarrow \equiv_{op}$.

So it is in this sense that we can say that the denotational semantics is *more abstract* then the operational one, because the former identifies more terms than the latter. Note that if we assume $t_0 \equiv_{den} t_1$ and $[\![t_0]\!]\,\rho \neq \bot$ then we can only conclude that $t_0 \to c_0$ and $t_1 \to c_1$ for some canonical forms c_0 and c_1. We have $[\![c_0]\!]\,\rho = [\![c_1]\!]\,\rho$, but nothing ensures that $c_0 = c_1$ (see Example 10.1 at the beginning of this chapter).

Only when we restrict our attention to the terms of HOFL that are typed as integers, then the corresponding operational and denotational semantics fully agree. This is because if c_0 and c_1 are canonical forms in C_{int} then it holds that $[\![c_0]\!]\,\rho = [\![c_1]\!]\,\rho \Leftrightarrow c_0 = c_1$. It can be proved *int* is the only type for which the full correspondence holds.

Theorem 10.4. *Let t : int be a closed term of HOFL and $n \in \mathbb{Z}$. Then:*

$$\forall \rho.\, [\![t]\!]\,\rho = \lfloor n \rfloor \quad \Leftrightarrow \quad t \to n$$

Proof. We prove the two implications separately.

⇒) If $[\![t]\!]\rho = \lfloor n \rfloor$, then $t \Downarrow$ and thus $t \downarrow$ by the soundness of denotational semantics, namely $\exists m$ such that $t \to m$, but then $[\![t]\!]\rho = \lfloor m \rfloor$ by Theorem 10.1, thus $n = m$ and $t \to n$.

⇐) Just Theorem 10.1, because $[\![n]\!]\rho = \lfloor n \rfloor$. □

10.5 A Simpler Denotational Semantics

We conclude this chapter by presenting a simpler denotational semantics which we call *unlifted*, because it does not use the lifted domains. This semantics is simpler but also less expressive than the lifted one. We define the following new domains:

$$D_{int} \stackrel{\text{def}}{=} \mathbb{Z}_\perp \qquad D_{\tau_1 * \tau_2} \stackrel{\text{def}}{=} D_{\tau_1} \times D_{\tau_2} \qquad D_{\tau_1 \to \tau_2} \stackrel{\text{def}}{=} [D_{\tau_1} \to D_{\tau_2}]$$

Now we can let $Env' \stackrel{\text{def}}{=} Var \to \bigcup_\tau D_\tau$ and define the simpler interpretation function $[\![t : \tau]\!]' : Env' \to D_\tau$ as follows (where $\rho \in Env'$):

(exactly as before)

$$\begin{aligned}
[\![n]\!]'\rho &= \lfloor n \rfloor \\
[\![x]\!]'\rho &= \rho(x) \\
[\![t_1 \text{ op } t_2]\!]'\rho &= [\![t_1]\!]'\rho \ \underline{\text{op}}_\perp \ [\![t_2]\!]'\rho \\
[\![\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2]\!]'\rho &= \mathit{Cond}([\![t_0]\!]'\rho, [\![t_1]\!]'\rho, [\![t_2]\!]'\rho) \\
[\![\textbf{rec } x.\ t]\!]'\rho &= \text{fix}\,\lambda d.\ [\![t]\!]'\rho[^d/_x]
\end{aligned}$$

(updated definitions)

$$\begin{aligned}
[\![(t_1, t_2)]\!]'\rho &= ([\![t_1]\!]'\rho, [\![t_2]\!]'\rho) \\
[\![\textbf{fst}(t)]\!]'\rho &= \pi_1([\![t]\!]'\rho) \\
[\![\textbf{snd}(t)]\!]'\rho &= \pi_2([\![t]\!]'\rho) \\
[\![\lambda x.\ t]\!]'\rho &= \lambda d.\ [\![t]\!]'\rho[^d/_x] \\
[\![(t_1\ t_2)]\!]'\rho &= ([\![t_1]\!]'\rho)\ ([\![t_2]\!]'\rho)
\end{aligned}$$

Note that the "unlifted" semantics differ from the "lifted" one only in the cases of pairing, projections, abstraction and application. On the one hand the unlifted denotational semantics is much simpler to read than the lifted one. On the other hand the unlifted semantics is more abstract than the lifted one and does not express some interesting properties. For instance, consider the two HOFL terms:

$$t_1 \stackrel{\text{def}}{=} \textbf{rec } x.\ x \ :\ \mathit{int} \to \mathit{int} \quad \text{and} \quad t_2 \stackrel{\text{def}}{=} \lambda x.\ \textbf{rec } y.\ y \ :\ \mathit{int} \to \mathit{int}$$

In the lifted semantics we have $[\![t_1]\!]\rho = \perp_{[\mathbb{Z}_\perp \longrightarrow \mathbb{Z}_\perp]_\perp}$ and $[\![t_2]\!]\rho = \lfloor \perp_{[\mathbb{Z}_\perp \longrightarrow \mathbb{Z}_\perp]} \rfloor$, thus

$$t_1 \not\Downarrow \quad \text{and} \quad t_2 \Downarrow .$$

In the unlifted semantics $[\![t_1]\!]'\rho = [\![t_2]\!]'\rho = \perp_{[\mathbb{Z}_\perp \longrightarrow \mathbb{Z}_\perp]}$, thus

$$t_1 \not\Downarrow' \quad \text{and} \quad t_2 \not\Downarrow' .$$

Note however that $t_1 \uparrow$ while $t_2 \downarrow$, thus the property $t \downarrow \Rightarrow t \Downarrow'$ does not hold, at least for some $t : \mathit{int} \to \mathit{int}$, since $t_2 \downarrow$ but $t_2 \not\Downarrow'$. However, the property holds for the unlifted semantics in the case of integers.

As a concluding remark, we observe 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.

Problems

10.1. Prove that the HOFL terms:

$$t_1 \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ ((\lambda y.\ 1)\ (f\ x)) \qquad t_2 \stackrel{\text{def}}{=} \lambda x.\ 1$$

have the same type and the same denotational semantics but different canonical forms.

10.2. Let us consider the HOFL term

$$\mathit{map} \stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ ((f\ \mathbf{fst}(x)), (f\ \mathbf{snd}(x)))$$

from Problem 7.5.

1. Write the denotational semantics of *map* and of $(\mathit{map}\ \lambda z.\ z)$.
2. Give two terms $t_1 : \mathit{int}$ and $t_2 : \mathit{int}$ such that the terms

$$((\mathit{map}\ \lambda z.\ z)(t_1, t_2)) \qquad ((\mathit{map}\ \lambda z.\ z)(t_2, t_1))$$

have different canonical forms but the same denotational semantics.

10.3. Consider the HOFL term

$$t \stackrel{\text{def}}{=} \mathbf{rec}\ x.\ ((\lambda y.\ \mathbf{if}\ y\ \mathbf{then}\ 0\ \mathbf{else}\ 0)\ x).$$

from Problem 7.6. Compute its denotational semantics, checking the equivalence with its operational semantics.

10.4. Consider the HOFL term:

$$t \stackrel{\text{def}}{=} \mathbf{rec}\ f.\ \lambda x.\ \mathbf{if}\ \mathbf{fst}(x) \times \mathbf{snd}(x)\ \mathbf{then}\ x\ \mathbf{else}\ (f\ (f\ x)).$$

Derive the type, the canonical form and the denotational semantics of t. Finally show another term t' with the same denotational semantics as t but with different canonical form.

10.5. Consider the HOFL term:

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ \textbf{fst}(x) - \textbf{snd}(x)\ \textbf{then}\ x\ \textbf{else}\ (f\ x).$$

Derive the type, the canonical form and the denotational semantics of t. Finally show another term t' with the same denotational semantics as t but with different canonical form.

10.6. Consider the HOFL term:

$$t \stackrel{\text{def}}{=} \textbf{rec}\ F.\ \lambda f.\ \lambda n.\ \textbf{if}\ (f\ n)\ \textbf{then}\ 0\ \textbf{else}\ ((F\ f)\ n).$$

Derive the type, the canonical form and the denotational semantics of t. Finally show another term t' with the same denotational semantics as t but with different canonical form.

10.7. Consider the HOFL term:

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ (\textbf{fst}(x)\ \ \textbf{snd}(x))\ \textbf{then}\ x\ \textbf{else}\ (f\ x).$$

Derive the type, the canonical form and the denotational semantics of t. Finally show another term t' with the same denotational semantics as t but with different canonical form.

10.8. Modify the ordinary HOFL semantics by defining the denotational semantics of the conditional construct as follows

$$[\![\textbf{if}\ t\ \textbf{then}\ t_0\ \textbf{else}\ t_1]\!]\rho = \mathit{Condd}([\![t]\!]\rho, [\![t_0]\!]\rho, [\![t_1]\!]\rho)$$

where

$$\mathit{Condd}(z, z_0, z_1) = \begin{cases} z_0 & \text{if } z = \lfloor 0 \rfloor \ \vee\ z_0 = z_1 \\ z_1 & \text{if } z = \lfloor n \rfloor \ \wedge\ n \neq 0 \\ \bot & \text{otherwise} \end{cases}$$

Assume that $t_0, t_1 : \mathit{int}$.

1. Prove that *Condd* is a monotonic, continuous function.
2. Show a HOFL term with a different semantics than the ordinary, and explain how the relation between operational and denotational semantics of HOFL is actually changed.

10.9. Modify the semantics of HOFL assuming the following operational semantics for the conditional command:

$$\frac{t_0 \to 0 \quad t_1 \to c_1 \quad t_2 \to c_2}{\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \to\ c_1} \qquad \frac{t_0 \to n \quad n \neq 0 \quad t_1 \to c_1 \quad t_2 \to c_2}{\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \to\ c_2}.$$

1. Exibit the corresponding denotational semantics.
2. Prove that also for the modified semantics it holds that $t \to c$ implies $[\![t]\!] = [\![c]\!]$.

3. Finally, compute the operational and the denotational semantics of $(\mathit{fact}\ 0)$, with

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

and check if they coincide.

10.10. Suppose the operational semantics of projections is changed

$$\text{from} \quad \frac{t \to (t_1, t_2) \quad t_1 \to c}{\mathbf{fst}(t) \to c} \qquad \text{to} \quad \frac{t \to (t_1, t_2) \quad t_1 \to c \quad t_2 \to c'}{\mathbf{fst}(t) \to c}$$

and analogously for **snd**, without changing the denotational semantics.

1. Prove that the property $t \to c \Rightarrow [\![t]\!]\rho = [\![c]\!]\rho$ is still valid.
2. Exhibit a counterexample showing that the property $[\![t]\!]\rho \neq \bot \Rightarrow t \to c$ is no longer valid.
3. Finally, modify the denotational semantics to recover the above property and illustrate its validity for the counterexample previously proposed.

10.11. Modify the operational semantics of HOFL by taking the following rules for conditionals:

$$\frac{t \to 0 \quad t_0 \to c_0 \quad t_1 \to c_1}{\mathbf{if}\ t\ \mathbf{then}\ t_0\ \mathbf{else}\ t_1 \to c_0} \qquad \frac{t \to n \quad n \neq 0 \quad t_0 \to c_0 \quad t_1 \to c_1}{\mathbf{if}\ t\ \mathbf{then}\ t_0\ \mathbf{else}\ t_1 \to c_1}.$$

without changing the denotational semantics. Prove that:

1. for any term t and canonical form c, we have $t \to c \ \Rightarrow\ \forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho$;
2. in general $t \Downarrow \not\Rightarrow t \downarrow$ (and exhibit a counterexample).

10.12. Suppose we extend HOFL with the inference rule:

$$\frac{t_1 \to 0}{t_1 \times t_2 \to 0}$$

as in Problem, 7.12.

1. Exhibit a counterexample showing that the property

$$\forall t, c.\quad t \to c \quad \Rightarrow \quad \forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho$$

is no longer valid.
2. Modify the denotational semantics so that the above correspondence is obtained, and prove that this is the case.
3. Repeat the exercise adding also the inference rule:

$$\frac{t_2 \to 0}{t_1 \times t_2 \to 0}.$$

10.13. Prove formally that

$$\text{if} \quad x \notin \mathrm{fv}(t) \quad \text{then} \quad \textbf{rec}\ x.\ t \text{ is equivalent to } t$$

employing both the operational and the denotational semantics.

10.14. Assume that the HOFL term t_0 has c_0 as canonical form.

1. Exploit the Substitution Lemma (Theorem 9.4) to prove that for every term t_1' we have
$$[\![t_1'[^{t_0}/_x]]\!]\,\rho \;=\; [\![t_1'[^{c_0}/_x]]\!]\,\rho.$$
2. Prove that if $t_1' : \mathit{int}$ and $\mathrm{fv}(t_1') \subseteq \{x\}$, then $t_1'[^{t_0}/_x] \equiv_{op} t_1'[^{c_0}/_x]$.
3. Conclude that if we replace the lazy evaluation rule
$$\frac{t_1 \to \lambda x.\ t_1' \quad t_1'[t_0/x] \to c}{(t_1\ t_0) \to c}$$
with the eager rule
$$\frac{t_1 \to \lambda x.\ t_1' \quad t_0 \to c_0 \quad t_1'[c_0/x] \to c}{(t_1\ t_0) \to c}$$
then, if $(t_1\ t_0) \to c : \mathit{int}$ in the eager semantics, then $(t_1\ t_0) \to c$ in the lazy semantics.
4. Exhibit a simple counterexample such that $\exists c.\ (t_1\ t_0) \to c$ according to the lazy semantics but not to the eager one.
5. Finally, exhibit another counterexample where the type of t_1' is not *int* and the properties at points 2 and 3 do not hold.

