The Formal Semantics of Programming Languages

Foundations of Computing
Michael Garey and Albert Meyer, editors

Complexity Issues in VLSI: Optimal Layouts for the Shuffle-Exchange Graph and Other Networks, Frank Thomson Leighton, 1983

Equational Logic as a Programming Language, Michael J. O'Donnell, 1985

General Theory of Deductive Systems and Its Applications, S. Yu Maslov, 1987

Resource Allocation Problems: Algorithmic Approaches, Toshihide Ibaraki and Naoki Katoh, 1988

Algebraic Theory of Processes, Matthew Hennessy, 1988

PX: A Computational Logic, Susumu Hayashi and Hiroshi Nakano, 1989

The Stable Marriage Problem: Structure and Algorithms, Dan Gusfield and Robert Irving, 1989

Realistic Compiler Generation, Peter Lee, 1989

*Single-Layer Wire Routing and Compaction,*F. Miller Maley, 1990

Basic Category Theory for Computer Scientists, Benjamin C. Pierce, 1991

Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist, Andrea Asperti and Giuseppe Longo, 1991

Semantics of Programming Languages: Structures and Techniques, Carl A. Gunter, 1992

The Formal Semantics of Programming Languages: An Introduction, Glynn Winskel, 1993

The Formal Semantics of Programming Languages
An Introduction

Glynn Winskel

The MIT Press
Cambridge, Massachusetts
London, England

Third printing, 1996
©1993 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

This book was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Winskel, G. (Glynn)

The formal semantics of programming languages : an introduction
Glynn Winskel.

p. cm. – (Foundations of computing)
Includes bibliographical references and index.
ISBN 0-262-23169-7
1. Programming languages (Electronic computers)–Semantics.
I. Title. II. Series.
QA76.7.W555 1993
005.13'1–dc20 **92-36718**
CIP

To Kirsten, Sofie and Stine

Contents

Series foreword

Theoretical computer science has now undergone several decades of development. The "classical" topics of automata theory, formal languages, and computational complexity have become firmly established, and their importance to other theoretical work and to practice is widely recognized. Stimulated by technological advances, theoreticians have been rapidly expanding the areas under study, and the time delay between theoretical progress and its practical impact has been decreasing dramatically. Much publicity has been given recently to breakthroughs in cryptography and linear programming, and steady progress is being made on programming language semantics, computational geometry, and efficient data structures. Newer, more speculative, areas of study include relational databases, VLSI theory, and parallel and distributed computation. As this list of topics continues expanding, it is becoming more and more difficult to stay abreast of the progress that is being made and increasingly important that the most significant work be distilled and communicated in a manner that will facilitate further research and application of this work. By publishing comprehensive books and specialized monographs on the theoretical aspects of computer science, the series on Foundations of Computing provides a forum in which important research topics can be presented in their entirety and placed in perspective for researchers, students, and practitioners alike.

Michael R. Garey
Albert R. Meyer

Preface

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 simply the activity of trying to define the meaning of program constructions precisely can reveal all kinds of subtleties of which it is important to be aware. This book introduces the mathematics, techniques and concepts on which formal semantics rests.

For historical reasons the semantics of programming languages is often viewed as consisting of three strands:

> *Operational semantics* describes the meaning of a programming language by specifying how it executes on an abstract machine. We concentrate on the method advocated by Gordon Plotkin in his lectures at Aarhus on "structural operational semantics" in which evaluation and execution relations are specified by rules in a way directed by the syntax.
>
> *Denotational semantics* is a technique for defining the meaning of programming languages pioneered by Christopher Strachey and provided with a mathematical foundation by Dana Scott. At one time called "mathematical semantics," it uses the more abstract mathematical concepts of complete partial orders, continuous functions and least fixed points.
>
> *Axiomatic semantics* tries to fix the meaning of a programming contruct by giving proof rules for it within a program logic. The chief names associated with this approach are that of R.W.Floyd and C.A.R.Hoare. Thus axiomatic semantics emphasises proof of correctness right from the start.

It would however be wrong to view these three styles as in opposition to each other. They each have their uses. A clear operational semantics is very helpful in implementation. Axiomatic semantics for special kinds of languages can give strikingly elegant proof systems, useful in developing as well as verifying programs. Denotational semantics provides the deepest and most widely applicable techniques, underpinned by a rich mathematical theory. Indeed, the different styles of semantics are highly dependent on eachother. For example, showing that the proof rules of an axiomatic semantics are correct relies on an underlying denotational or operational semantics. To show an implementation correct, as judged against a denotational semantics, requires a proof that the operational and denotational semantics agree. And, in arguing about an operational semantics it can be an enormous help to use a denotational semantics, which often has the advantage of abstracting away from unimportant, implementation details, as well as providing higher-level concepts with which to understand computational behaviour. Research of the last

few years promises a unification of the different approaches, an approach in which we can hope to see denotational, operational and logics of programs developed hand-in-hand. An aim of this book has been to show how operational and denotational semantics fit together.

The techniques used in semantics lean heavily on mathematical logic. They are not always easily accessible to a student of computer science or mathematics, without a good background in logic. There is an attempt here to present them in a thorough and yet as elementary a way as possible. For instance, a presentation of operational semantics leads to a treatment of inductive definitions, and techniques for reasoning about operational semantics, and this in turn places us in a good position to take the step of abstraction to complete partial orders and continuous functions—the foundation of denotational semantics. It is hoped that this passage from finitary rules of the operational semantics, to continuous operators on sets, to continuous functions is also a help in understanding why continuity is to be expected of computable functions. Various induction principles are treated, including a general version of well-founded recursion, which is important for defining functions on a set with a well-founded relation. In the more advanced work on languages with recursive types the use of information systems not only provides an elementary way of solving recursive domain equations, but also yields techniques for relating operational and denotational semantics.

Book description: This is a book based on lectures given at Cambridge and Aarhus Universities. It is introductory and is primarily addressed to undergraduate and graduate students in Computer Science and Mathematics beginning a study of the methods used to formalise and reason about programming languages. It provides the mathematical background necessary for the reader to invent, formalise and justify rules with which to reason about a variety of programming languages. Although the treatment is elementary, several of the topics covered are drawn from recent research. The book contains many exercises ranging from the simple to mini projects.

Starting with basic set theory, structural operational semantics (as advocated by Plotkin) is introduced as a means to define the meaning of programming languages along with the basic proof techniques to accompany such definitions. Denotational and axiomatic semantics are illustrated on a simple language of while-programs, and full proofs are given of the equivalence of the operational and denotational semantics and soundness and relative completeness of the axiomatic semantics. A proof of Gödel's incompleteness theorem is included. It emphasises the impossibility of ever achieving a fully complete axiomatic semantics. This is backed up by an appendix providing an introduction to the theory of computability based on while programs. After domain theory, the foundations of denotational semantics is presented, and the semantics and methods of proof for sev-

eral functional languages are treated. The simplest language is that of recursion equations with both call-by-value and call-by-name evaluation. This work is extended to languages with higher and recursive types, which includes a treatment of the eager and lazy λ-calculi. Throughout, the relationship between denotational and operational semantics is stressed, and proofs of the correspondence between the operational and denotational semantics are provided. The treatment of recursive types—one of the more advanced parts of the book—relies on the use of information systems to represent domains. The book concludes with a chapter on parallel programming languages, accompanied by a discussion of methods for verifying nondeterministic and parallel programs.

How to use this book

The dependencies between the chapters are indicated below. It is hoped that this is a help in reading, reference and designing lecture courses. For example, an introductory course on "Logic and computation" could be based on chapters 1 to 7 with additional use of the Appendix. The Appendix covers computability, on the concepts of which Chapter 7 depends—it could be bypassed by readers with a prior knowledge of this topic. Instead, a mini course on "Introductory semantics" might be built on chapters 1 to 5, perhaps supplemented by 14. The chapters 8, 10 and 12 could form a primer in "Domain theory"—this would require a very occasional and easy reference to Chapter 5. Chapters 8-13 provide "A mathematical foundation for functional programming." Chapter 14, a survey of "Nondeterminism and parallelism," is fairly self-contained relying, in the main, just on Chapter 2; however, its discussion of model checking makes use of the Knaster-Tarski Theorem, of which a proof can be found in Chapter 5.

Some of the exercises include small implementation tasks. In the course at Aarhus it was found very helpful to use Prolog, for example to enliven the early treatment of the operational semantics. The use of Standard ML or Miranda is perhaps even more appropriate, given the treatment of such languages in the later chapters.

Acknowledgements

Right at the start I should acknowledge the foundational work of Dana Scott and Gordon Plotkin as having a basic influence on this book. As will be clear from reading the book, it has been influenced a great deal by Gordon Plotkin's work, especially by his notes for lectures on complete partial orders and denotational semantics at Edinburgh University.

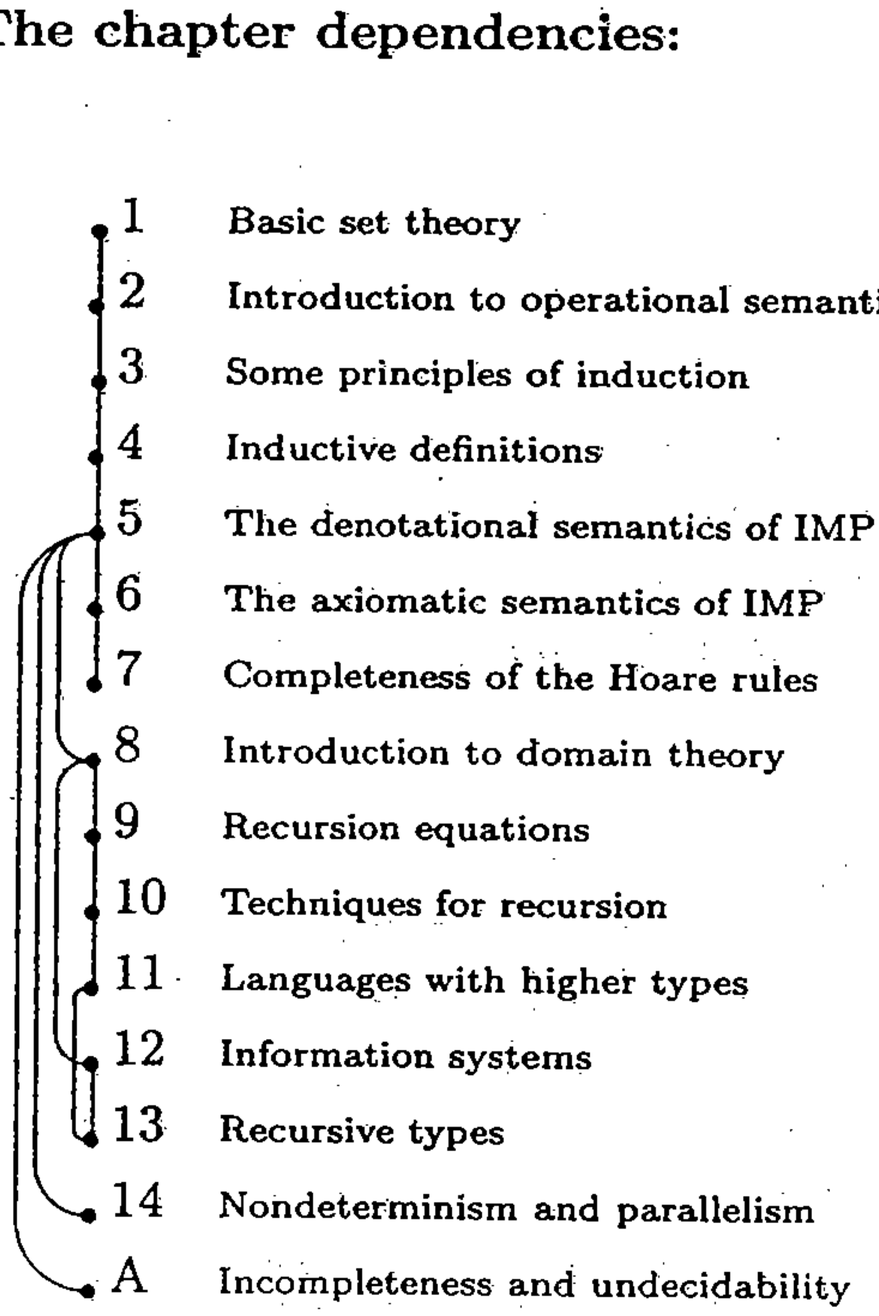
At Cambridge, comments of Tom Melham, Ken Moody, Larry Paulson and Andy Pitts have been very helpful (in particular, Andy's lecture notes and comments on Eugenio Moggi's work have been incorporated into my presentation of domain theory). At Aarhus, Mogens Nielsen provided valuable feedback and encouragement from a course

he gave from an early draft. Recommendations of Erik Meineche Schmidt improved the proofs of relative completeness and Gödel's theorem. Numerous students at Aarhus have supplied corrections and suggestions. I especially thank Henrik Reif Andersen, Torben Brauner, Christian Clausen, Allan Cheng, Urban Engberg, Torben Amtoft Hansen, Ole Hougaard and Jakob Seligman. Added thanks are due to Bettina Blaaberg Sørensen for her prompt reading and suggestions at various stages in the preparation of this book. I'm grateful to Douglas Gurr for his conscientious criticism of the chapters on domain theory. Kim Guldstrand Larsen suggested improvements to the chapter on nondeterminism and concurrency.

In the fall of '91, Albert Meyer gave a course based on this book. He, with instructors A.Lent, M.Sheldon, and C.Yoder, very kindly provided a wealth of advice from notification of typos to restructuring of proofs. In addition, Albert kindly provided his notes on computability on which the appendix is based. I thank them and hope they are not disappointed with the outcome.

My thanks go to Karen Møller for help with the typing. Finally, I express my gratitude to MIT Press, especially Terry Ehling, for their patience.

The chapter dependencies:



The Formal Semantics of Programming Languages

1 Basic set theory

This chapter presents the informal, logical and set-theoretic notation and concepts we shall use to write down and reason about our ideas. It simply presents an extension of our everyday language, extended to talk about mathematical objects like sets; it is not to be confused with the formal languages of programming languages or the formal assertions about them that we'll encounter later.

This chapter is meant as a review and for future reference. It is suggested that on a first reading it is read fairly quickly, without attempting to absorb it fully.

1.1 Logical notation

We shall use some informal logical notation in order to stop our mathematical statements getting out of hand. For statements (or assertions) A and B, we shall commonly use abbreviations like:

- $A \,\&\, B$ for (A and B), the conjunction of A and B,
- $A \Rightarrow B$ for (A implies B), which means (if A then B),
- $A \iff B$ to mean (A iff B), which abbreviates (A if and only if B), and expresses the logical equivalence of A and B.

We shall also make statements by forming disjunctions (A or B), with the self-evident meaning, and negations (not A), sometimes written $\neg A$, which is true iff A is false. There is a tradition to write for instance $7 \not< 5$ instead of $\neg(7 < 5)$, which reflects what we generally say: "7 is not less than 5" rather than "not 7 is less than 5."

The statements may contain variables (or unknowns, or place-holders), as in

$$(x \leq 3) \,\&\, (y \leq 7)$$

which is true when the variables x and y over integers stand for integers less than or equal to 3 and 7 respectively, and false otherwise. A statement like $P(x, y)$, which involves variables x, y, is called a predicate (or property, or relation, or condition) and it only becomes true or false when the pair x, y stand for particular things.

We use logical quantifiers $\exists$, read "there exists", and $\forall$, read " for all". Then you can read assertions like

$$\exists x.\ P(x)$$

as abbreviating "for some x, $P(x)$" or "there exists x such that $P(x)$", and

$$\forall x.\ P(x)$$

as abbreviating " for all x, $P(x)$" or "for any x, $P(x)$". The statement

$$\exists x, y, \cdots, z.\ P(x, y, \cdots, z)$$

abbreviates

$$\exists x \exists y \cdots \exists z.\ P(x, y, \cdots, z),$$

and

$$\forall x, y, \cdots, z.\ P(x, y, \cdots, z)$$

abbreviates

$$\forall x \forall y \cdots \forall z.\ P(x, y, \cdots, z).$$

Later, we often wish to specify a set X over which a quantifier ranges. Then one writes $\forall x \in X.\ P(x)$ instead of $\forall x.\ x \in X \Rightarrow P(x)$, and $\exists x \in X.\ P(x)$ instead of $\exists x.\ x \in X\ \&\ P(x)$.