10.15. Extend the operational semantics of HOFL to non-closed terms, by allowing canonical forms that are not closed but otherwise keeping the same inference rules. Show an example of reduction to canonical form for a non closed term t. Then, prove that the following properties are still valid:

1. subject reduction: $t : \tau$ and $t \to c$ implies $c : \tau$;
2. $t \to c$ implies $[\![t]\!]\,\rho = [\![c]\!]\,\rho$ (remind that the Substitution Lemma holds for any terms, also not closed ones);
3. $t \downarrow$ implies $t \Downarrow$;
4. $t_1 : \mathit{int} \to c_1$, $t_2 : \mathit{int} \to c_2$ and $[\![t_1]\!] = [\![t_2]\!]$ imply $c_1 = c_2$;
5. $t \to c$ implies $[\![t[^{\textbf{rec}\ z.\ z}/_x]]\!]\,\rho = [\![c[^{\textbf{rec}\ z.\ z}/_x]]\!]\,\rho$.

Hint: Exploit property 2 above and the Substitution Lemma.

10.16. Modify the denotational semantics of HOFL by restricting the use of the *lifting* domain construction only to integers, namely $V_{\mathit{int}} = \mathbb{Z}_\perp$ but $V_{\tau_1 * \tau_2} = V_{\tau_1} \times V_{\tau_2}$ and similarly for functions.

1. List all the modified clauses of the denotational semantics.
2. Prove that $t \to c$ implies $[\![t]\!]\,\rho = [\![c]\!]\,\rho$.
3. Finally, prove that it is not true that $t \to c$ implies $[\![t]\!]\,\rho \neq \perp$.

Hint: consider the HOFL term $t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ (f\ x)\ :\ \mathit{int} \to \mathit{int}$.

Solutions

Problems of Chapter 2

2.1

1. The strings in L_B are all non-empty sequences of $\mathtt{b}$'s. The strings in L_A are all non-empty sequences of $\mathtt{a}$'s followed by strings in L_B.
2. Letting s^n denote the string obtained by concatenating n replicas of the string s, we have $L_B = \{\mathtt{b}^n \mid n > 0\}$ and $L_A = \{\mathtt{a}^n\mathtt{b}^m \mid n,m > 0\}$.
3. $$\frac{s \in L_A}{\mathtt{a}\ s \in L_A}(1) \qquad \frac{s \in L_B}{\mathtt{a}\ s \in L_A}(2) \qquad \frac{}{\mathtt{b} \in L_B}(3) \qquad \frac{s \in L_B}{\mathtt{b}\ s \in L_B}(4)$$
4. Proof tree:
$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{b} \in L_B}(3)}{\mathtt{b}\,\mathtt{b} \in L_B}(4)}{\mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A}(2)}{\mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A}(1)}{\mathtt{a}\,\mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A}(1)$$
Goal-oriented derivation:
$$\begin{aligned} \mathtt{a}\,\mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A &\nwarrow \mathtt{a}\,\mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A \\ &\nwarrow \mathtt{a}\,\mathtt{b}\,\mathtt{b} \in L_A \\ &\nwarrow \mathtt{b}\,\mathtt{b} \in L_B \\ &\nwarrow \mathtt{b} \in L_B \\ &\nwarrow \square \end{aligned}$$
5. We first prove the correspondence for B, i.e., that $s \in L_B$ is a theorem iff there exists some $n > 0$ with $s = \mathtt{b}^n$. For the 'only if' part, by rule induction, since $s \in L_B$, either $s = \mathtt{b}$ (by rule (3)), or $s = \mathtt{b}\ s'$ for some $s' \in L_B$ (by rule (4)). In the former case, we take $n = 1$ and we are done. In the latter case, by $s' \in L_B$ we have that there is $n' > 0$ with $s' = \mathtt{b}^{n'}$ and take $n = n' + 1$. For the 'if' part, by induction on n, if $n = 1$ we conclude by applying axiom (3); if $n = n' + 1$, we can assume that $\mathtt{b}^{n'} \in L_B$ and conclude by applying rule (4).
Then we prove the correspondence for A, i.e., that $s \in L_A$ is a theorem iff there exists some $n,m > 0$ with $s = \mathtt{a}^n\mathtt{b}^m$. For the 'only if' part, by rule induction, since $s \in L_A$, either $s = \mathtt{a}\ s'$ for some $s' \in L_A$ (by rule (1)), or $s = \mathtt{a}\ s'$ for some

$s' \in L_B$ (by rule (2)). In the former case, by $s' \in L_A$ we have that there is $n', m' > 0$ with $s' = \mathtt{a}^{n'}\mathtt{b}^{n'}$ and take $n = n' + 1$, $m = m'$. In the latter case, by the previous correspondence on B, by $s' \in L_B$ we have that $s' = \mathtt{b}^k$ for some $k > 0$ and conclude by taking $n = 1$ and $m = k$. For the 'if' part, take $s = \mathtt{a}^n\mathtt{b}^m$. By induction on n, if $n = 1$ we conclude by applying axiom (2), since for the previous correspondence we know that $\mathtt{b}^m \in L_B$; if $n = n' + 1$, we can assume that $\mathtt{a}^{n'}\mathtt{b}^m \in L_A$ and conclude by applying rule (1).

2.3

1. The predicate $even(x)$ is a theorem iff x represents an even number (i.e., x is the repeated application of $s(\cdot)$ to 0 for an even number of times).
2. The predicate $odd(x)$ if not a theorem for any x, because there is no axiom.
3. The predicate $leq(x,y)$ is a theorem iff x represents a natural number which is less than or equal to the natural number represented by y.

2.5 Take $t = s(x)$ and $t' = s(y)$.

2.8

$$\begin{aligned} \mathsf{fib}(0,1) &:- . \\ \mathsf{fib}(s(0),1) &:- . \\ \mathsf{fib}(s(s(x)),y) &:- \mathsf{fib}(x,u), \mathsf{fib}(s(x),v), \mathsf{sum}(u,v,y). \end{aligned}$$

2.11 Pgvdrk is intelligent.

Problems of Chapter 3

3.2 Let us denote by c the body of the while command:

$$c \stackrel{\text{def}}{=} \textbf{if } y = 0 \textbf{ then } y := y+1 \textbf{ else skip}$$

Let us take a generic memory σ and consider the goal $\langle w, \sigma\rangle \to \sigma'$.
If $\sigma(y) < 0$ we have:

$$\begin{array}{l} \langle w,\sigma\rangle \to \sigma' \\ \quad \nwarrow_{\sigma'=\sigma} \ \langle y \geq 0, \sigma\rangle \to \textbf{false} \\ \quad\quad \nwarrow^* \ \ \Box \end{array}$$

If instead $\sigma(y) > 0$, we have:

$$\begin{array}{lll} \langle w,\sigma\rangle \to \sigma' & \nwarrow & \langle y \geq 0, \sigma\rangle \to \textbf{true}, \langle c,\sigma\rangle \to \sigma'', \langle w,\sigma''\rangle \to \sigma' \\ & \nwarrow^* & \langle c,\sigma\rangle \to \sigma'', \langle w,\sigma''\rangle \to \sigma' \\ & \nwarrow^* & \langle \textbf{skip},\sigma\rangle \to \sigma'', \langle w,\sigma''\rangle \to \sigma' \\ & \nwarrow^*_{\sigma''=\sigma} & \langle w,\sigma\rangle \to \sigma' \end{array}$$

Since we reach the same goal from which we started, the command diverges.
Finally, if instead $\sigma(y) = 0$, we have:

$$\begin{array}{lll}
\langle w, \sigma\rangle \to \sigma' & \nwarrow & \langle y \geq 0, \sigma\rangle \to \textbf{true}\,, \langle c, \sigma\rangle \to \sigma''\,, \langle w, \sigma''\rangle \to \sigma' \\
& \nwarrow^* & \langle c, \sigma\rangle \to \sigma''\,, \langle w, \sigma''\rangle \to \sigma' \\
& \nwarrow^* & \langle y := y+1, \sigma\rangle \to \sigma''\,, \langle w, \sigma''\rangle \to \sigma' \\
& \nwarrow^*_{\sigma''=\sigma[^1/_y]} & \langle w, \sigma[^1/_y]\rangle \to \sigma'
\end{array}$$

We reach a goal where w is to be evaluated in a memory $\sigma[^1/_y]$ such that $\sigma[^1/_y](y) > 0$. Thus we are in the previous case and we know that the command diverges.

Summing up, $\langle w, \sigma\rangle \to \sigma'$ iff $\sigma(y) < 0 \wedge \sigma' = \sigma$.

3.4 Let us denote by c' the body of c_2:

$$c' \stackrel{\text{def}}{=} \textbf{if } b \textbf{ then } c \textbf{ else skip}$$

We proceed by contradiction. First, assume that there exist σ, σ' such that $\langle c_1, \sigma\rangle \to \sigma'$ and $\langle c_2, \sigma\rangle \not\to \sigma'$. Let us take such σ, σ' for which $\langle c_1, \sigma\rangle \to \sigma'$ has the shortest derivation.

If $\langle b, \sigma\rangle \to \textbf{false}$, we have

$$\begin{array}{lll}
\langle c_1, \sigma\rangle \to \sigma' & \nwarrow_{\sigma'=\sigma} & \langle b, \sigma\rangle \to \textbf{false} \\
& \nwarrow^* & \Box \\
\langle c_2, \sigma\rangle \to \sigma' & \nwarrow_{\sigma'=\sigma} & \langle b, \sigma\rangle \to \textbf{false} \\
& \nwarrow^* & \Box
\end{array}$$

Thus it must be $\langle b, \sigma\rangle \to \textbf{true}$. In this case, we have

$$\begin{array}{lll}
\langle c_1, \sigma\rangle \to \sigma' & \nwarrow_{\sigma'=\sigma} & \langle b, \sigma\rangle \to \textbf{true}\,, \langle c, \sigma\rangle \to \sigma''\,, \langle c_1, \sigma''\rangle \to \sigma' \\
& \nwarrow^* & \langle c, \sigma\rangle \to \sigma''\,, \langle c_1, \sigma''\rangle \to \sigma' \\
\langle c_2, \sigma\rangle \to \sigma' & \nwarrow_{\sigma'=\sigma} & \langle b, \sigma\rangle \to \textbf{true}\,, \langle c', \sigma\rangle \to \sigma''\,, \langle c_2, \sigma''\rangle \to \sigma' \\
& \nwarrow^* & \langle c', \sigma\rangle \to \sigma''\,, \langle c_2, \sigma''\rangle \to \sigma' \\
& \nwarrow & \langle b, \sigma\rangle \to \textbf{true}\,, \langle c, \sigma\rangle \to \sigma''\,, \langle c_2, \sigma''\rangle \to \sigma' \\
& \nwarrow^* & \langle c, \sigma\rangle \to \sigma''\,, \langle c_2, \sigma''\rangle \to \sigma'
\end{array}$$

Now, since σ and σ' were chosen so to allow for the shortest derivation $\langle c_1, \sigma\rangle \to \sigma'$ that cannot be mimicked by $\langle c_2, \sigma\rangle$, it must be the case that $\langle c_1, \sigma''\rangle \to \sigma'$, which is shorter, can still be mimicked, thus $\langle c_2, \sigma''\rangle \to \sigma'$ is provable, but then $\langle c_2, \sigma\rangle \to \sigma'$ holds, leading to a contradiction.

Second, assume that there exist σ, σ' such that $\langle c_2, \sigma\rangle \to \sigma'$ and $\langle c_1, \sigma\rangle \not\to \sigma'$. Then the proof is completed analogously to the previous case.

3.6 Take any σ such that $\sigma(x) = 0$. Then $\langle c_1, \sigma\rangle \to \sigma$, while $\langle c_2, \sigma\rangle \not\to$.

3.9

1. Take $a = 0/0$. Then, for any σ, we have, e.g., $\langle a, \sigma\rangle \to 1$ (since $0 = 0 \times 1$) and $\langle a, \sigma\rangle \to 2$ (since $0 = 0 \times 2$) by straightforward application of rule (div).
2. Take $a = 1/2$. Then, we cannot find an integer n such that $1 = 2 \times n$ and the rule (div) cannot be applied.