There is another useful notation associated with quantifiers. Occasionally one wants to say not just that there exists some x satisfying a property $P(x)$ but also that x is the *unique* object satisfying $P(x)$. It is traditional to write

$$\exists! x.\ P(x)$$

as an abbreviation for

$$(\exists x.\ P(x))\ \&\ (\forall y, z.\ P(y)\ \&\ P(z) \Rightarrow y = z)$$

which means that there is some x satisfying the property P and also that if any y, z both satisfy the property P they are equal. This expresses that there exists a unique x satisfying $P(x)$.

1.2 Sets

Intuitively, a set is an (unordered) collection of objects, called its *elements* or *members*. We write $a \in X$ when a is an element of the set X. Sometimes we write *e.g.* $\{a, b, c, \cdots\}$ for the set of elements $a, b, c, \cdots$.

A set X is said to be a *subset* of a set Y, written $X \subseteq Y$, iff every element of X is an element of Y, *i.e.*

$$X \subseteq Y \iff \forall z \in X.\ z \in Y.$$

A set is determined solely by its elements in the sense that two sets are equal iff they have the same elements. So, sets X and Y are equal, written $X = Y$, iff every element of A is a element of B and *vice versa*. This furnishes a method for showing two sets X and Y are equal and, of course, is equivalent to showing $X \subseteq Y$ and $Y \subseteq X$.

1.2.1 Sets and properties

Sometimes a set is determined by a property, in the sense that the set has as elements precisely those which satisfy the property. Then we write

$$X = \{x \mid P(x)\},$$

meaning the set X has as elements precisely all those x for which $P(x)$ is true.

When set theory was being invented it was thought, first of all, that any property $P(x)$ determined a set

$$\{x \mid P(x)\}.$$

It came as a shock when Bertrand Russell realised that assuming the existence of certain sets described in this way gave rise to contradictions.

Russell's paradox is really the demonstration that a contradiction arises from the liberal way of constructing sets above. It proceeds as follows: consider the property

$$x \notin x$$

a way of writing "x is not an element of x". If we assume that properties determine sets, just as described, we can form the set

$$R = \{x \mid x \notin x\}.$$

Either $R \in R$ or not. If so, *i.e.* $R \in R$, then in order for R to qualify as an element of R, from the definition of R, we deduce $R \notin R$. So we end up asserting both something and is negation—a contradiction. If, on the other hand, $R \notin R$ then from the definition of R we see $R \in R$—a contradiction again. Either $R \in R$ or $R \notin R$ lands us in trouble.

We need to have some way which stops us from considering things like R as a sets. In general terms, the solution is to discipline the way in which sets are constructed, so that starting from certain given sets, new sets can only be formed when they are constructed by using particular, safe ways from old sets. We shall not be formal about it, but state those sets we assume to exist right from the start and methods we allow for constructing new sets. Provided these are followed we avoid trouble like Russell's paradox and at the same time have a rich enough world of sets to support most mathematics.

1.2.2 Some important sets

We take the existence of the empty set for granted, along with certain sets of basic elements.

Write $\emptyset$ for the *null*, or *empty* set, and

ω for the set of natural numbers $0, 1, 2, \ldots$.

We shall also take sets of symbols like

$$\{\text{“}a\text{”}, \text{“}b\text{”}, \text{“}c\text{”}, \text{“}d\text{”}, \text{“}e\text{”}, \cdots, \text{“}z\text{”}\}$$

for granted, although we could, alternatively have represented them as particular numbers, for example. The equality relation on a set of symbols is that given by syntactic identity; two symbols are equal iff they are the same.

1.2.3 Constructions on sets

We shall take for granted certain operations on sets which enable us to construct sets from given sets.

Comprehension: If X is a set and $P(x)$ is a property, we can form the set

$$\{x \in X \mid P(x)\}$$

which is another way of writing

$$\{x \mid x \in X \ \& \ P(x)\}.$$

This is the subset of X consisting of all elements x of X which satisfy $P(x)$.

Sometimes we'll use a further abbreviation. Suppose $e(x_1, \ldots, x_n)$ is some expression which for particular elements $x_1 \in X_1, \cdots x_n \in X_n$ yields a particular element and $P(x_1, \ldots, x_n)$ is a property of such $x_1, \ldots, x_n$. We use

$$\{e(x_1, \ldots, x_n) \mid x_1 \in X_1 \ \& \ \cdots \& \ x_n \in X_n \ \& \ P(x_1, \ldots, x_n)\}$$

to abbreviate

$$\{y \mid \exists x_1 \in X_1, \cdots, x_n \in X_n .\ y = e(x_1, \ldots, x_n) \& \ P(x_1, \ldots, x_n)\}.$$

For example,

$$\{2m + 1 \mid m \in \omega \ \& \ m > 1\}$$

is the set of odd numbers greater than 3.

Powerset: We can form a set consisting of the set of all subsets of a set, the so-called *powerset*:

$$\mathcal{P}ow(X) = \{Y \mid Y \subseteq X\}.$$

Indexed sets: Suppose I is a set and that for any $i \in I$ there is a unique object x_i, maybe a set itself. Then

$$\{x_i \mid i \in I\}$$

is a set. The elements x_i are said to be *indexed* by the elements $i \in I$.

Union: The set consisting of the *union* of two sets has as elements those elements which are either elements of one or the other set. It is written and described by:

$$X \cup Y = \{a \mid a \in X \text{ or } a \in Y\}.$$

Big union: Let X be a set of sets. Their *union*

$$\bigcup X = \{a \mid \exists x \in X.\ a \in x\}$$

is a set. When $X = \{x_i \mid i \in I\}$ for some indexing set I we often write $\bigcup X$ as $\bigcup_{i \in I} x_i$.

Intersection: Elements are in the *intersection* $X \cap Y$, of two sets X and Y, iff they are in both sets, *i.e.*

$$X \cap Y = \{a \mid a \in X \ \& \ a \in Y\}.$$

Big intersection: Let X be a nonempty set of sets. Then

$$\bigcap X = \{a \mid \forall x \in X.\ a \in x\}$$

is a set called its *intersection*. When $X = \{x_i \mid i \in I\}$ for a nonempty indexing set I we often write $\bigcap X$ as $\bigcap_{i \in I} x_i$.

Product: Given two elements a, b we can form a set (a, b) which is their ordered pair. To be definite we can take the ordered pair (a, b) to be the set $\{\{a\}, \{a, b\}\}$—this is one particular way of coding the idea of ordered pair as a *set*. As one would hope, two ordered pairs, represented in this way, are equal iff their first components are equal and their second components are equal too, *i.e.*

$$(a, b) = (a', b') \iff a = a' \ \& \ b = b'.$$

In proving properties of ordered pairs this property should be sufficient irrespective of the way in which we have represented ordered pairs as sets.

Exercise 1.1 Prove the property above holds of the suggested representation of ordered pairs. (Don't expect it to be too easy! Consult [39], page 36, or [47], page 23, in case of difficulty.) □

For sets X and Y, their *product* is the set

$$X \times Y = \{(a, b) \mid a \in X \ \& \ b \in Y\},$$

the set of ordered pairs of elements with the first from X and the second from Y.

A triple (a, b, c) is the set $(a, (b, c))$, and the product $X \times Y \times Z$ is the set of triples $\{(x, y, z) \mid x \in X \ \& \ y \in Y \ \& \ z \in Z\}$. More generally $X_1 \times X_2 \times \cdots \times X_n$ consists of the set of n-tuples $(x_1, x_2, \ldots, x_n) = (x_1, (x_2, (x_3, \cdots)))$.

Disjoint union: Frequently we want to join sets together but, in a way which, unlike union, does not identify the same element when it comes from different sets. We do this by making copies of the elements so that when they are copies from different sets they are forced to be distinct.

$$X_0 \uplus X_1 \uplus \cdots \uplus X_n = (\{0\} \times X_0) \cup (\{1\} \times X_1) \cup \cdots \cup (\{n\} \times X_n).$$

In particular, for $X \uplus Y$ the copies $(\{0\} \times X)$ and $(\{1\} \times Y)$ have to be disjoint, in the sense that

$$(\{0\} \times X) \cap (\{1\} \times Y) = \emptyset,$$

because any common element would be a pair with first element both equal to 0 and 1, clearly impossible.

Set difference: We can subtract one set Y from another X, an operation which removes all elements from X which are also in Y.

$$X \setminus Y = \{x \mid x \in X \ \& \ x \notin Y\}.$$

1.2.4 The axiom of foundation

A set is built-up starting from basic sets by using the constructions above. We remark that a property of sets, called the axiom of foundation, follows from our informal understanding of sets and how we can construct them. Consider an element b_1 of a set b_0. It is either a basic element, like an integer or a symbol, or a set. If b_1 is a set then it must have been constructed from sets which have themselves been constructed earlier. Intuitively, we expect any chain of memberships

$$\cdots b_n \in \cdots \in b_1 \in b_0$$

to end in some b_n which is some basic element or the empty set. The statement that any such descending chain of memberships must be finite is called the axiom of foundation, and is an assumption generally made in set theory. Notice the axiom implies that no set X can be a member of itself as, if this were so, we'd get the infinite descending chain

$$\cdots X \in \cdots \in X \in X,$$

—a contradiction.

1.3 Relations and functions

A *binary relation* between X and Y is an element of $\mathcal{P}ow(X \times Y)$, and so a subset of pairs in the relation. When R is a relation $R \subseteq X \times Y$ we shall often write xRy for $(x, y) \in R$.

A *partial function* from X to Y is a relation $f \subseteq X \times Y$ for which

$$\forall x, y, y'.\ (x, y) \in f\ \&\ (x, y') \in f \Rightarrow y = y'.$$

We use the notation $f(x) = y$ when there is a y such that $(x, y) \in f$ and then say $f(x)$ is *defined*, and otherwise say $f(x)$ is *undefined*. Sometimes we write $f : x \mapsto y$, or just $x \mapsto y$ when f is understood, for $y = f(x)$. Occasionally we write just fx, without the brackets, for $f(x)$.

A *(total) function* from X to Y is a partial function from X to Y such that for all $x \in X$ there is some $y \in Y$ such that $f(x) = y$. Although total functions are a special kind of partial function it is traditional to understand something described as simply a function to be a total function, so we always say explicitly when a function is partial.

Note that relations and functions are also sets.

To stress the fact that we are thinking of a partial function f from X to Y as taking an element of X and yielding an element of Y we generally write it as $f : X \rightharpoonup Y$. To indicate that a function f from X to Y is total we write $f : X \to Y$.

We write $(X \rightharpoonup Y)$ for the set of all partial functions from X to Y, and $(X \to Y)$ for the set of all total functions.

Exercise 1.2 Why are we justified in calling $(X \rightharpoonup Y)$ and $(X \to Y)$ sets when X, Y are sets? □

1.3.1 Lambda notation

It is sometimes useful to use the lambda notation (or λ-notation) to describe functions. It provides a way of refering to functions without having to name them. Suppose $f : X \to Y$ is a function which for any element x in X gives a value $f(x)$ which is exactly described by expression e, probably involving x. Then we sometime write

$$\lambda x \in X.e$$

for the function f. Thus

$$\lambda x \in X.e = \{(x, e) \mid x \in X\},$$

so $\lambda x \in X.e$ is just an abbreviation for the set of input-output values determined by the expression e. For example, $\lambda x \in \omega.(x + 1)$ is the successor function.

1.3.2 Composing relations and functions

We compose relations, and so partial and total functions, R between X and Y and S between Y and Z by defining their *composition*, a relation between X and Z, by

$$S \circ R =_{def} \{(x, z) \in X \times Z \mid \exists y \in Y.\ (x, y) \in R\ \&\ (y.\ z) \in S\}.$$

Thus for functions $f : X \to Y$ and $g : Y \to Z$ their composition is the function $g \circ f : X \to Z$. Each set X is associated with an identity function Id_X where $Id_X = \{(x, x) \mid x \in X\}$.

Exercise 1.3 Let $R \subseteq X \times Y$, $S \subseteq Y \times Z$ and $T \subseteq Z \times W$. Convince yourself that $T \circ (S \circ R) = (T \circ S) \circ R$ (*i.e.* composition is associative) and that $R \circ Id_X = Id_Y \circ R = R$ (*i.e.* identity functions act like identities with respect to composition). □

A function $f : X \to Y$ has an *inverse* $g : Y \to X$ iff $g(f(x)) = x$ for all $x \in X$, and $f(g(y)) = y$ for all $y \in Y$. Then the sets X and Y are said to be in *1-1 correspondence.* (Note a function with an inverse has to be total.)

Any set in 1-1 correspondence with a subset of natural numbers ω is said to be *countable.*

Exercise 1.4 Let X and Y be sets. Show there is a 1-1 correspondence between the set of functions $(X \to \mathcal{P}ow(Y))$ and the set of relations $\mathcal{P}ow(X \times Y)$. □

Cantor's diagonal argument

Late last century, Georg Cantor, one of the pioneers in set theory, invented a method of argument, the gist of which reappears frequently in the theory of computation. Cantor used a *diagonal argument* to show that X and $\mathcal{P}ow(X)$ are never in 1-1 correspondence for any set X. This fact is intuitively clear for finite sets but also holds for infinite sets. He argued by *reductio ad absurdum, i.e.*, by showing that supposing otherwise led to a contradiction:

Suppose a set X is in 1-1 correspondence with its powerset $\mathcal{P}ow(X)$. Let $\theta : X \to \mathcal{P}ow(X)$ be the 1-1 correspondence. Form the set

$$Y = \{x \in X \mid x \notin \theta(x)\}$$

which is clearly a subset of X and therefore in correspondence with an element $y \in X$. That is $\theta(y) = Y$. Either $y \in Y$ or $y \notin Y$. But both possibilities are absurd. For, if $y \in Y$ then $y \in \theta(y)$ so $y \notin Y$, while, if $y \notin Y$ then $y \notin \theta(y)$ so $y \in Y$. We conclude that our first supposition must be false, so there is no set in 1-1 correspondence with its powerset.

Cantor's argument is reminiscient of Russell's paradox. But whereas the contradiction in Russell's paradox arises out of a fundamental, mistaken assumption about how to construct sets, the contradiction in Cantor's argument comes from denying the fact one wishes to prove.

To see why it is called a diagonal argument, imagine that the set X, which we suppose is in 1-1 correspondence with $\mathcal{P}ow(X)$, can be enumerated as $x_0, x_1, x_2, \cdots, x_n, \cdots$. Imagine we draw a table to represent the 1-1 correspondence θ along the following lines. In the

ith row and jth column is placed 1 if $x_i \in \theta(x_j)$ and 0 otherwise. The table below, for instance, represents a situation where $x_0 \notin \theta(x_0)$, $x_1 \in \theta(x_0)$ and $x_i \in \theta(x_j)$.

	$\theta(x_0)$	$\theta(x_1)$	$\theta(x_2)$	$\cdots$	$\theta(x_j)$	$\cdots$
x_0	0	1	1	$\cdots$	1	$\cdots$
x_1	1	1	1	$\cdots$	0	$\cdots$
x_2	0	0	1	$\cdots$	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	
x_i	0	1	0	$\cdots$	1	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	

The set Y which plays a key role in Cantor's argument is defined by running down the diagonal of the table interchanging 0's and 1's in the sense that x_n is put in the set iff the nth entry along the diagonal is a 0.

Exercise 1.5 Show for any sets X and Y, with Y containing at least two elements, that there cannot be a 1-1 correspondence between X and the set of functions $(X \to Y)$. □

1.3.3 Direct and inverse image of a relation

We extend relations, and thus partial and total functions, $R : X \times Y$ to functions on subsets by taking

$$RA = \{y \in Y \mid \exists x \in A.\ (x, y) \in R\}$$

for $A \subseteq X$. The set RA is called the *direct image* of A under R. We define

$$R^{-1}B = \{x \in X \mid \exists y \in B.\ (x, y) \in R\}$$

for $B \subseteq Y$. The set $R^{-1}B$ is called the *inverse image* of B under R. Of course, the same notions of direct and inverse image also apply in the special case where the relation is a function.