Problems of Chapter 4

4.2 We let:

$$\begin{aligned} locs(\textbf{skip}) &\stackrel{\text{def}}{=} \varnothing \\ locs(x := a) &\stackrel{\text{def}}{=} \{x\} \\ locs(c_0; c_1) \;=\; locs(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) &\stackrel{\text{def}}{=} locs(c_0) \cup locs(c_1) \\ locs(\textbf{while } b \textbf{ do } c) &\stackrel{\text{def}}{=} locs(c) \end{aligned}$$

We prove the property

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

by rule induction.

skip: We need to prove $P(\langle \textbf{skip}, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \forall y \notin locs(\textbf{skip}).\ \sigma(y) = \sigma(y)$ that holds trivially.

assign: We need to prove

$$P(\langle x := a, \sigma\rangle \to \sigma[^n/_x]) \stackrel{\text{def}}{=} \forall y \notin locs(x := a).\ \sigma(y) = \sigma[^n/_x](y)$$

Trivially: $locs(x := a) = \{x\}$ and $\forall y \neq x.\ \sigma[^n/_x](y) = \sigma(y)$.

seq: We assume

$$\begin{aligned} P(\langle c_0, \sigma\rangle \to \sigma'') &\stackrel{\text{def}}{=} \forall y \notin locs(c_0).\ \sigma(y) = \sigma''(y) \\ P(\langle c_1, \sigma''\rangle \to \sigma') &\stackrel{\text{def}}{=} \forall y \notin locs(c_0).\ \sigma''(y) = \sigma'(y) \end{aligned}$$

and we need to prove

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

Take $y \notin locs(c_0;c_1) = locs(c_0) \cup locs(c_1)$. It follows that $y \notin locs(c_0)$ and $y \notin locs(c_1)$. By $y \notin locs(c_0)$ and the first inductive hypothesis we have $\sigma(y) = \sigma''(y)$. By $y \notin locs(c_1)$ and the second inductive hypothesis we have $\sigma''(y) = \sigma'(y)$. By transitivity, we conclude $\sigma(y) = \sigma'(y)$.

iftt: We assume

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

and we need to prove

$$P(\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall y \notin locs(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1).\ \sigma(y) = \sigma'(y)$$

Take $y \notin locs(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) = locs(c_0) \cup locs(c_1)$. It follows that $y \notin locs(c_0)$ and hence, by the inductive hypothesis, $\sigma(y) = \sigma'(y)$.

ifff: This case is analogous to the previous one and thus omitted.

whff: We need to prove

$$P(\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \forall y \notin locs(\textbf{while } b \textbf{ do } c).\ \sigma(y) = \sigma(y)$$

which is obvious (as for the case of rule skip).

whtt: We assume

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

$$P(\langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \forall y \notin locs(\textbf{while } b \textbf{ do } c).\ \sigma''(y) = \sigma'(y)$$

and we need to prove

$$P(\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \forall y \notin locs(\textbf{while } b \textbf{ do } c).\ \sigma(y) = \sigma'(y)$$

Take $y \notin locs(\textbf{while } b \textbf{ do } c) = locs(c)$. By the first inductive hypothesis, it follows that $\sigma(y) = \sigma''(y)$, while by the second inductive hypothesis we have $\sigma''(y) = \sigma'(y)$. By transitivity, we conclude $\sigma(y) = \sigma'(y)$.

4.3 We prove the property

$$P(\langle w, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \sigma(x) \geq 0 \ \wedge\ \sigma' = \sigma\left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x}\right]$$

by rule induction. Since the property is concerned with the command w, it is enough to consider the two rules for the while construct.

whff: We assume

$$\langle x \neq 0, \sigma\rangle \to \textbf{false}$$

We need to prove

$$P(\langle w, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \sigma(x) \geq 0 \ \wedge\ \sigma = \sigma\left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x}\right]$$

Since $\langle x \neq 0, \sigma\rangle \to \textbf{false}$ it follows that $\sigma(x) = 0$ and thus $\sigma(x) \geq 0$. Then, $\sigma\left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x}\right] = \sigma\left[{}^{0+\sigma(y)}/_{y}, {}^{\sigma(x)}/_{x}\right] = \sigma\left[{}^{\sigma(y)}/_{y}, {}^{\sigma(x)}/_{x}\right] = \sigma$.

whtt: Let $c \stackrel{\text{def}}{=} x := x-1; y := y+1$. We assume

$$\langle x \neq 0, \sigma\rangle \to \textbf{false} \qquad \langle c, \sigma\rangle \to \sigma'' \qquad \langle w, \sigma''\rangle \to \sigma'$$
$$P(\langle w, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \sigma''(x) \geq 0 \ \wedge\ \sigma' = \sigma''\left[{}^{\sigma''(x)+\sigma''(y)}/_{y}, {}^{0}/_{x}\right]$$

We need to prove

$$P(\langle w, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \sigma(x) \geq 0 \ \wedge\ \sigma' = \sigma\left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x}\right]$$

From $\langle c, \sigma\rangle \to \sigma''$ it follows that $\sigma'' = \sigma\left[{}^{\sigma(y)+1}/_{y}, {}^{\sigma(x)-1}/_{x}\right]$. By inductive hypothesis we have $\sigma''(x) \geq 0$, thus $\sigma(x) \geq 1$ and hence $\sigma(x) \geq 0$. Moreover, by inductive hypothesis, we have also

$$\sigma' = \sigma''\left[{}^{\sigma''(x)+\sigma''(y)}/_{y}, {}^{0}/_{x}\right] = \sigma''\left[{}^{\sigma(x)-1+\sigma(y)+1}/_{y}, {}^{0}/_{x}\right] = \sigma''\left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x}\right] = \sigma\left[{}^{\sigma(x)+\sigma(y)}/_{y}, {}^{0}/_{x}\right].$$

4.4 We prove the two implication separately. First we prove the property

$$P(x\, R^{+}\, y) \stackrel{\text{def}}{=} \exists k > 0.\ \exists z_0, \ldots, z_k.\ x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ \ldots \wedge\ z_{k-1}\, R\, z_k \ \wedge\ z_k = y$$

by rule induction.

For the first rule

$$\frac{x\, R\, y}{x\, R^{+}\, y}$$

we assume $x\, R\, y$ and we need to prove $P(x\, R^{+}\, y)$. We take $k = 1$, $z_0 = x$ and $z_1 = y$ and we are done.

For the second rule

$$\frac{x\, R^{+}\, y \quad y\, R^{+}\, z}{x\, R^{+}\, z}$$

we assume

$$P(x\, R^{+}\, y) \stackrel{\text{def}}{=} \exists n > 0.\ \exists u_0, \ldots, u_n.\ x = u_0 \ \wedge\ u_0\, R\, u_1 \ \wedge\ \ldots \wedge\ u_{n-1}\, R\, u_n \ \wedge\ u_n = y$$
$$P(y\, R^{+}\, z) \stackrel{\text{def}}{=} \exists m > 0.\ \exists v_0, \ldots, v_m.\ y = v_0 \ \wedge\ v_0\, R\, v_1 \ \wedge\ \ldots \wedge\ v_{m-1}\, R\, v_m \ \wedge\ v_m = z$$

and we need to prove

$$P(x\, R^{+}\, z) \stackrel{\text{def}}{=} \exists k > 0.\ \exists z_0, \ldots, z_k.\ x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ \ldots \wedge\ z_{k-1}\, R\, z_k \ \wedge\ z_k = z$$

Take $n, u_0, ..., u_n$ and $m, v_0, ..., v_m$ as provided by the inductive hypotheses. We set $k = n+m$, from which it follows $k > 0$ since $n > 0$ and $m > 0$. Note that $u_n = y = v_0$.

Finally, we let

$$z_i \stackrel{\text{def}}{=} \begin{cases} u_i & \text{if } i \in [0,n] \\ v_{i-n} & \text{if } i \in [n+1,k] \end{cases}$$

and it is immediate to check that the conditions are satisfied.

To prove the reverse implication, we exploit the logical equivalence

$$(\exists k.\, A(k)) \Rightarrow B \quad \Leftrightarrow \quad \forall k.\, (A(k) \Rightarrow B)$$

that holds whenever k does not appear (free) in the predicate B, to prove the universally quantified statement

$$\forall k > 0. \forall x, y. \left((\exists z_0, \ldots, z_k.\, x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ \ldots \wedge\ z_{k-1}\, R\, z_k \ \wedge\ z_k = y) \Rightarrow x\, R^+\, y \right)$$

by mathematical induction on k.

The base case is when $k = 1$. Take generic x and y. We assume the premise

$$\exists z_0, z_1.\, x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ z_1 = y$$

and the thesis $x\, R^+\, y$ follows by applying the first inference rule.

For the inductive case, we assume that

$$\forall x, y. \ \left((\exists z_0, \ldots, z_k.\, x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ \ldots \wedge\ z_{k-1}\, R\, z_k \ \wedge\ z_k = y) \Rightarrow x\, R^+\, y \right)$$

and we want to prove that

$$\forall x, z. \ \left((\exists z_0, \ldots, z_{k+1}.\, x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ \ldots \wedge\ z_k\, R\, z_{k+1} \ \wedge\ z_{k+1} = z) \Rightarrow x\, R^+\, z \right)$$

Take generic x, z and assume that there exist $z_0, \ldots, z_{k+1}$ satisfying the premise of the implication:

$$x = z_0 \ \wedge\ z_0\, R\, z_1 \ \wedge\ \ldots \wedge\ z_k\, R\, z_{k+1} \ \wedge\ z_{k+1} = z$$

By the inductive hypothesis, it follows that $x\, R^+\, z_k$. Moreover, from $z_k\, R\, z_{k+1} = z$ we can apply the first inference rule to derive $z_k\, R^+\, z$. Finally, we conclude by applying the second inference rule to $x\, R^+\, z_k$ and $z_k\, R^+\, z$, obtaining $x\, R^+\, z$.

Regarding the second question, the relation R' is just the reflexive and transitive closure of R.

Problems of Chapter 5

5.2

1. It can be readily checked that f is monotone: let us take $S_1, S_2 \in \wp(\mathbb{N})$, with $S_1 \subseteq S_2$; we need to check that $f(S_1) \subseteq f(S_2)$. Let $x \in f(S_1) = S_1 \cap X$. Then $x \in S_1$ and $x \in X$. Since $S_1 \subseteq S_2$, we have also $x \in S_2$ and thus $x \in S_2 \cap X = f(S_2)$.

The case of g is more subtle. Intuitively, the larger is S, the smaller is $\wp(\mathbb{N}) \setminus S$ and consequently $(\wp(\mathbb{N}) \setminus S) \cap X$. Let us take $S_1, S_2 \in \wp(\mathbb{N})$, with $S_1 \subset S_2$, and let $x \in S_2 \setminus S_1$. Then $x \in \wp(\mathbb{N}) \setminus S_1$ and $x \notin \wp(\mathbb{N}) \setminus S_2$. Now if $x \in X$ we have $x \in g(S_1)$ and $x \notin g(S_2)$, contradicting the requirement $g(S_1) \subseteq g(S_2)$. Note that, unless $X = \varnothing$, such a counterexample can always be constructed.

2. Let us take a chain $\{S_i\}_{i \in \mathbb{N}}$ in $\wp(\mathbb{N})$. We need to prove that $f(\bigcup_{i \in \mathbb{N}} S_i) = \bigcup_{i \in \mathbb{N}} f(S_i)$, i.e., that $(\bigcup_{i \in \mathbb{N}} S_i) \cap X = \bigcup_{i \in \mathbb{N}} (S_i \cap X)$. We have

$$\begin{aligned} x \in \left(\bigcup_{i \in \mathbb{N}} S_i \right) \cap X &\Leftrightarrow x \in \left(\bigcup_{i \in \mathbb{N}} S_i \right) \wedge x \in X \\ &\Leftrightarrow \exists k \in \mathbb{N}.\, x \in S_k \wedge x \in X \\ &\Leftrightarrow \exists k \in \mathbb{N}.\, x \in S_k \cap X \\ &\Leftrightarrow x \in \bigcup_{i \in \mathbb{N}} (S_i \cap X) \end{aligned}$$

Since g is in general not monotone, it is not continuous (unless $X = \varnothing$, in which case g is the constant function returning $\varnothing$ and thus trivially monotone and continuous).

3. f is monotone and continuous for any X, while g is monotone and continuous only when $X = \varnothing$.

5.3

1. Let D_1 be the discrete order with two elements 0 and 1. All chains in D_1 are constant (and finite) and all functions $f : D_1 \to D_1$ are monotone and continuous. The identity function $f_1(x) = x$ has two fixpoints but no least fixpoint (as discussed also in Example 5.18).
2. Let $D_2 = D_1$. If we let $f_2(0) = 1$ and $f_2(1) = 0$, then f_2 has no fixpoint.
3. If D_3 is finite, then any chain is finite and any monotone function is continuous. So we must choose D_3 with infinitely many elements. We take D_3 and f_3 as in Example 5.17.

5.4 Let us take $D = \mathbb{N}$ with the usual "less than or equal to" order. As discussed in Chapter 5, it is a partial order with bottom but it is not complete, because, e.g., the chain of even numbers has no upper bound.

1. From what said above, the chain

$$0 \succ 2 \succ 4 \succ 6 \succ \cdots$$

is an infinite descending chain, and thus $\mathscr{D}'$ is not well-founded.