1.3.4 Equivalence relations

An *equivalence relation* is a relation $R \subseteq X \times X$ on a set X which is

- reflexive: $\forall x \in X.\ xRx$,
- symmetric: $\forall x, y \in X.\ xRy \Rightarrow yRx$ and
- transitive: $\forall x, y, z \in X.\ xRy\ \&\ yRz \Rightarrow xRz$.

If R is an equivalence relation on X then the *(R-)equivalence class* of an element $x \in X$ is the subset $\{x\}_R =_{def} \{y \in X \mid yRx\}$.

Exercise 1.6 Let R be an equivalence relation on a set X. Show if $\{x\}_R \cap \{y\}_R \neq \emptyset$ then $\{x\}_R = \{y\}_R$, for any elements $x, y \in X$. □

Exercise 1.7 Let xRy be a relation on a set of sets X which holds iff the sets x and y in X are in 1-1 correspondence. Show that R is an equivalence relation. □

Let R be a relation on a set X. Define $R^0 = Id_X$, the identity relation on the set X, and $R^1 = R$ and, assuming R^n is defined, define

$$R^{n+1} = R \circ R^n.$$

So, R^n is the relation $R \circ \cdots \circ R$, obtained by taking n compositions of R. Define the *transitive closure* of R to be the relation

$$R^+ = \bigcup_{n \in \omega} R^{n+1}.$$

Define the transitive, reflexive closure of a relation R on X to be the relation

$$R^* = \bigcup_{n \in \omega} R^n,$$

so $R^* = Id_X \cup R^+$.

Exercise 1.8 Let R be a relation on a set X. Write R^{op} for the opposite, or converse, relation $R^{op} = \{(y, x) \mid (x, y) \in R\}$. Show $(R \cup R^{op})^*$ is an equivalence relation. Show $R^* \cup (R^{op})^*$ need not be an equivalence relation. □

1.4 Further reading

Our presentation amounts to an informal introduction to the Zermelo-Fraenkel axiomatisation of set theory but with atoms, to avoid thinking of symbols as being coded by sets. If you'd like more material to read I recommend Halmos's "Naive Set Theory" [47] for a very readable introduction to sets. Another good book is Enderton's "Elements of set theory" [39], though this is a much larger work.

2 Introduction to operational semantics

This chapter presents the syntax of a programming language, **IMP**, a small language of while programs. **IMP** is called an "imperative" language because program execution involves carrying out a series of explicit commands to change state. Formally, **IMP**'s behaviour is described by rules which specify how its expressions are evaluated and its commands are executed. The rules provide an operational semantics of **IMP** in that they are close to giving an implementation of the language, for example, in the programming language Prolog. It is also shown how they furnish a basis for simple proofs of equivalence between commands.

2.1 IMP—a simple imperative language

Firstly, we list the syntactic sets associated with **IMP**:

- numbers **N**, consisting of positive and negative integers with zero,
- truth values $\mathbf{T} = \{\mathbf{true}, \mathbf{false}\}$,
- locations **Loc**,
- arithmetic expressions **Aexp**,
- boolean expressions **Bexp**,
- commands **Com**.

We assume the syntactic structure of numbers and locations is given. For instance, the set **Loc** might consist of non-empty strings of letters or such strings followed by digits, while **N** might be the set of signed decimal numerals for positive and negative whole numbers—indeed these are the representations we use when considering specific examples. (Locations are often called program variables but we reserve that term for another concept.)

For the other syntactic sets we have to say how their elements are built-up. We'll use a variant of BNF (Backus-Naur form) as a way of writing down the rules of formation of the elements of these syntactic sets. The formation rules will express things like:

> If a_0 and a_1 are arithmetic expressions then so is $a_0 + a_1$.

It's clear that the symbols a_0 and a_1 are being used to stand for any arithmetic expression. In our informal presentation of syntax we'll use such *metavariables* to range over the syntactic sets—the metavariables a_0, a_1 above are understood to range over the set of arithmetic expressions. In presenting the syntax of **IMP** we'll follow the convention that

- n, m range over numbers N,
- X, Y range over locations **Loc**,
- a ranges over arithmetic expressions **Aexp**,
- b ranges over boolean expressions **Bexp**,
- c ranges over commands **Com**.

The metavariables we use to range over the syntactic categories can be primed or subscripted. So, *e.g.*, X, X', X_0, X_1, Y'' stand for locations.

We describe the formation rules for arithmetic expressions **Aexp** by:

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

The symbol "::=" should be read as "can be" and the symbol "|" as "or". Thus an arithmetic expression a can be a number n or a location X or $a_0 + a_1$ or $a_0 - a_1$ or $a_0 \times a_1$, built from arithmetic expressions a_0 and a_1.

Notice our notation for the formation rules of arithmetic expressions does not tell us how to parse

$$2 + 3 \times 4 - 5,$$

whether as $2 + ((3 \times 4) - 5)$ or as $(2 + 3) \times (4 - 5)$ *etc.*. The notation gives the so-called *abstract syntax* of arithmetic expressions in that it simply says how to build up new arithmetic expressions. For any arithmetic expression we care to write down it leaves us the task of putting in enough parentheses to ensure it has been built-up in a unique way. It is helpful to think of abstract syntax as specifying the parse trees of a language; it is the job of *concrete syntax* to provide enough information through parentheses or orders of precedence between operation symbols for a string to parse uniquely. Our concerns are with the meaning of programming languages and not with the theory of how to write them down. Abstract syntax suffices for our purposes.

Here are the formation rules for the whole of **IMP**:

For **Aexp**:

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

For **Bexp**:

$$b ::= \mathbf{true} \mid \mathbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

For **Com**:

$$c ::= \mathbf{skip} \mid X := a \mid c_0; c_1 \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b\ \mathbf{do}\ c$$

From a set-theory point of view this notation provides an *inductive definition* of the syntactic sets of IMP, which are the least sets closed under the formation rules, in a sense we'll make clear in the next two chapters. For the moment, this notation should be viewed as simply telling us how to construct elements of the syntactic sets.

We need some notation to express when two elements e_0, e_1 of the same syntactic set are identical, in the sense of having been built-up in exactly the same way according to the abstract syntax or, equivalently, having the same parse tree. We use $e_0 \equiv e_1$ to mean e_0 is identical to e_1. The arithmetic expression $3 + 5$ built up from the numbers 3 and 5 is not syntactically identical to the expression 8 or $5 + 3$, though of course we expect them to evaluate to the same number. Thus we do *not* have $3 + 5 \equiv 5 + 3$. Note we *do* have $(3 + 5) \equiv 3 + 5$!

Exercise 2.1 If you are familiar with the programming language ML (see *e.g.*[101]) or Miranda (see *e.g.*[22]) define the syntactic sets of **IMP** as datatypes. If you are familiar with the programming language Prolog (see *e.g.*[31]) program the formation rules of **IMP** in it. Write a program to check whether or not $e_0 \equiv e_1$ holds of syntactic elements e_0, e_1. □

So much for the syntax of **IMP**. Let's turn to its semantics, how programs behave when we run them.

2.2 The evaluation of arithmetic expressions

Most probably, the reader has an intuitive model with which to understand the behaviours of programs written in **IMP**. Underlying most models is an idea of state determined by what contents are in the locations. With respect to a state, an arithmetic expression evaluates to an integer and a boolean expression evaluates to a truth value. The resulting values can influence the execution of commands which will lead to changes in state. Our formal description of the behaviour of **IMP** will follow this line. First we define *states* and then the *evaluation* of integer and boolean expressions, and finally the *execution* of commands.

The set of *states* Σ consists of functions $\sigma : \mathbf{Loc} \to \mathbf{N}$ from locations to numbers. Thus $\sigma(X)$ is the value, or contents, of location X in state σ.

Consider the evaluation of an arithmetic expression a in a state σ. We can represent the situation of expression a waiting to be evaluated in state σ by the pair $\langle a, \sigma \rangle$. We shall define an evaluation relation between such pairs and numbers

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

meaning: expression a in state σ evaluates to n. Call pairs $\langle a, \sigma \rangle$, where a is an arithmetic expression and σ is a state, arithmetic-expression *configurations*.

Consider how we might explain to someone how to evaluate an arithmetic expression $(a_0 + a_1)$. We might say something along the lines of:

1. Evaluate a_0 to get a number n_0 as result and
2. Evaluate a_1 to get a number n_1 as result.
3. Then add n_0 and n_1 to get n, say, as the result of evaluating $a_0 + a_1$.

Although informal we can see that this specifies how to evaluate a sum in terms of how to evaluate its summands; the specification is *syntax-directed*. The formal specification of the evaluation relation is given by rules which follow intuitive and informal descriptions like this rather closely.

We specify the evaluation relation in a syntax-directed way, by the following rules:

Evaluation of numbers:

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

Thus any number is already evaluated with itself as value.

Evaluation of locations:

$$\langle X, \sigma \rangle \rightarrow \sigma(X)$$

Thus a location evaluates to its contents in a state.

Evaluation of sums:

$$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma \rangle \rightarrow n}$$ where n is the sum of n_0 and n_1.

Evaluation of subtractions:

$$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 - a_1, \sigma \rangle \rightarrow n}$$ where n is the result of subtracting n_1 from n_0.

Evaluation of products:

$$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 \times a_1, \sigma \rangle \rightarrow n}$$ where n is the product of n_0 and n_1.

How are we to read such rules? The rule for sums can be read as:
If $\langle a_0, \sigma \rangle \rightarrow n_0$ and $\langle a_1, \sigma \rangle \rightarrow n_1$ then $\langle a_0 + a_1, \sigma \rangle \rightarrow n$, where n is the sum of n_0 and n_1. The rule has a *premise* and a *conclusion* and we have followed the common practice of writing the rule with the premise above and the conclusion below a solid line. The rule will be applied in derivations where the facts below the line are derived from facts above.

Some rules like those for evaluating numbers or locations require no premise. Sometimes they are written with a line, for example, as in

$$\frac{}{\langle n, \sigma\rangle \to n}.$$

Rules with empty premises are called *axioms*. Given any arithmetic expression a, state σ and number n, we take a in σ to evaluate to n, *i.e.* $\langle a, \sigma\rangle \to n$, if it can be derived from the rules starting from the axioms, in a way to be made precise soon.

The rule for sums expresses that the sum of two expressions evaluates to the number which is obtained by summing the two numbers which the summands evaluate to. It leaves unexplained the mechanism by which the sum of two numbers is obtained. I have chosen not to analyse in detail how numerals are constructed and the above rules only express how locations and operations $+, -, \times$ can be eliminated from expressions to give the number they evaluate to. If, on the other hand, we chose to describe a particular numeral system, like decimal or roman, further rules would be required to specify operations like multiplication. Such a level of description can be important when considering devices in hardware, for example. Here we want to avoid such details—we all know how to do simple arithmetic!

The rules for evaluation are written using metavariables n, X, a_0, a_1 ranging over the appropriate syntactic sets as well as σ ranging over states. A *rule instance* is obtained by instantiating these to particular numbers, locations and expressions and states. For example, when σ_0 is the particular state, with 0 in each location, this is a rule instance:

$$\frac{\langle 2, \sigma_0\rangle \to 2 \quad \langle 3, \sigma_0\rangle \to 3}{\langle 2 \times 3, \sigma_0\rangle \to 6}$$

So is this:

$$\frac{\langle 2, \sigma_0\rangle \to 3 \quad \langle 3, \sigma_0\rangle \to 4}{\langle 2 \times 3, \sigma_0\rangle \to 12},$$

though not one in which the premises, or conclusion, can ever be derived.

To see the structure of derivations, consider the evaluation of $a \equiv (\mathrm{Init} + 5) + (7 + 9)$ in state σ_0, where Init is a location with $\sigma_0(\mathrm{Init}) = 0$. Inspecting the rules we see that this requires the evaluation of $(\mathrm{Init} + 5)$ and $(7 + 9)$ and these in turn may depend on other evaluations. In fact the evaluation of $\langle a, \sigma_0\rangle$ can be seen as depending on a tree of evaluations:

$$\frac{\dfrac{\dfrac{}{\langle \mathrm{Init}, \sigma_0\rangle \to 0} \quad \dfrac{}{\langle 5, \sigma_0\rangle \to 5}}{\langle (\mathrm{Init} + 5), \sigma_0\rangle \to 5} \qquad \dfrac{\dfrac{}{\langle 7, \sigma_0\rangle \to 7} \quad \dfrac{}{\langle 9, \sigma_0\rangle \to 9}}{\langle 7 + 9, \sigma_0\rangle \to 16}}{\langle (\mathrm{Init} + 5) + (7 + 9), \sigma_0\rangle \to 21}$$

We call such a structure a *derivation tree* or simply a *derivation.* It is built out of instances of the rules in such a way that all the premises of instances of rules which occur are conclusions of instances of rules immediately above them, so right at the top come the axioms, marked by the lines with no premises above them. The conclusion of the bottom-most rule is called the conclusion of the derivation. Something is said to be *derived* from the rules precisely when there is a derivation with it as conclusion.

In general, we write $\langle a, \sigma \rangle \rightarrow n$, and say a in σ evaluates to n, iff it can be derived from the rules for the evaluation of arithmetic expressions. The particular derivation above concludes with

$$\langle (\mathrm{Init} + 5) + (7 + 9), \sigma_0 \rangle \rightarrow 21.$$

It follows that $(\mathrm{Init} + 5) + (7 + 9)$ in state σ evaluates to 21—just what we want.

Consider the problem of evaluating an arithmetic expression a in some state σ. This amounts to finding a derivation in which the left part of the conclusion matches $\langle a, \sigma \rangle$. The search for a derivation is best achieved by trying to build a derivation in an upwards fashion: Start by finding a rule with conclusion matching $\langle a, \sigma \rangle$; if this is an axiom the derivation is complete; otherwise try to build derivations up from the premises, and, if successful, fill in the conclusion of the first rule to complete the derivation with conclusion of the form $\langle a, \sigma \rangle \rightarrow n$.

Although it doesn't happen for the evaluation of arithmetic expressions, in general, more than one rule has a left part which matches a given configuration. To guarantee finding a derivation tree with conclusion that matches, when one exists, all of the rules with left part matching the configuration must be considered, to see if they can be the conclusions of derivations. All possible derivations with conclusion of the right form must be constructed "in parallel".

In this way the rules provide an algorithm for the evaluation of arithmetic expressions based on the search for a derivation tree. Because it can be implemented fairly directly the rules specify the meaning, or semantics, of arithmetic expressions in an operational way, and the rules are said to give an *operational semantics* of such expressions. There are other ways to give the meaning of expressions in a way that leads fairly directly to an implementation. The way we have chosen is just one—any detailed description of an implementation is also an operational semantics. The style of semantics we have chosen is one which is becoming prevalent however. It is one which is often called *structural operational semantics* because of the syntax-directed way in which the rules are presented. It is also called *natural semantics* because of the way derivations resemble proofs in natural deduction—a method of constructing formal proofs. We shall see more complicated, and perhaps more convincing, examples of operational semantics later.

The evaluation relation determines a natural equivalence relation on expressions. De-

fine

$$a_0 \sim a_1 \text{ iff } (\forall n \in \mathbf{N} \forall \sigma \in \Sigma.\ \langle a_0, \sigma\rangle \to n \iff \langle a_1, \sigma\rangle \to n),$$

which makes two arithmetic expressions equivalent if they evaluate to the same value in all states.