2. The answer is no: if $\mathscr{D}$ is not complete, then $\mathscr{D}'$ is not well-founded. To show this, let us take a chain

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots$$

that has no least upper bound (it must exists, because $\mathscr{D}$ is not complete). The chain $\{d_i\}_{i \in \mathbb{N}}$ cannot be finite, as otherwise the maximum element would be the least upper bound. However, it is not necessarily the case that

$$d_0 \succ d_1 \succ d_2 \succ \cdots$$

is an infinite descending chain of $\mathscr{D}'$, because $\{d_i\}_{i\in\mathbb{N}}$ can contain repeated elements. To discard clones, we define the function $next : \mathbb{N} \to \mathbb{N}$ to select the smallest index with which the ith different element appears in the chain (letting d_0 be the 0th element)

$$\begin{aligned} next(0) &\stackrel{\text{def}}{=} 0 \\ next(i+1) &\stackrel{\text{def}}{=} \min\{j \mid d_j \neq d_{j-1} \wedge next(i) < j\} \end{aligned}$$

and take the infinite descending chain

$$d_{next(0)} \succ d_{next(1)} \succ d_{next(2)} \succ \cdots$$

5.5

1. We need to check that $\sqsubseteq$ is reflexive, antisymmetric and transitive.

reflexive:	for any string $\alpha \in V^* \cup V^\infty$ we have $\alpha = \alpha\varepsilon$ and hence $\alpha \sqsubseteq \alpha$;
antisymmetric:	we assume $\alpha \sqsubseteq \beta$ and $\beta \sqsubseteq \alpha$ and we need to prove that $\alpha = \beta$; let γ and δ such that $\beta = \alpha\gamma$ and $\alpha = \beta\delta$, then $\alpha = \alpha\gamma\delta$: if $\alpha \in V^*$, then it must be $\gamma = \delta = \varepsilon$ and $\alpha = \beta$; if $\alpha \in V^\infty$, from $\beta = \alpha\gamma$ it follows $\beta = \alpha$;
transitive:	we assume $\alpha \sqsubseteq \beta$ and $\beta \sqsubseteq \gamma$ and we need to prove that $\alpha \sqsubseteq \gamma$; let δ and ω such that $\beta = \alpha\delta$ and $\gamma = \beta\omega$, then $\gamma = \alpha\delta\omega$ and thus $\alpha \sqsubseteq \gamma$.

2. To prove that the order is complete we must show that any chain has a limit. Take

   $$\alpha_0 \sqsubseteq \alpha_1 \sqsubseteq \alpha_2 \sqsubseteq \cdots \sqsubseteq \alpha_n \sqsubseteq \cdots$$

   If the chain is finite, then the greatest element of the chain is the least upper bound. Otherwise, it must be $\alpha_i \in V^*$ for any $i \in \mathbb{N}$ and for any length n we can find a string α_{k_n} in the sequence such that $|\alpha_{k_n}| \geq n$ (if not, the chain would be finite). Then we can construct a string $\alpha \in V^\infty$ such that for any position n in α the nth symbol of α appears in the same position in one of the strings in the chain. In fact we let $\alpha(n) \stackrel{\text{def}}{=} \alpha_{k_n}(n)$ and α is the limit of the chain.
3. The bottom element is the empty string ε, in fact for any $\alpha \in V^* \cup V^\infty$ we have $\varepsilon\alpha = \alpha$ and thus $\varepsilon \sqsubseteq \alpha$.
4. The maximal elements are all and only the strings in V^∞. In fact, on the one hand, taken $\alpha \in V^\infty$ we have

   $$\alpha \sqsubseteq \beta \quad \Leftrightarrow \quad \exists\gamma.\, \beta = \alpha\gamma \quad \Leftrightarrow \quad \beta = \alpha$$

   On the other hand, if $\alpha \in V^*$, then $\alpha \sqsubseteq \alpha a$ and $\alpha \neq \alpha a$.

Problems of Chapter 6

6.1

1. The expression $\lambda x.\ \lambda x.\ x$ is α-convertible to the expressions a, c, e.
2. The expression $((\lambda x.\ \lambda y.\ x)\ y)$ is equivalent to the expressions d and e.

6.3 Let $c'' \stackrel{\text{def}}{=} \textbf{if } x = 0 \textbf{ then } c_1 \textbf{ else } c_2$. Using the operational semantics we have:

$$\begin{array}{rll}
\langle c, \sigma\rangle \to \sigma' & \nwarrow & \langle x := 0, \sigma\rangle \to \sigma'', \quad \langle c'', \sigma''\rangle \to \sigma' \\
& \nwarrow *_{\sigma''=\sigma[^0/_x]} & \langle x = 0, \sigma[^0/_x]\rangle \to \textbf{true}, \quad \langle c_1, \sigma[^0/_x]\rangle \to \sigma' \\
& \nwarrow^* & \langle c_1, \sigma[^0/_x]\rangle \to \sigma' \\
\langle c', \sigma\rangle \to \sigma' & \nwarrow & \langle x := 0, \sigma\rangle \to \sigma'', \quad \langle c_1, \sigma''\rangle \to \sigma' \\
& \nwarrow *_{\sigma''=\sigma[^0/_x]} & \langle c_1, \sigma[^0/_x]\rangle \to \sigma'
\end{array}$$

Since both goals reduce to the same goal $\langle c_1, \sigma[^0/_x]\rangle \to \sigma'$, the two commands c and c' are equivalent.

Using the denotational semantics, we have:

$$\begin{aligned}
\mathscr{C}[\![c]\!]\sigma &= \mathscr{C}[\![c'']\!]^*(\mathscr{C}[\![x := 0]\!]\sigma) \\
&= \mathscr{C}[\![c'']\!]^*(\sigma[^0/_x]) \\
&= \mathscr{C}[\![c'']\!](\sigma[^0/_x]) \\
&= (\lambda\sigma'.\ (\mathscr{B}[\![x = 0]\!]\sigma' \to \mathscr{C}[\![c_1]\!]\sigma', \mathscr{C}[\![c_2]\!]\sigma'))(\sigma[^0/_x]) \\
&= \mathscr{B}[\![x = 0]\!]\sigma[^0/_x] \to \mathscr{C}[\![c_1]\!]\sigma[0/x], \mathscr{C}[\![c_2]\!]\sigma[0/x] \\
&= \textbf{true} \to \mathscr{C}[\![c_1]\!]\sigma[0/x], \mathscr{C}[\![c_2]\!]\sigma[0/x] \\
&= \mathscr{C}[\![c_1]\!]\sigma[0/x] \\
\mathscr{C}[\![c']\!]\sigma &= \mathscr{C}[\![c_1]\!]^*(\mathscr{C}[\![x := 0]\!]\sigma) \\
&= \mathscr{C}[\![c_1]\!]^*(\sigma[0/x]) \\
&= \mathscr{C}[\![c_1]\!](\sigma[0/x]).
\end{aligned}$$

6.4 Let $c' \stackrel{\text{def}}{=} \textbf{if } b \textbf{ then } c \textbf{ else skip}$. We have that

$$\begin{aligned}
\Gamma_{b,c}\ \varphi\ \sigma &= \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![c]\!]\sigma), \sigma \\
\Gamma_{b,c'}\ \varphi\ \sigma &= \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![c']\!]\sigma), \sigma \\
&= \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c]\!]\sigma, \mathscr{B}[\![\textbf{skip}]\!]\sigma), \sigma \\
&= \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c]\!]\sigma, \sigma), \sigma
\end{aligned}$$

Let us show that $\Gamma_{b,c} = \Gamma_{b,c'}$.
If $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$, then $\Gamma_{b,c}\ \varphi\ \sigma = \sigma = \Gamma_{b,c}\ \varphi\ \sigma$.
If $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$, then

$$\begin{aligned}\Gamma_{b,c}\ \varphi\ \sigma &= \varphi^*(\mathscr{C}[\![c]\!]\sigma)\\ \Gamma_{b,c'}\ \varphi\ \sigma &= \varphi^*(\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c]\!]\sigma, \sigma)\\ &= \varphi^*(\mathscr{C}[\![c]\!]\sigma)\end{aligned}$$

6.5 We have already seen in Example 6.6 that $\mathscr{C}[\![\textbf{while true do skip}]\!] = \lambda\sigma.\ \bot_{\Sigma_\bot}$.
For the second command we have $\mathscr{C}[\![\textbf{while true do } x := x+1]\!] = \text{fix } \Gamma$, where

$$\begin{aligned}\Gamma\ \varphi\ \sigma &= \mathscr{B}[\![\textbf{true}]\!]\sigma \to \varphi^*(\mathscr{C}[\![x := x+1]\!]\sigma), \sigma\\ &= \textbf{true} \to \varphi^*(\mathscr{C}[\![x := x+1]\!]\sigma), \sigma\\ &= \varphi^*(\mathscr{C}[\![x := x+1]\!]\sigma)\\ &= \varphi^*(\sigma[\sigma(x)+1/x])\\ &= \varphi(\sigma[\sigma(x)+1/x])\end{aligned}$$

Let us compute the first elements of the chain $\{\varphi_n\}_{n\in\mathbb{N}}$ with $\varphi_n = \Gamma^n\ \bot_{\Sigma\to\Sigma_\bot}$:

$$\begin{aligned}\varphi_0\ \sigma &= \bot_{\Sigma_\bot}\\ \varphi_1\ \sigma &= \Gamma\ \varphi_0\ \sigma\\ &= \varphi_0(\sigma[\sigma(x)+1/x])\\ &= (\lambda\sigma.\ \bot_{\Sigma_\bot})(\sigma[\sigma(x)+1/x])\\ &= \bot_{\Sigma_\bot}\end{aligned}$$

Since $\varphi_1 = \varphi_0$ we have reached the fixpoint and have $\mathscr{C}[\![\textbf{while true do } x := x+1]\!] = \lambda\sigma.\ \bot_{\Sigma_\bot}$.

6.6 We have immediately $\mathscr{C}[\![x := 0]\!]\sigma = \sigma[0/x]$.
Moreover, we have $\mathscr{C}[\![\textbf{while } x \neq 0 \textbf{ do } x := 0]\!] = \text{fix } \Gamma$, where

$$\begin{aligned}\Gamma\ \varphi\ \sigma &= \mathscr{B}[\![x \neq 0]\!]\sigma \to \varphi^*(\mathscr{C}[\![x := 0]\!]\sigma), \sigma\\ &= \sigma(x) \neq 0 \to \varphi^*(\sigma[0/x]), \sigma\\ &= \sigma(x) \neq 0 \to \varphi(\sigma[0/x]), \sigma\end{aligned}$$

Let us compute the first elements of the chain $\{\varphi_n\}_{n\in\mathbb{N}}$ with $\varphi_n = \Gamma^n\ \bot_{\Sigma\to\Sigma_\bot}$:

$$
\begin{aligned}
\varphi_0\ \sigma &= \bot_{\Sigma_\bot} \\
\varphi_1\ \sigma &= \Gamma\ \varphi_0\ \sigma \\
&= \sigma(x) \neq 0 \to \varphi_0(\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to \bot_{\Sigma_\bot}\,,\sigma \\
\varphi_2\ \sigma &= \Gamma\ \varphi_1\ \sigma \\
&= \sigma(x) \neq 0 \to \varphi_1(\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to (\lambda\sigma'.\ \sigma'(x) \neq 0 \to \bot_{\Sigma_\bot}\,,\sigma')(\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to (\sigma[0/x](x) \neq 0 \to \bot_{\Sigma_\bot}\,,\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to (\mathbf{false} \to \bot_{\Sigma_\bot}\,,\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to \sigma[0/x]\,,\sigma \\
&= \sigma(x) \neq 0 \to \sigma[0/x]\,,\sigma[0/x] \\
&= \sigma[0/x] \\
\varphi_3\ \sigma &= \Gamma\ \varphi_2\ \sigma \\
&= \sigma(x) \neq 0 \to \varphi_2(\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to (\lambda\sigma'.\ \sigma'[0/x])(\sigma[0/x])\,,\sigma \\
&= \sigma(x) \neq 0 \to \sigma[0/x][0/x]\,,\sigma \\
&= \sigma(x) \neq 0 \to \sigma[0/x]\,,\sigma[0/x] \\
&= \sigma[0/x]
\end{aligned}
$$

Note in fact that, when $\sigma(x) \neq 0$ is false, then $\sigma = \sigma[0/x]$.

Since $\varphi_3 = \varphi_2$ we have reached the fixpoint and have $\mathscr{C}[\![\mathbf{while}\ x \neq 0\ \mathbf{do}\ x := 0]\!] = \lambda\sigma.\ \sigma[0/x]$.

We conclude by observing that since φ_2 is a maximal element of its domain, it must be already the lub of the chain, namely the fixpoint. Thus it is not necessary to compute φ_3.

6.10

1.

$$\frac{\langle c,\sigma\rangle \to \sigma' \quad \langle b,\sigma'\rangle \to \mathbf{false}}{\langle \mathbf{do}\ c\ \mathbf{undoif}\ b,\sigma\rangle \to \sigma'}\ (\text{do}) \qquad \frac{\langle c,\sigma\rangle \to \sigma' \quad \langle b,\sigma'\rangle \to \mathbf{true}}{\langle \mathbf{do}\ c\ \mathbf{undoif}\ b,\sigma\rangle \to \sigma}\ (\text{undo})$$

2.

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

where $\mathscr{B}[\![b]\!]^* : \Sigma_\bot \to \mathbb{B}_\bot$ denotes the lifted version of the interpretation functions for boolean expressions (as c can diverge) and $t \to^* t_0,t_1$ denotes the lifted version of the conditional operator, such that it returns $\bot_{\Sigma_\bot}$ when t is $\bot_{\mathbb{B}_\bot}$.

3. First we extend the proof of completeness by rule induction. We recall that:

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

do: we assume that $\langle b, \sigma' \rangle \to \textbf{false}$ and $P(\langle c, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma'$. We need to prove that

$$P\left(\langle \textbf{do } c \textbf{ undoif } b, \sigma \rangle \to \sigma'\right) \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{do } c \textbf{ undoif } b]\!]\sigma = \sigma'$$

From $\langle b, \sigma' \rangle \to \textbf{false}$ it follows $\mathscr{B}[\![b]\!](\sigma') = \textbf{false}$. We have

$$\begin{aligned}
\mathscr{C}[\![\textbf{do } c \textbf{ undoif } b]\!]\sigma &\stackrel{\text{def}}{=} \mathscr{B}[\![b]\!]^*(\mathscr{C}[\![c]\!]\sigma) \to^* \sigma, \mathscr{C}[\![c]\!]\sigma \\
&= \mathscr{B}[\![b]\!]^*\sigma' \to^* \sigma, \sigma' \\
&= \mathscr{B}[\![b]\!]\sigma' \to^* \sigma, \sigma' \\
&= \textbf{false} \to^* \sigma, \sigma' \\
&= \textbf{false} \to \sigma, \sigma' \\
&= \sigma'.
\end{aligned}$$

undo: we assume that $\langle b, \sigma' \rangle \to \textbf{true}$ and $P(\langle c, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma'$. We need to prove that

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

From $\langle b, \sigma' \rangle \to \textbf{true}$ it follows $\mathscr{B}[\![b]\!](\sigma') = \textbf{true}$. We have

$$\begin{aligned}
\mathscr{C}[\![\textbf{do } c \textbf{ undoif } b]\!]\sigma &\stackrel{\text{def}}{=} \mathscr{B}[\![b]\!]^*(\mathscr{C}[\![c]\!]\sigma) \to^* \sigma, \mathscr{C}[\![c]\!]\sigma \\
&= \mathscr{B}[\![b]\!]^*\sigma' \to^* \sigma, \sigma' \\
&= \mathscr{B}[\![b]\!]\sigma' \to^* \sigma, \sigma' \\
&= \textbf{true} \to^* \sigma, \sigma' \\
&= \textbf{true} \to \sigma, \sigma' \\
&= \sigma.
\end{aligned}$$

Finally, we extend the proof of correctness by structural induction. We assume

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

and we want to prove that

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

Let us take σ and σ' such that $\mathscr{C}[\![\textbf{do } c \textbf{ undoif } b]\!]\sigma = \sigma'$. We need to prove that $\langle \textbf{do } c \textbf{ undoif } b, \sigma \rangle \to \sigma'$. Since $\mathscr{C}[\![\textbf{do } c \textbf{ undoif } b]\!]\sigma = \sigma'$ it must be $\mathscr{C}[\![c]\!]\sigma = \sigma''$ for some $\sigma'' \neq \bot_{\Sigma_\bot}$ and by inductive hypothesis $\langle c, \sigma \rangle \to \sigma''$. We distinguish two cases.

$\mathscr{B}[\![b]\!]\sigma'' = \textbf{false}$: then $\sigma' = \sigma''$ and $\langle b, \sigma'' \rangle \to \textbf{false}$. Since $\langle c, \sigma \rangle \to \sigma''$ we apply rule (do) to derive $\langle \textbf{do } c \textbf{ undoif } b, \sigma \rangle \to \sigma'' = \sigma'$.

$\mathscr{B}[\![b]\!]\,\sigma'' = \mathbf{true}$: then $\sigma' = \sigma$ and $\langle b, \sigma''\rangle \to \mathbf{true}$. Since $\langle c, \sigma\rangle \to \sigma''$ we apply rule (undo) to conclude that $\langle \mathbf{do}\ c\ \mathbf{undoif}\ b, \sigma\rangle \to \sigma$.

Problems of Chapter 7

7.2

$$\underset{\tau_2\to int}{f}\ .\ \lambda\ \underset{\tau*int}{x}\ .\ \mathbf{if}\ \underbrace{\mathbf{snd}(\ \underset{\tau*int}{x}\ )}_{int}\ \mathbf{then}\ \underset{int}{1}\ \mathbf{else}\ \underset{\tau_2\to int}{f}\ (\ \underbrace{\mathbf{fst}(x)}_{int\to\tau_1}\ ,\ (\ \underbrace{\mathbf{fst}(x)}_{\tau=int\to\tau_1}\ \underbrace{\mathbf{snd}(x)}_{int}\ )\ )}_{\tau_1}}_{\tau_2=(int\to\tau_1)*\tau_1}}_{int}}_{(\tau*int)\to int}$$

From which we must have $\tau_2 \to int = (\tau * int) \to int$, i.e., $\tau_2 = (\tau * int)$. But since $\tau_2 = (int \to \tau_1) * \tau_1$, it must be $\tau = (int \to \tau_1)$ and $int = \tau_1$. Summing up, we have $\tau_1 = int$, $\tau = int \to int$ and $\tau_2 = (int \to int) * int$ and the principal type of t is $((int \to int) * int) \to int$.

7.3

1. We let $\tau = int * (int * (int * int))$ be the type of a list of integers with three elements (the last element of type *int* is 0 and it marks the end of the list) and we define

$$t \stackrel{\text{def}}{=} \lambda\ \underset{\tau}{\ell}.\ \mathbf{fst}(\mathbf{snd}(\ \underbrace{\mathbf{snd}(\underset{\tau}{\ell})}_{\tau}\ ))}_{int*(int*int)}}_{int*int}}_{int}}_{\tau\to int}$$

Let $L = (n_1, (n_2, (n_3, 0))) : \tau$ be a generic list of integers with three elements. Now we check that $(t\ L) \to n_3$:

$$\begin{array}{rl}
(t\ L) \to c & \nwarrow\ t \to \lambda x.\, t',\quad t'[{}^{L}/_{x}] \to c \\
\nwarrow^{*}_{x=\ell,\ t'=\mathbf{fst}(\mathbf{snd}(\mathbf{snd}(\ell)))} & \mathbf{fst}(\mathbf{snd}(\mathbf{snd}(L))) \to c \\
& \nwarrow\ \mathbf{snd}(\mathbf{snd}(L)) \to (t_1,t_2),\quad t_1 \to c \\
& \nwarrow\ \mathbf{snd}(L) \to (t_3,t_4),\quad t_4 \to (t_1,t_2),\quad t_1 \to c \\
& \nwarrow\ L \to (t_5,t_6),\quad t_6 \to (t_3,t_4),\quad t_4 \to (t_1,t_2),\quad t_1 \to c \\
\nwarrow_{t_5=n_1,\ t_6=(n_2,(n_3,0))} & (n_2,(n_3,0)) \to (t_3,t_4),\quad t_4 \to (t_1,t_2),\quad t_1 \to c \\
\nwarrow_{t_3=n_2,\ t_4=(n_3,0)} & (n_3,0) \to (t_1,t_2),\quad t_1 \to c \\
\nwarrow_{t_1=n_3,\ t_2=0} & n_3 \to c \\
\nwarrow_{c=n_3} & \square
\end{array}$$

2. The answer is negative. In fact a generic list of k integers has a type that depends on the length of the list itself and we do not have polymorphic functions in HOFL. The natural candidate

$$t \stackrel{\text{def}}{=} \textbf{rec}\ f.\ \lambda x.\ \textbf{if}\ \textbf{snd}(x)\ \textbf{then}\ \textbf{fst}(x)\ \textbf{else}\ f\ (\textbf{snd}(x))$$

is not typable, in fact we have

$$\underset{int\to\tau}{f}\ .\ \lambda\ \underset{\tau*int}{x}\ .\ \textbf{if}\ \textbf{snd}(\underbrace{\underset{\tau*int}{x}}_{int})\ \textbf{then}\ \underbrace{\textbf{fst}(x)}_{\tau}\ \textbf{else}\ \underbrace{\underset{int\to\tau}{f}\ \underbrace{(\textbf{snd}(x))}_{int}}_{\tau}$$

From which we must have $int \to \tau = (\tau * int) \to \tau$, i.e., $int = (\tau * int)$, which is not possible.

7.4

1. We have:

$$t_1 \stackrel{\text{def}}{=} \lambda\ \underset{int}{x}\ .\ \lambda\ \underset{\tau_1}{y}\ .\ \underbrace{\underbrace{\underset{int}{x}\ \underset{int}{3}}_{int}}_{\tau_1\to int}$$

$$t_2 \stackrel{\text{def}}{=} \lambda\ \underset{int*\tau_2}{z}\ .\ \underbrace{\textbf{fst}(\underbrace{\underset{int*\tau_2}{z}}_{int*\tau_2})}_{int}\ \underbrace{+\ \underset{int}{3}}_{int}$$

2. Assume $\tau_1 = \tau_2 = \tau$ with $c : \tau$ in canonical form. We compute the canonical forms of $((t_1\ 1)\ c)$ and $(t_2\ (1,c))$ as follows:

$$\begin{aligned}
((t_1\ 1)\ c) \to c_1 &\quad \nwarrow (t_1\ 1) \to \lambda y'.\ t',\quad t'[^{c}/_{y'}] \to c_1\\
&\quad \nwarrow t_1 \to \lambda x'.\ t'',\quad t''[^{1}/_{x'}] \to \lambda y'.\ t',\quad t'[^{c}/_{y'}] \to c_1\\
&\quad \nwarrow_{x'=x,\ t''=\lambda y.\ x+3} \lambda y.\ 1+3 \to \lambda y'.\ t',\quad t'[^{c}/_{y'}] \to c_1\\
&\quad \nwarrow_{y'=y,\ t'=1+3} 1+3 \to c_1\\
&\quad \nwarrow_{c_1=n_1\underline{+}n_2} 1 \to n_1,\quad 3 \to n_2\\
&\quad \nwarrow^{*}_{n_1=1,\ n_2=3} \square
\end{aligned}$$

Thus $c_1 = n_1 \underline{+} n_2 = 1 \underline{+} 3 = 4$ is the canonical form of $((t_1\ 1)\ c)$.

$$\begin{aligned}
(t_2\ (1,c)) \to c_2 &\quad \nwarrow t_2 \to \lambda z'.\ t',\quad t'[^{(1,c)}/_{z'}] \to c_2\\
&\quad \nwarrow_{z'=z,\ t'=\textbf{fst}(z)+3} \textbf{fst}((1,c)) + 3 \to c_2\\
&\quad \nwarrow_{c_2=n_1\underline{+}n_2} \textbf{fst}((1,c)) \to n_1,\quad 3 \to n_2\\
&\quad \nwarrow (1,c) \to (t'',t'''),\quad t'' \to n_1,\quad 3 \to n_2\\
&\quad \nwarrow_{t''=1,\ t'''=c} 1 \to n_1,\quad 3 \to n_2\\
&\quad \nwarrow^{*}_{n_1=1,\ n_2=3} \square
\end{aligned}$$

Thus $c_2 = n_1 \underline{+} n_2 = 1 \underline{+} 3 = 4$ is the canonical form also of $(t_2\ (1,c))$.

7.5 We find the principal type of *map*:

$$map \stackrel{\text{def}}{=} \lambda \underset{\tau_1\to\tau}{f}.\ \lambda \underset{\tau_1*\tau_1}{x}.\ ((\underset{\tau_1\to\tau}{f}\ \underbrace{\mathbf{fst}(\underset{\tau_1*\tau_2}{x})}_{\tau_1}), (\underset{\tau_1\to\tau}{f}\ \underbrace{\mathbf{snd}(\underset{\tau_1*\tau_2}{x})}_{\tau_2=\tau_1}))$$
$$\underbrace{\qquad\qquad\qquad}_{\tau}\ \underbrace{\qquad\qquad\qquad}_{\tau}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\tau*\tau}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{(\tau_1*\tau_1)\to(\tau,\tau)}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{(\tau_1\to\tau)\to(\tau_1*\tau_1)\to(\tau,\tau)}$$

We now compute the canonical form of the term $((map\ t)\ (1,2))$ where $t \stackrel{\text{def}}{=} \lambda x.\ 2\times x$:

$$\begin{array}{rl}
((map\ t)\ (1,2)) \to c & \nwarrow\ (map\ t) \to \lambda y.t',\quad t'[{}^{(1,2)}/_{y}] \to c \\
& \nwarrow\ map \to \lambda g.t'',\quad t''[{}^{t}/_{g}] \to \lambda y.t',\quad t'[{}^{(1,2)}/_{y}] \to c \\
& \nwarrow_{g=f,\ t''=\ldots}\ \lambda x.\ ((t\ \mathbf{fst}(x)), (t\ \mathbf{snd}(x))) \to \lambda y.t',\quad t'[{}^{(1,2)}/_{y}] \to c \\
& \nwarrow_{y=x,\ t'=\ldots}\ ((t\ \mathbf{fst}((1,2))), (t\ \mathbf{snd}((1,2)))) \to c \\
\nwarrow_{c=((t\ \mathbf{fst}((1,2))),(t\ \mathbf{snd}((1,2))))}\ \square &
\end{array}$$

So the canonical form is $c = (((\lambda x.\ 2\times x)\ \mathbf{fst}((1,2))), ((\lambda x.\ 2\times x)\ \mathbf{snd}((1,2))))$.

Problems of Chapter 8

8.4 We prove the monotonicity of the lifting operator $(\cdot)^* : [D\to E] \to [D_\perp \to E]$. Let us take two continuous functions $f, g \in [D\to E]$ such that $f \sqsubseteq_{D\to E} g$. We want to prove that $f^* \sqsubseteq_{D_\perp\to E} g^*$. So we need to prove that for any $x \in D_\perp$ we have $f^*(x) \sqsubseteq_E g^*(x)$. We have two possibilities:

- if $x = \perp_{D_\perp}$, then $f^*(\perp_{D_\perp}) = \perp_E = g^*(\perp_{D_\perp})$;
- if $x = \lfloor d \rfloor$ for some $d \in D$, we have $f^*(\lfloor d \rfloor) = f(d) \sqsubseteq g(d) = g^*(\lfloor d \rfloor)$, because $f \sqsubseteq_{D\to E} g$ by hypothesis.