Exercise 2.2 Program the rules for the evaluation of arithmetic expressions in Prolog and/or ML (or another language of your choice). This, of course, requires a representation of the abstract syntax of such expressions in Prolog and/or ML. □

2.3 The evaluation of boolean expressions

We show how to evaluate boolean expressions to truth values (**true**, **false**) with the following rules:

$$\langle \mathbf{true}, \sigma\rangle \to \mathbf{true}$$

$$\langle \mathbf{false}, \sigma\rangle \to \mathbf{false}$$

$$\frac{\langle a_0, \sigma\rangle \to n \quad \langle a_1, \sigma\rangle \to m}{\langle a_0 = a_1, \sigma\rangle \to \mathbf{true}} \quad \text{if } n \text{ and } m \text{ are equal}$$

$$\frac{\langle a_0, \sigma\rangle \to n \quad \langle a_1, \sigma\rangle \to m}{\langle a_0 = a_1, \sigma\rangle \to \mathbf{false}} \quad \text{if } n \text{ and } m \text{ are unequal}$$

$$\frac{\langle a_0, \sigma\rangle \to n \quad \langle a_1, \sigma\rangle \to m}{\langle a_0 \leq a_1, \sigma\rangle \to \mathbf{true}} \quad \text{if } n \text{ is less than or equal to } m$$

$$\frac{\langle a_0, \sigma\rangle \to n \quad \langle a_1, \sigma\rangle \to m}{\langle a_0 \leq a_1, \sigma\rangle \to \mathbf{false}} \quad \text{if } n \text{ is not less than or equal to } m$$

$$\frac{\langle b, \sigma\rangle \to \mathbf{true}}{\langle \neg b, \sigma\rangle \to \mathbf{false}} \qquad \frac{\langle b, \sigma\rangle \to \mathbf{false}}{\langle \neg b, \sigma\rangle \to \mathbf{true}}$$

$$\frac{\langle b_0, \sigma\rangle \to t_0 \quad \langle b_1, \sigma\rangle \to t_1}{\langle b_0 \wedge b_1, \sigma\rangle \to t}$$

where t is **true** if $t_0 \equiv$ **true** and $t_1 \equiv$ **true**, and is **false** otherwise.

$$\frac{\langle b_0, \sigma\rangle \to t_0 \quad \langle b_1, \sigma\rangle \to t_1}{\langle b_0 \vee b_1, \sigma\rangle \to t}$$

where t is **true** if $t_0 \equiv$ **true** or $t_1 \equiv$ **true**, and is **false** otherwise.

This time the rules tell us how to eliminate all boolean operators and connectives and so reduce a boolean expression to a truth value.

Again, there is a natural equivalence relation on boolean expressions. Two expressions are equivalent if they evaluate to the same truth value in all states. Define

$$b_0 \sim b_1 \text{ iff } \forall t \forall \sigma \in \Sigma.\ \langle b_0, \sigma\rangle \to t \iff \langle b_1, \sigma\rangle \to t.$$

It may be a concern that our method of evaluating expressions is not the most efficient. For example, according to the present rules, to evaluate a conjunction $b_0 \wedge b_1$ we must evaluate both b_0 and b_1 which is clearly unnecessary if b_0 evaluates to **false** before b_1 is fully evaluated. A more efficient evaluation strategy is to first evaluate b_0 and then only in the case where its evaluation yields **true** to proceed with the evaluation of b_1. We can call this strategy *left-first-sequential* evaluation. Its evaluation rules are:

$$\frac{\langle b_0, \sigma\rangle \to \textbf{false}}{\langle b_0 \wedge b_1, \sigma\rangle \to \textbf{false}}$$

$$\frac{\langle b_0, \sigma\rangle \to \textbf{true} \quad \langle b_1, \sigma\rangle \to \textbf{false}}{\langle b_0 \wedge b_1, \sigma\rangle \to \textbf{false}}$$

$$\frac{\langle b_0, \sigma\rangle \to \textbf{true} \quad \langle b_1, \sigma\rangle \to \textbf{true}}{\langle b_0 \wedge b_1, \sigma\rangle \to \textbf{true}}$$

Exercise 2.3 Write down rules to evaluate boolean expressions of the form $b_0 \vee b_1$, which take advantage of the fact that there is no need to evaluate b in **true** $\vee\, b$ as the result will be true independent of the result of evaluating b. The rules written down should describe a method of left-sequential evaluation. Of course, by symmetry, there is a method of right-sequential evaluation. □

Exercise 2.4 Write down rules which express the "parallel" evaluation of b_0 and b_1 in $b_0 \vee b_1$ so that $b_0 \vee b_1$ evaluates to **true** if either b_0 evaluates to **true**, and b_1 is unevaluated, or b_1 evaluates to **true**, and b_0 is unevaluated. □

It may have been felt that we side-stepped too many issues by assuming we were given mechanisms to perform addition or conjunction of truth values for example. If so try:

Exercise 2.5 Give a semantics in the same style but for expressions which evaluate to strings (or lists) instead of integers and truth-values. Choose your own basic operations on strings, define expressions based on them, define the evaluation of expressions in the style used above. Can you see how to use your language to implement the expression part of **IMP** by representing integers as strings and operations on integers as operations on strings? (Proving that you have implemented the operations on integers correctly is quite hard.) □

2.4 The execution of commands

The role of expressions is to evaluate to values in a particular state. The role of a program, and so commands, is to execute to change the state. When we execute an **IMP** program we shall assume that initially the state is such that all locations are set to zero. So the *initial state* σ_0 has the property that $\sigma_0(X) = 0$ for all locations X. As we all know the execution may *terminate* in a final state, or may *diverge* and never yield a final state. A pair $\langle c, \sigma \rangle$ represents the *(command) configuration* from which it remains to execute command c from state σ. We shall define a relation

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

which means the (full) execution of command c in state σ terminates in final state σ'. For example,

$$\langle X := 5, \sigma \rangle \to \sigma'$$

where σ' is the state σ updated to have 5 in location X. We shall use this notation:

Notation: Let σ be a state. Let $m \in \mathbf{N}$. Let $X \in \mathbf{Loc}$. We write $\sigma[m/X]$ for the state obtained from σ by replacing its contents in X by m, i.e. define

$$\sigma[m/X](Y) = \begin{cases} m & \text{if } Y = X, \\ \sigma(Y) & \text{if } Y \neq X. \end{cases}$$

Now we can instead write

$$\langle X := 5, \sigma \rangle \to \sigma[5/X].$$

The execution relation for arbitrary commands and states is given by the following rules.

Rules for commands

Atomic commands:

$$\langle \mathbf{skip}, \sigma \rangle \to \sigma$$

$$\frac{\langle a, \sigma \rangle \to m}{\langle X := a, \sigma \rangle \to \sigma[m/X]}$$

Sequencing:

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

Conditionals:

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

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

While-loops:

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

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

Again there is a natural equivalence relation on commands. Define

$$c_0 \sim c_1 \text{ iff } \forall \sigma, \sigma' \in \Sigma.\ \langle c_0, \sigma \rangle \to \sigma' \iff \langle c_1, \sigma \rangle \to \sigma'.$$

Exercise 2.6 Complete Exercise 2.2 of Section 2.2, by coding the rules for the evaluation of boolean expressions and execution of commands in Prolog and/or ML. □

Exercise 2.7 Let $w \equiv$ **while true do skip**. By considering the form of derivations, explain why, for any state σ, there is no state σ' such that $\langle w, \sigma \rangle \to \sigma'$. □

2.5 A simple proof

The operational semantics of the syntactic sets **Aexp**, **Bexp** and **Com** has been given using the same method. By means of rules we have specified the evaluation relations of

both types of expressions and the execution relation of commands. All three relations are examples of the general notion of *transition relations*, or *transition systems*, in which the configurations are thought of as some kind of state and the relations as expressing possible transitions, or changes, between states. For instance, we can consider each of

$$\langle 3, \sigma\rangle \to 3, \quad \langle \mathbf{true}, \sigma\rangle \to \mathbf{true}, \quad \langle X := 2, \sigma\rangle \to \sigma[2/X].$$

to be transitions.

Because the transition systems for **IMP** are given by rules, we have an elementary, but very useful, proof technique for proving properties of the operational semantics **IMP**.

As an illustration, consider the execution of a while-command $w \equiv \mathbf{while}\ b\ \mathbf{do}\ c$, with $b \in \mathbf{Bexp}, c \in \mathbf{Com}$, in a state σ. We expect that if b evaluates to **true** in σ then w executes as c followed by w again, and otherwise, in the case where b evaluates to **false**, that the execution of w terminates immediately with the state unchanged. This informal explanation of the execution of commands leads us to expect that for all states σ, σ'

$$\langle w, \sigma\rangle \to \sigma' \text{ iff } \langle \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}, \sigma\rangle \to \sigma',$$

i.e., that the following proposition holds.