8.5 We prove that the function apply : $[D\to E]\times D \to E$ is monotone. Let us take two continuous functions $f_1, f_2 \in [D\to E]$ and two elements $d_1, d_2 \in D$ such that $(f_1, d_1) \sqsubseteq_{[D\to E]\times D} (f_2, d_2)$, we want to prove that apply $(f_1, d_1) \sqsubseteq_E$ apply (f_2, d_2). By definition of the cartesian product domain, $(f_1, d_1) \sqsubseteq_{[D\to E]\times D} (f_2, d_2)$ means that $f_1 \sqsubseteq_{[D\to E]} f_2$ and $d_1 \sqsubseteq_D d_2$. Then, we have:

$$\begin{array}{rcll}
\text{apply}\ (f_1, d_1) & = & f_1(d_1) & \text{(by definition of apply)} \\
& \sqsubseteq_E & f_1(d_2) & \text{(by monotonicity of } f_1\text{)} \\
& \sqsubseteq_E & f_2(d_2) & \text{(because } f_1 \sqsubseteq_{[D\to E]} f_2\text{)} \\
& = & \text{apply}\ (f_1, d_1) & \text{(by definition of apply).}
\end{array}$$

8.6 Let $\mathbf{F}_f = \{d \mid d = f(d)\} \subseteq D$ be the set of fixpoints of $f : D \to D$. It is immediate that $\mathbf{F}_f$ is a PO, because it is a subset of the partial order D from which it inherits

the order relation. It remains to be proved that it is complete. Take a chain $\{d_i\}_{i\in\mathbb{N}}$ in $\mathbf{F}_f$. Since $\mathbf{F}_f \subseteq D$ and D is a CPO, the chain $\{d_i\}_{i\in\mathbb{N}}$ has a limit $d = \bigsqcup_{i\in\mathbb{N}} d_i$ in D. We want to prove that $d \in \mathbf{F}_f$, i.e., that $d = f(d)$. We note that for any $i \in \mathbb{N}$ we have $d_i = f(d_i)$, because $d_i \in \mathbf{F}_f$. Since f is continuous, we have:

$$f(d) = f\left(\bigsqcup_{i\in\mathbb{N}} d_i\right) = \bigsqcup_{i\in\mathbb{N}} f(d_i) = \bigsqcup_{i\in\mathbb{N}} d_i = d.$$

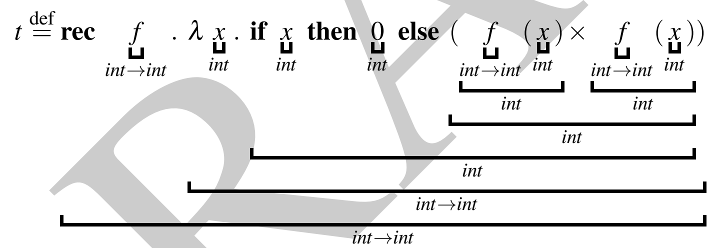
8.8 We divide the proof in two parts: first we show that $f \sqsubseteq g$ implies $f \preceq g$ and then that $f \preceq g$ implies $f \sqsubseteq g$.

For the first implication, suppose that $f \sqsubseteq g$. Taken any two elements $d_1, d_2 \in D$ such that $d_1 \sqsubseteq_D d_2$ we want to prove that $f(d_1) \sqsubseteq_E g(d_2)$. From the monotonicity of f we have $f(d_1) \sqsubseteq_E f(d_2)$ and by the hypothesis $f \sqsubseteq g$ it follows that $f(d_2) \sqsubseteq_E g(d_2)$; thus, $f(d_1) \sqsubseteq_E f(d_2) \sqsubseteq_E g(d_2)$.

For the second implication, suppose $f \preceq g$. We want to prove that for any element $d \in D$ we have $f(d) \sqsubseteq_E g(d)$. But this is immediate, because by reflexivity we have $d \sqsubseteq_D d$ and thus $f(d) \sqsubseteq_E g(d)$ by definition of $\preceq$.

Problems of Chapter 9

9.1 We show that t is typable:



So we conclude $t : int \to int$.

The canonical form is readily obtained by unfolding once the recursive definition:

$$t \to c \qquad \nwarrow \lambda x.\ \textbf{if}\ x\ \textbf{then}\ 0\ \textbf{else}\ (t(x) \times t(x)) \to c$$
$$\nwarrow_{c=\lambda x.\ \textbf{if}\ x\ \textbf{then}\ 0\ \textbf{else}\ (t(x)\times t(x))}\ \square$$

Finally, the denotational semantics is computed as follows:

$$\begin{aligned}
[\![t]\!]\rho &= \text{fix } \lambda d_f.\ [\![\lambda x.\ \textbf{if } x \textbf{ then } 0 \textbf{ else } (f(x)\times f(x))]\!]\rho[^{d_f}/_f] \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ [\![\textbf{if } x \textbf{ then } 0 \textbf{ else } (f(x)\times f(x))]\!] \underbrace{\rho[^{d_f}/_f, ^{d_x}/_x]}_{\rho'} \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ Cond([\![x]\!]\rho', [\![0]\!]\rho', [\![f(x)\times f(x)]\!]\rho') \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, [\![f(x)]\!]\rho' \underline{\times}_\perp [\![f(x)]\!]\rho') \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, (\textbf{let } \varphi \Leftarrow d_f.\ \varphi(d_x)) \underline{\times}_\perp (\textbf{let } \varphi \Leftarrow d_f.\ \varphi(d_x))) \rfloor
\end{aligned}$$

because

$$\begin{aligned}
[\![f(x)]\!]\rho' &= \textbf{let } \varphi \Leftarrow [\![f]\!]\rho'.\ \varphi([\![x]\!]\rho') \\
&= \textbf{let } \varphi \Leftarrow d_f.\ \varphi(d_x)
\end{aligned}$$

Let us compute the fixpoint by successive approximations:

$$\begin{aligned}
f_0 &= \perp_{(V_{int}\to int)_\perp} \\
f_1 &= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, (\textbf{let } \varphi \Leftarrow f_0.\ \varphi(d_x)) \underline{\times}_\perp (\textbf{let } \varphi \Leftarrow f_0.\ \varphi(d_x))) \rfloor \\
&= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, (\perp_{(V_{int})_\perp}) \underline{\times}_\perp (\perp_{(V_{int})_\perp})) \rfloor \\
&= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, \perp_{(V_{int})_\perp}) \rfloor \\
f_2 &= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, (\textbf{let } \varphi \Leftarrow f_1.\ \varphi(d_x)) \underline{\times}_\perp (\textbf{let } \varphi \Leftarrow f_1.\ \varphi(d_x))) \rfloor \\
&= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, (Cond(d_x, \lfloor 0\rfloor, \perp_{(V_{int})_\perp})) \underline{\times}_\perp (Cond(d_x, \lfloor 0\rfloor, \perp_{(V_{int})_\perp}))) \rfloor \\
&= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, (\perp_{(V_{int})_\perp}) \underline{\times}_\perp (\perp_{(V_{int})_\perp})) \rfloor \\
&= \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, \perp_{(V_{int})_\perp}) \rfloor \\
&= f_1
\end{aligned}$$

So we have reached the fixpoint and

$$[\![t]\!]\rho = \lfloor \lambda d_x.\ Cond(d_x, \lfloor 0\rfloor, \perp_{(V_{int})_\perp}) \rfloor$$

9.9

1. Assume $t_1 : \tau$. We have

$$t_2 \stackrel{\text{def}}{=} \underbrace{\lambda \underbrace{x}_{\tau_1}.\ \underbrace{(\underbrace{t_1}_{\tau_1\to\tau_2}\ \underbrace{x}_{\tau_1})}_{\tau_2}}_{\tau_1\to\tau_2}$$

   Unless $\tau = \tau_1 \to \tau_2$ the pre-term t_2 is not typable.
2. Let us compute the denotational semantics of t_2:

$$\begin{aligned}
[\![t_2]\!]\rho &= \left\lfloor \lambda d_x.\ [\![t_1\ x]\!]\rho[{}^{d_x}/_x] \right\rfloor \\
&= \left\lfloor \lambda d_x.\ \textbf{let}\ \varphi \Leftarrow [\![t_1]\!]\rho[{}^{d_x}/_x].\ \varphi([\![x]\!]\rho[{}^{d_x}/_x]) \right\rfloor \\
&= \left\lfloor \lambda d_x.\ \textbf{let}\ \varphi \Leftarrow [\![t_1]\!]\rho[{}^{d_x}/_x].\ \varphi(d_x) \right\rfloor
\end{aligned}$$

Suppose $x \notin \mathrm{fv}(t_1)$. Then we have $\forall y \in \mathrm{fv}(t_1).\ \rho(y) = \rho[{}^{d_x}/_x](y)$ and thus by Theorem 9.5 we have $[\![t_1]\!]\rho[{}^{d_x}/_x] = [\![t_1]\!]\rho$.
Now, if $[\![t_1]\!]\rho = \bot_{(V_\tau)_\perp}$, then $[\![t_2]\!]\rho = \left\lfloor \lambda d_x.\ \bot_{(V_{\tau_2})_\perp} \right\rfloor \neq [\![t_1]\!]\rho$.
Otherwise, it must be $[\![t_1]\!]\rho = \lfloor f \rfloor$ for some $f \in V_{\tau_1 \to \tau_2}$ and hence $[\![t_2]\!]\rho = \lfloor \lambda d_x.\ f\ d_x \rfloor = \lfloor f \rfloor = [\![t_1]\!]\rho$.

9.10

1. Let us compute the principal types for t_1 and t_2:

$$t_1 \stackrel{\text{def}}{=} \lambda \underset{\tau_1}{x}.\ \textbf{rec}\ \underset{int}{y}.\ \underbrace{\underbrace{\underset{int}{y} + \underset{int}{1}}_{int}}_{int}}_{\tau_1 \to int}$$

$$t_2 \stackrel{\text{def}}{=} \textbf{rec}\ \underset{\tau_2 \to int}{y}\ .\ \lambda \underset{\tau_2}{x}.\ \underbrace{\underbrace{\underbrace{\underbrace{(\underbrace{\underset{\tau_2 \to int}{y}\ \underset{\tau_2}{x}}_{int}) + \underset{int}{2}}_{int}}_{int}}_{\tau_2 \to int}}_{\tau_2 \to int}$$

Therefore t_1 and t_2 have the same type if and only if $\tau_1 = \tau_2$.

2. Let us compute the denotational semantics of t_1:

$$\begin{aligned}
[\![t_1]\!]\rho &= \lfloor \lambda d_x.\ [\![\textbf{rec}\ y.\ y+1]\!]\rho[{}^{d_x}/_x] \rfloor \\
&= \lfloor \lambda d_x.\ \mathrm{fix}\ \lambda d_y.\ [\![y+1]\!]\rho[{}^{d_x}/_x, {}^{d_y}/_y] \rfloor \\
&= \lfloor \lambda d_x.\ \mathrm{fix}\ \lambda d_y.\ [\![y]\!]\rho[{}^{d_x}/_x, {}^{d_y}/_y] \mathbin{\underline{+}_\perp} [\![1]\!]\rho[{}^{d_x}/_x, {}^{d_y}/_y] \rfloor \\
&= \lfloor \lambda d_x.\ \mathrm{fix}\ \lambda d_y.\ d_y \mathbin{\underline{+}_\perp} \lfloor 1 \rfloor \rfloor
\end{aligned}$$

We need to compute the fixpoint $\mathrm{fix}\ \lambda d_y.\ d_y \mathbin{\underline{+}_\perp} \lfloor 1 \rfloor$:

$$\begin{aligned}
d_0 &= \bot_{(V_{int})_\perp} \\
d_1 &= d_0 \mathbin{\underline{+}_\perp} \lfloor 1 \rfloor = \bot_{(V_{int})_\perp} = d_0
\end{aligned}$$

From which it follows

$$[\![t_1]\!]\rho = \lfloor \lambda d_x.\ \bot_{(V_{int})_\perp} \rfloor = \lfloor \bot_{(V_{\tau \to int})} \rfloor$$

Let us now turn the attention to t_2:

$$\begin{aligned}
[\![t_2]\!]\rho &= \text{fix } \lambda d_y.\ [\![\lambda x.\ (y\ x)+2]\!]\rho[{}^{d_y}/_{y}] \\
&= \text{fix } \lambda d_y.\ \lfloor \lambda d_x.\ [\![(y\ x)+2]\!]\underbrace{\rho[{}^{d_y}/_{y}, {}^{d_x}/_{x}]}_{\rho'}\rfloor \\
&= \text{fix } \lambda d_y.\ \lfloor \lambda d_x.\ [\![y\ x]\!]\rho' \pm_\perp [\![2]\!]\rho'\rfloor \\
&= \text{fix } \lambda d_y.\ \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow [\![y]\!]\rho'.\ \varphi([\![x]\!]\rho')) \pm_\perp \lfloor 2\rfloor\rfloor \\
&= \text{fix } \lambda d_y.\ \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow d_y.\ \varphi(d_x)) \pm_\perp \lfloor 2\rfloor\rfloor
\end{aligned}$$

Let us compute the fixpoint:

$$\begin{aligned}
f_0 &= \perp_{(V_{\tau\to int})_\perp} \\
f_1 &= \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow f_0.\ \varphi(d_x)) \pm_\perp \lfloor 2\rfloor\rfloor \\
&= \lfloor \lambda d_x.\ (\perp_{(V_{int})_\perp}) \pm_\perp \lfloor 2\rfloor\rfloor \\
&= \lfloor \lambda d_x.\ \perp_{(V_{int})_\perp}\rfloor \\
&= \lfloor \perp_{(V_{\tau\to int})}\rfloor \\
f_2 &= \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow f_1.\ \varphi(d_x)) \pm_\perp \lfloor 2\rfloor\rfloor \\
&= \lfloor \lambda d_x.\ (\perp_{(V_{\tau\to int})}(d_x)) \pm_\perp \lfloor 2\rfloor\rfloor \\
&= \lfloor \lambda d_x.\ (\perp_{(V_{int})_\perp}) \pm_\perp \lfloor 2\rfloor\rfloor \\
&= \lfloor \lambda d_x.\ \perp_{(V_{int})_\perp}\rfloor \\
&= \lfloor \perp_{(V_{\tau\to int})}\rfloor \\
&= f_1
\end{aligned}$$

So we have computed the fixpoint and got

$$[\![t_2]\!]\rho = \lfloor \perp_{(V_{\tau\to int})}\rfloor = [\![t_1]\!]\rho.$$

9.15 Let us try to change the denotational semantics of the conditional construct of HOFL by defining:

$$[\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho \stackrel{\text{def}}{=} \mathit{Cond}'([\![t]\!]\rho, [\![t_0]\!]\rho, [\![t_1]\!]\rho)$$

where

$$\mathit{Cond}'(x, d_0, d_1) = \begin{cases} d_0 & \text{if } x = \lfloor n\rfloor \text{ for some } n \in \mathbb{Z} \\ d_1 & \text{if } x = \perp_{(V_{int})_\perp}. \end{cases}$$

The problem is that the newly defined operation Cond' is not monotone (and thus not continuous)! To see this, remind that $\perp_{(V_{int})_\perp} \sqsubseteq \lfloor 1\rfloor$ and take any d_0, d_1: we should have $\mathit{Cond}'(\perp_{(V_{int})_\perp}, d_0, d_1) \sqsubseteq \mathit{Cond}'(\lfloor 1\rfloor, d_0, d_1)$. However, if we take d_0, d_1 such that $d_1 \not\sqsubseteq d_0$ it follows that:

$$\mathit{Cond}'(\perp_{(V_{int})_\perp}, d_0, d_1) = d_1 \not\sqsubseteq d_0 = \mathit{Cond}'(\lfloor 1\rfloor, d_0, d_1)$$

For a concrete example, take $d_1 = \lfloor 1\rfloor$ and $d_0 = \lfloor 0\rfloor$.

At the level of HOFL syntax, the previous cases arise when considering, e.g., the terms $t_1 \stackrel{\text{def}}{=} \textbf{if } (\textbf{rec } x.\ x) \textbf{ then } 0 \textbf{ else } 1$ and $t_2 \stackrel{\text{def}}{=} \textbf{if } 1 \textbf{ then } 0 \textbf{ else } 1$, as

$$[\![t_1]\!]\rho = \lfloor 1 \rfloor \not\sqsubseteq \lfloor 0 \rfloor = [\![t_2]\!]\rho$$

As a consequence, typable terms such as

$$t \stackrel{\text{def}}{=} \lambda x.\ \textbf{if } x \textbf{ then } 0 \textbf{ else } 1 \ :\ \mathit{int} \to \mathit{int}$$

would not be assigned a semantics in $(V_{\mathit{int}\to\mathit{int}})_\perp$ because the function $[\![t]\!]\rho$ would not be continuous.

Problems of Chapter 10

10.1 Let us check the type of t_1 and t_2:

$$\underbrace{\textbf{rec } \underbrace{f}_{\tau_2\to\tau_1}.\ \lambda \underbrace{x}_{\tau_2}.\ ((\lambda \underbrace{y}_{\tau_1}.\ \underbrace{1}_{\mathit{int}})\ (\underbrace{f}_{\tau_2\to\tau_1}\ \underbrace{x}_{\tau_2}))}_{\tau_1\to\mathit{int}\quad\tau_1}}_{\mathit{int}}}_{\tau_2\to\mathit{int}}}_{\tau_2\to\tau_1=\tau_2\to\mathit{int}} \qquad \underbrace{\lambda \underbrace{x}_{\tau}.\ \underbrace{1}_{\mathit{int}}}_{\tau\to\mathit{int}}$$

So it must be $\tau_1 = \mathit{int}$ and the terms have the same type if $\tau_2 = \tau$.

The denotational semantics of t_1 requires the computation of the fixpoint:

$$\begin{aligned}
[\![t_1]\!]\rho &= \text{fix } \lambda d_f.\ [\![\lambda x.\ ((\lambda y.\ 1)\ (f\ x))]\!]\rho[{}^{d_f}/_f] \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ [\![((\lambda y.\ 1)\ (f\ x))]\!]\underbrace{\rho[{}^{d_f}/_f, {}^{d_x}/_x]}_{\rho'} \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow [\![\lambda y.\ 1]\!]\rho'.\ \varphi([\![f\ x]\!]\rho')) \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ (\textbf{let } \varphi \Leftarrow \lfloor \lambda d_y.\ \lfloor 1 \rfloor \rfloor.\ (\varphi(\textbf{let } \varphi' \Leftarrow d_f.\ \varphi'(d_x)))) \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ ((\lambda d_y.\ \lfloor 1 \rfloor)(\textbf{let } \varphi' \Leftarrow d_f.\ \varphi'(d_x))) \rfloor \\
&= \text{fix } \lambda d_f.\ \lfloor \lambda d_x.\ \lfloor 1 \rfloor \rfloor \\
f_0 &= \perp_{(V_{\tau\to\mathit{int}})_\perp} \\
f_1 &= \lfloor \lambda d_x.\ \lfloor 1 \rfloor \rfloor
\end{aligned}$$

We can stop the calculation of the fixpoint, as we have reached a maximal element. Thus $[\![t_1]\!]\rho = \lfloor \lambda d_x.\ \lfloor 1 \rfloor \rfloor$. For t_2 we have directly:

$$\begin{aligned}\llbracket t_2 \rrbracket \rho &= \lfloor \lambda d_x.\ \llbracket 1 \rrbracket \rho[^{d_x}/_x] \rfloor \\ &= \lfloor \lambda d_x.\ \lfloor 1 \rfloor \rfloor \\ &= \llbracket t_1 \rrbracket \rho\end{aligned}$$

To show that the canonical forms are different, we note that t_2 is already in canonical form, while for t_1 we have:

$$t_1 \to c_1 \qquad \nwarrow \lambda x.\ ((\lambda y.\ 1)\ (t_1\ x)) \to c_1$$
$$\nwarrow_{c_1 = \lambda x.\ ((\lambda y.\ 1)\ (t_1\ x))} \square$$

10.2

1. We compute the denotational semantics of *map* and of $t \stackrel{\text{def}}{=} (map\ \lambda z.\ z)$:

$$\begin{aligned}\llbracket map \rrbracket \rho &= \lfloor \lambda d_f.\ \llbracket \lambda x.\ ((f\ \mathbf{fst}(x)), (f\ \mathbf{snd}(x))) \rrbracket \rho[^{d_f}/_f] \rfloor \\ &= \lfloor \lambda d_f.\ \lfloor \lambda d_x.\ \llbracket ((f\ \mathbf{fst}(x)), (f\ \mathbf{snd}(x))) \rrbracket \underbrace{\rho[^{d_f}/_f, {}^{d_x}/_x]}_{\rho'} \rfloor \rfloor \\ &= \lfloor \lambda d_f.\ \lfloor \lambda d_x.\ \lfloor (\llbracket (f\ \mathbf{fst}(x)) \rrbracket \rho', \llbracket (f\ \mathbf{snd}(x)) \rrbracket \rho') \rfloor \rfloor \rfloor \\ &= \lfloor \lambda d_f.\ \lfloor \lambda d_x.\ \lfloor ((\mathbf{let}\ \varphi_1 \Leftarrow \llbracket f \rrbracket \rho'.\ \varphi_1(\llbracket \mathbf{fst}(x) \rrbracket \rho')), \\ &\qquad\qquad (\mathbf{let}\ \varphi_2 \Leftarrow \llbracket f \rrbracket \rho'.\ \varphi_2(\llbracket \mathbf{snd}(x) \rrbracket \rho'))) \rfloor \rfloor \rfloor \\ &= \lfloor \lambda d_f.\ \lfloor \lambda d_x.\ \lfloor ((\mathbf{let}\ \varphi_1 \Leftarrow d_f.\ \varphi_1(\mathbf{let}\ d_1 \Leftarrow \llbracket x \rrbracket \rho'.\ \pi_1\ d_1)), \\ &\qquad\qquad (\mathbf{let}\ \varphi_2 \Leftarrow d_f.\ \varphi_2(\mathbf{let}\ d_2 \Leftarrow \llbracket x \rrbracket \rho'.\ \pi_2\ d_2))) \rfloor \rfloor \rfloor \\ &= \lfloor \lambda d_f.\ \lfloor \lambda d_x.\ \lfloor ((\mathbf{let}\ \varphi_1 \Leftarrow d_f.\ \varphi_1(\mathbf{let}\ d_1 \Leftarrow d_x.\ \pi_1\ d_1)), \\ &\qquad\qquad (\mathbf{let}\ \varphi_2 \Leftarrow d_f.\ \varphi_2(\mathbf{let}\ d_2 \Leftarrow d_x.\ \pi_2\ d_2))) \rfloor \rfloor \rfloor \\ \llbracket t \rrbracket \rho &= \mathbf{let}\ \varphi \Leftarrow \llbracket map \rrbracket \rho.\ \varphi(\llbracket \lambda z.\ z \rrbracket \rho) \\ &= \mathbf{let}\ \varphi \Leftarrow \llbracket map \rrbracket \rho.\ \varphi(\lfloor \lambda d_z.\ \llbracket z \rrbracket \rho[^{d_z}/_z] \rfloor) \\ &= \mathbf{let}\ \varphi \Leftarrow \llbracket map \rrbracket \rho.\ \varphi(\lfloor \lambda d_z.\ d_z \rfloor) \\ &= \lfloor \lambda d_x.\ \lfloor ((\mathbf{let}\ \varphi_1 \Leftarrow \lfloor \lambda d_z.\ d_z \rfloor.\ \varphi_1(\mathbf{let}\ d_1 \Leftarrow d_x.\ \pi_1\ d_1)), \\ &\qquad\qquad (\mathbf{let}\ \varphi_2 \Leftarrow \lfloor \lambda d_z.\ d_z \rfloor.\ \varphi_2(\mathbf{let}\ d_2 \Leftarrow d_x.\ \pi_2\ d_2))) \rfloor \rfloor \\ &= \lfloor \lambda d_x.\ \lfloor (((\lambda d_z.\ d_z)(\mathbf{let}\ d_1 \Leftarrow d_x.\ \pi_1\ d_1)), \\ &\qquad\qquad ((\lambda d_z.\ d_z)(\mathbf{let}\ d_2 \Leftarrow d_x.\ \pi_2\ d_2))) \rfloor \rfloor \\ &= \lfloor \lambda d_x.\ \lfloor ((\mathbf{let}\ d_1 \Leftarrow d_x.\ \pi_1\ d_1), (\mathbf{let}\ d_2 \Leftarrow d_x.\ \pi_2\ d_2)) \rfloor \rfloor\end{aligned}$$

2. It suffices to take $t_1 \stackrel{\text{def}}{=} 1+1$ and $t_2 \stackrel{\text{def}}{=} 2$. It can be readily checked that

$$\llbracket (t_1, t_2) \rrbracket \rho = \lfloor (\lfloor 2 \rfloor, \lfloor 2 \rfloor) \rfloor = \llbracket (t_2, t_1) \rrbracket \rho.$$

Letting $t_0 \stackrel{\text{def}}{=} (map\ \lambda z.\ z)$, we have that the terms $(t_0\ (t_1, t_2))$ and $(t_0\ (t_2, t_1))$ have the same denotational semantics

$$\begin{aligned}
[\![t_0\ (t_1,t_2)]\!]\rho &= \textbf{let}\ \varphi \Leftarrow [\![t_0]\!].\ \varphi([\![(t_1,t_2)]\!]\rho)\\
&= \textbf{let}\ \varphi \Leftarrow [\![t_0]\!].\ \varphi(\lfloor(\lfloor 2\rfloor,\lfloor 2\rfloor)\rfloor)\\
&= \lfloor((\textbf{let}\ d_1 \Leftarrow \lfloor(\lfloor 2\rfloor,\lfloor 2\rfloor)\rfloor.\ \pi_1\ d_1),(\textbf{let}\ d_2 \Leftarrow \lfloor(\lfloor 2\rfloor,\lfloor 2\rfloor)\rfloor.\ \pi_2\ d_2))\rfloor\\
&= \lfloor((\pi_1\ (\lfloor 2\rfloor,\lfloor 2\rfloor)),(\pi_2\ (\lfloor 2\rfloor,\lfloor 2\rfloor)))\rfloor\\
&= \lfloor(\lfloor 2\rfloor,\lfloor 2\rfloor)\rfloor\\
[\![t_0\ (t_2,t_1)]\!]\rho &= \textbf{let}\ \varphi \Leftarrow [\![t_0]\!].\ \varphi([\![(t_2,t_1)]\!]\rho)\\
&= \textbf{let}\ \varphi \Leftarrow [\![t_0]\!].\ \varphi(\lfloor(\lfloor 2\rfloor,\lfloor 2\rfloor)\rfloor)\\
&= [\![t_0\ (t_1,t_2)]\!]\rho
\end{aligned}$$

The same result can be obtained by observing that $(t_0\ (t_1,t_2)) = (t_0\ y)[^{(t_1,t_2)}/_y]$ and $(t_0\ (t_2,t_1)) = (t_0\ y)[^{(t_2,t_1)}/_y]$. Then, by compositionality we have:

$$\begin{aligned}
[\![t_0\ (t_1,t_2)]\!]\rho &= \left[\!\!\left[(t_0\ y)[^{(t_1,t_2)}/_y]\right]\!\!\right]\rho\\
&= [\![(t_0\ y)]\!]\rho[^{[\![(t_1,t_2)]\!]\rho}/_y]\\
&= [\![(t_0\ y)]\!]\rho[^{\lfloor(\lfloor 2\rfloor,\lfloor 2\rfloor)\rfloor}/_y]\\
&= [\![(t_0\ y)]\!]\rho[^{[\![(t_2,t_1)]\!]\rho}/_y]\\
&= \left[\!\!\left[(t_0\ y)[^{(t_2,t_1)}/_y]\right]\!\!\right]\rho\\
&= [\![t_0\ (t_2,t_1)]\!]\rho.
\end{aligned}$$

We conclude by showing that the terms $(t_0\ (t_1,t_2))$ and $(t_0\ (t_2,t_1))$ have different canonical forms:

$$\begin{array}{ll}
(t_0\ (t_1,t_2)) \to c_1 & \nwarrow\ t_0 \to \lambda x'.t,\quad t[^{(t_1,t_2)}/_{x'}] \to c_1\\
& \nwarrow\ map \to \lambda f'.t',\quad t'[^{\lambda z.\ z}/_{f'}] \to \lambda x'.t,\quad t[^{(t_1,t_2)}/_{x'}] \to c_1\\
& \nwarrow_{f'=f,t'=\ldots}\ \lambda x.\ (((\lambda z.\ z)\ \textbf{fst}(x)),((\lambda z.\ z)\ \textbf{snd}(x))) \to \lambda x'.t,\\
& \qquad t[^{(t_1,t_2)}/_{x'}] \to c_1\\
& \nwarrow_{x'=x,t=\ldots}\ (((\lambda z.\ z)\ \textbf{fst}((t_1,t_2))),((\lambda z.\ z)\ \textbf{snd}((t_1,t_2)))) \to c_1\\
& \nwarrow_{c_1=(\ldots,\ldots)}\ \square\\
(t_0\ (t_2,t_1)) \to c_2 & \nwarrow^{*}\ (((\lambda z.\ z)\ \textbf{fst}((t_2,t_1))),((\lambda z.\ z)\ \textbf{snd}((t_2,t_1)))) \to c_2\\
& \nwarrow_{c_2=(\ldots,\ldots)}\ \square
\end{array}$$

10.11

1. We extend the proof of correctness to take into account the new rules. We recall that the predicate to be proved is

$$P(t \to c) \stackrel{\text{def}}{=} \forall\rho.\ [\![t]\!]\rho = [\![c]\!]\rho.$$

For the rule

$$\frac{t \to 0 \quad t_0 \to c_0 \quad t_1 \to c_1}{\textbf{if}\ t\ \textbf{then}\ t_0\ \textbf{else}\ t_1 \to c_0}$$

we can assume

$$P(t \to 0) \stackrel{\text{def}}{=} \forall \rho.\ [\![t]\!]\rho = [\![0]\!]\rho = \lfloor 0 \rfloor$$
$$P(t_0 \to c_0) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_0]\!]\rho = [\![c_0]\!]\rho$$
$$P(t_1 \to c_1) \stackrel{\text{def}}{=} \forall \rho.\ [\![t_1]\!]\rho = [\![c_1]\!]\rho$$

and we want to prove

$$P(\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_0) \stackrel{\text{def}}{=} \forall \rho.\ [\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho = [\![c_0]\!]\rho.$$

We have:

$$\begin{array}{lll} [\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho & = \mathit{Cond}([\![t]\!]\rho, [\![t_0]\!]\rho, [\![t_1]\!]\rho) & \text{(by definition)} \\ & = \mathit{Cond}(\lfloor 0 \rfloor, [\![t_0]\!]\rho, [\![t_1]\!]\rho) & \text{(by inductive hypothesis)} \\ & = [\![t_0]\!]\rho & \text{(by definition of } \mathit{Cond}) \\ & = [\![c_0]\!]\rho & \text{(by inductive hypothesis)} \end{array}$$

For the other rule the proof is analogous and thus omitted.

2. As a counterexample, we can take

$$t \stackrel{\text{def}}{=} \textbf{if } 0 \textbf{ then } 1 \textbf{ else rec } x.\ x.$$

In fact, its denotational semantics is

$$[\![t]\!]\rho = \mathit{Cond}([\![0]\!]\rho, [\![1]\!]\rho, [\![\textbf{rec } x.\ x]\!]\rho) = \mathit{Cond}(\lfloor 0 \rfloor, \lfloor 1 \rfloor, \bot_{(V_{int})_\bot} = \lfloor 1 \rfloor$$

and therefore $t \Downarrow$. Vice versa $t \uparrow$, as:

$$\begin{array}{rcl} t \to c & \nwarrow & 0 \to 0, \quad 1 \to c, \quad \textbf{rec } x.\ x \to c' \\ & \nwarrow & 1 \to c, \quad \textbf{rec } x.\ x \to c' \\ & \nwarrow_{c=1} & \textbf{rec } x.\ x \to c' \\ & \nwarrow & x[^{\textbf{rec } x.\ x}/_x] \to c' \\ & = & \textbf{rec } x.\ x \to c' \\ & \nwarrow & \cdots \end{array}$$

10.13 According to the operational semantics we have:

$$\begin{array}{rcl} \textbf{rec } x.\ t \to c & \nwarrow & t[^{\textbf{rec } x.\ t}/_x] \to c \\ & = & t \to c \end{array}$$

because by hypothesis $x \notin \mathrm{fv}(t)$. So we conclude that either both terms have the same canonical form or they do not have any canonical form.

According to the denotational semantics we have

$$\begin{array}{rcl} [\![\textbf{rec } x.\ t]\!]\rho & = & \mathrm{fix}\ \lambda d_x.\ [\![t]\!]\rho[^{d_x}/_x] \\ & = & \mathrm{fix}\ \lambda d_x.\ [\![t]\!]\rho \end{array}$$

When we compute the fixpoint, assuming $t : \tau$, we get:

$$\begin{aligned} d_0 &= \bot_{(V_\tau)_\bot} \\ d_1 &= (\lambda d_x.\ [\![t]\!]\rho)\ d_0 = [\![t]\!]\rho[{}^{d_0}/_{d_x}] = [\![t]\!]\rho \\ d_2 &= (\lambda d_x.\ [\![t]\!]\rho)\ d_1 = [\![t]\!]\rho[{}^{d_1}/_{d_x}] = [\![t]\!]\rho = d_1 \end{aligned}$$

So we have reached the fixpoint and have $[\![\mathbf{rec}\ x.\ t]\!]\rho = [\![t]\!]\rho$.

Alternatively, we could have computed the semantics as follows:

$$\begin{aligned} [\![\mathbf{rec}\ x.\ t]\!]\rho &= [\![t]\!]\rho[{}^{[\![\mathbf{rec}\ x.\ t]\!]\rho}/_x] && \text{(by definition)} \\ &= [\![t[{}^{\mathbf{rec}\ x.\ t}/_x]]\!]\rho && \text{(by Substituition Lemma)} \\ &= [\![t]\!]\rho && \text{(because } x \notin \mathrm{fv}(t)\text{)} \end{aligned}$$

10.14

1. By Theorem 10.1 (Correctness) we have $[\![t_0]\!]\rho = [\![c_0]\!]\rho$. Hence:
$$[\![t_1'[{}^{t_0}/_x]]\!]\rho = [\![t_1']\!]\rho[{}^{[\![t_0]\!]\rho}/_x] = [\![t_1']\!]\rho[{}^{[\![c_0]\!]\rho}/_x] = [\![t_1'[{}^{c_0}/_x]]\!]\rho.$$
2. If $[\![t_1'[{}^{t_0}/_x]]\!]\rho = [\![t_1'[{}^{c_0}/_x]]\!]\rho = \bot_{\mathbb{Z}_\bot}$, then we have that $t_1'[{}^{t_0}/_x] \uparrow$ and $t_1'[{}^{c_0}/_x] \uparrow$, because the operational semantics agrees on convergence with the denotational semantics.
   If $[\![t_1'[{}^{t_0}/_x]]\!]\rho = [\![t_1'[{}^{c_0}/_x]]\!]\rho \neq \bot_{\mathbb{Z}_\bot}$, it exists $n \in \mathbb{Z}$ such that $[\![t_1'[{}^{t_0}/_x]]\!]\rho = [\![t_1'[{}^{c_0}/_x]]\!]\rho = \lfloor n \rfloor$. Then, since $t_1'[{}^{t_0}/_x]$ and $t_1'[{}^{c_0}/_x]$ are closed, by Theorem 10.4, we have $t_1'[{}^{t_0}/_x] \to n$ and $t_1'[{}^{c_0}/_x] \to n$.
3. Suppose that $(t_1\ t_0) \to c$ in the eager semantics. Then it must be the case that $t_1 \to \lambda x.\ t_1'$ for some suitable x and t_1', and that $t_1'[c_0/x] \to c$ (we know that $t_0 \to c_0$ by initial hypothesis). Since $c : \mathit{int}$ it must be $c = n$ for some integer n. Then, by the previous point we know that $t_1'[{}^{t_0}/_x] \to n$ because $t_1'[{}^{c_0}/_x] \to n$. We conclude that $(t_1\ t_0) \to c$ in the lazy semantics by exploiting the (lazy) rule for function application.
4. As a simple counterexample, we can take, e.g., $t_1 = \lambda x.\ ((\lambda y.\ x)\ (\mathbf{rec}\ z.\ z))$ with $y \notin \mathrm{fv}(t_0)$. In fact, in the lazy semantics, we have:
$$\begin{aligned} (t_1\ t_0) \to c &\quad\nwarrow t_1 \to \lambda x'.\ t_1',\quad t_1'[{}^{t_0}/_{x'}] \to c \\ &\quad\nwarrow_{x'=x,\ t_1'=((\lambda y.\ x)\ (\mathbf{rec}\ z.\ z))} ((\lambda y.\ t_0)\ (\mathbf{rec}\ z.\ z)) \to c \\ &\quad\nwarrow (\lambda y.\ t_0) \to \lambda y'.\ t_2,\quad t_2[{}^{(\mathbf{rec}\ z.\ z)}/_{y'}] \to c \\ &\quad\nwarrow_{y'=y,\ t_2=t_0} t_0[{}^{(\mathbf{rec}\ z.\ z)}/_y] \to c \\ &\quad= t_0 \to c \\ &\quad\nwarrow_{c=c_0} \square \end{aligned}$$
   Whereas in the eager semantics we have:

$$
\begin{array}{rl}
(t_1\ t_0) \to c & \nwarrow\ t_1 \to \lambda x'.\, t_1',\quad t_0 \to c',\quad t_1'[{}^{c'}/_{x'}] \to c \\
\nwarrow_{x'=x,\ t_1'=((\lambda y.\ x)\ (\mathbf{rec}\ z.\ z))} & t_0 \to c',\quad ((\lambda y.\ c')\ (\mathbf{rec}\ z.\ z)) \to c \\
\nwarrow_{c'=c_0} & ((\lambda y.\ c_0)\ (\mathbf{rec}\ z.\ z)) \to c \\
\nwarrow & (\lambda y.\ c_0) \to \lambda y'.\, t_2,\quad \mathbf{rec}\ z.\ z \to c'',\quad t_2[{}^{c''}/_{y'}] \to c \\
\nwarrow_{y'=y,\ t_2=t_0} & \mathbf{rec}\ z.\ z \to c'',\quad t_0[{}^{c''}/_{y}] \to c \\
\nwarrow & \mathbf{rec}\ z.\ z \to c'',\quad t_0[{}^{c''}/_{y}] \to c \\
\nwarrow & \ldots
\end{array}
$$

5. Let $x, t_0 : \tau_0$ and $y : \tau$ and assume $t_0 \neq c_0$. As a last counterexample, let us take $t_1' = \lambda y.\ x$ (and $t_1 = \lambda x.\ t_1'$), with $t_1' : \tau \to \tau_0$. We have immediately that $t_1'[{}^{t_0}/_{x}] = \lambda y.\ t_0$ and $t_1'[{}^{c_0}/_{x}] = \lambda y.\ c_0$ are already in canonical form and they are different. Moreover, $(t_1\ t_0) \to \lambda y.\ t_0$ in the lazy semantics, but $(t_1\ t_0) \to \lambda y.\ c_0$ in the eager semantics.