Proposition 2.8 *Let* $w \equiv \mathbf{while}\ b\ \mathbf{do}\ c$ *with* $b \in \mathbf{Bexp}, c \in \mathbf{Com}$. *Then*

$$w \sim \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}.$$

Proof: We want to show

$$\langle w, \sigma\rangle \to \sigma' \text{ iff } \langle \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}, \sigma\rangle \to \sigma',$$

for all states σ, σ'.

"$\Rightarrow$": Suppose $\langle w, \sigma\rangle \to \sigma'$, for states σ, σ'. Then there must be a derivation of $\langle w, \sigma\rangle \to \sigma'$. Consider the possible forms such a derivation can take. Inspecting the rules for commands we see the final rule of the derivation is either

$$\frac{\langle b, \sigma\rangle \to \mathbf{false}}{\langle w, \sigma\rangle \to \sigma} \qquad (1 \Rightarrow)$$

or

$$\frac{\langle b, \sigma\rangle \to \mathbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle w, \sigma''\rangle \to \sigma'}{\langle w, \sigma\rangle \to \sigma'} \qquad (2 \Rightarrow)$$

In case $(1 \Rightarrow)$, the derivation of $\langle w, \sigma\rangle \to \sigma'$ must have the form

$$\frac{\begin{array}{c}\vdots\\ \langle b, \sigma\rangle \to \mathbf{false}\end{array}}{\langle w, \sigma\rangle \to \sigma}$$

which includes a derivation of $\langle b, \sigma\rangle \to \mathbf{false}$. Using this derivation we can build the following derivation of $\langle \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}, \sigma\rangle \to \sigma$:

$$\frac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \mathbf{false}} \quad \overline{\langle \mathbf{skip}, \sigma\rangle \to \sigma}}{\langle \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}, \sigma\rangle \to \sigma}$$

In case $(2 \Rightarrow)$, the derivation of $\langle w, \sigma\rangle \to \sigma'$ must take the form

$$\frac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \mathbf{true}} \quad \dfrac{\vdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\vdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle w, \sigma\rangle \to \sigma'}$$

which includes derivations of $\langle b, \sigma\rangle \to \mathbf{true}$, $\langle c, \sigma\rangle \to \sigma''$ and $\langle w, \sigma''\rangle \to \sigma'$. From these we can obtain a derivation of $\langle c; w, \sigma\rangle \to \sigma'$, *viz.*

$$\frac{\dfrac{\vdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\vdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle c; w, \sigma\rangle \to \sigma'}$$

We can incorporate this into a derivation:

$$\frac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \mathbf{true}} \quad \dfrac{\dfrac{\vdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\vdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle c; w, \sigma\rangle \to \sigma'}}{\langle \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}, \sigma\rangle \to \sigma'}$$

In either case, $(1 \Rightarrow)$ or $(2 \Rightarrow)$, we obtain a derivation of

$$\langle \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}, \sigma\rangle \to \sigma'$$

from a derivation of

$$\langle w, \sigma\rangle \to \sigma'.$$

Thus

$$\langle w, \sigma\rangle \to \sigma' \text{ implies } \langle \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}, \sigma\rangle \to \sigma',$$

for any states σ, σ'.

"$\Leftarrow$": We also want to show the converse, that $\langle \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}, \sigma\rangle \to \sigma'$ implies $\langle w, \sigma\rangle \to \sigma'$, for all states σ, σ'.

Suppose $\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma'$, for states σ, σ'. Then there is a derivation with one of two possible forms:

$$\frac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \textbf{false}} \quad \dfrac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma}}{\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma} \qquad (1 \Leftarrow)$$

$$\frac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \textbf{true}} \quad \dfrac{\vdots}{\langle c; w, \sigma\rangle \to \sigma'}}{\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma'} \qquad (2 \Leftarrow)$$

where in the first case, we also have $\sigma' = \sigma$, got by noting the fact that

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma}$$

is the only possible derivation associated with **skip**.

From either derivation, $(1 \Leftarrow)$ or $(2 \Leftarrow)$, we can construct a derivation of $\langle w, \sigma\rangle \to \sigma'$. The second case, $(2 \Leftarrow)$, is the more complicated. Derivation $(2 \Leftarrow)$ includes a derivation of $\langle c; w, \sigma\rangle \to \sigma'$ which has to have the form

$$\frac{\dfrac{\vdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\vdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle c; w, \sigma\rangle \to \sigma'}$$

for some state σ''. Using the derivations of $\langle c, \sigma\rangle \to \sigma''$ and $\langle w, \sigma''\rangle \to \sigma'$ with that for $\langle b, \sigma\rangle \to \textbf{true}$, we can produce the derivation

$$\frac{\dfrac{\vdots}{\langle b, \sigma\rangle \to \textbf{true}} \quad \dfrac{\vdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\vdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle w, \sigma\rangle \to \sigma'}$$

More directly, from the derivation $(1 \Leftarrow)$, we can construct a derivation of $\langle w, \sigma\rangle \to \sigma'$ (How?).

Thus if $\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma'$ then $\langle w, \sigma\rangle \to \sigma'$ for any states σ, σ'. We can now conclude that

$$\langle w, \sigma\rangle \to \sigma' \text{ iff } \langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma\rangle \to \sigma',$$

for all states σ, σ', and hence

$$w \sim \textbf{if } b \textbf{ then } c; w \textbf{ else skip}$$

as required. □

This simple proof of the equivalence of while-command and its conditional unfolding exhibits an important technique: in order to prove a property of an operational semantics it is helpful to consider the various possible forms of derivations. This idea will be used again and again, though never again in such laborious detail. Later we shall meet other techniques, like "rule induction" which, in principle, can supplant the technique used here. The other techniques are more abstract however, and sometimes more confusing to apply. So keep in mind the technique of considering the forms of derivations when reasoning about operational semantics.

2.6 Alternative semantics

The evaluation relations

$$\langle a, \sigma \rangle \to n \text{ and } \langle b, \sigma \rangle \to t$$

specify the evaluation of expressions in rather large steps; given an expression and a state they yield a value directly. It is possible to give rules for evaluation which capture single steps in the evaluation of expressions. We could instead have defined an evaluation relation between pairs of configurations, taking *e.g.*

$$\langle a, \sigma \rangle \to_1 \langle a', \sigma' \rangle$$

to mean one step in the evaluation of a in state σ yields a' in state σ'. This intended meaning is formalised by taking rules such as the following to specify single steps in the left-to-right evaluation of sum.

$$\frac{\langle a_0, \sigma \rangle \to_1 \langle a_0', \sigma \rangle}{\langle a_0 + a_1, \sigma \rangle \to_1 \langle a_0' + a_1, \sigma \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \to_1 \langle a_1', \sigma \rangle}{\langle n + a_1, \sigma \rangle \to_1 \langle n + a_1', \sigma \rangle}$$

$$\langle n + m, \sigma \rangle \to_1 \langle p, \sigma \rangle$$

where p is the sum of m and n.

Note how the rules formalise the intention to evaluate sums in a left-to-right sequential fashion. To spell out the meaning of the first sum rule above, it says: if one step in the evaluation of a_0 in state σ leads to a_0' in state σ then one step in the evaluation of $a_0 + a_1$ in state σ leads to $a_0' + a_1$ in state σ. So to evaluate a sum first evaluate the component

expression of the sum and when this leads to a number evaluate the second component of the sum, and finally add the corresponding numerals (and we assume a mechanism to do this is given).

Exercise 2.9 Complete the task, begun above, of writing down the rules for $\rightarrow_1$, one step in the evaluation of integer and boolean expressions. What evaluation strategy have you adopted (left-to-right sequential or $\cdots$) ? □

We have chosen to define full execution of commands in particular states through a relation

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

between command configurations. We could instead have based our explanation of the execution of commands on a relation expressing single steps in the execution. A single step relation between two command configurations

$$\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$$

means the execution of one instruction in c from state σ leads to the configuration in which it remains to execute c' in state σ'. For example,

$$\langle X := 5; Y := 1, \sigma \rangle \rightarrow_1 \langle Y := 1, \sigma[5/X] \rangle.$$

Of course, as this example makes clear, if we consider continuing the execution, we need some way to represent the fact that the command is empty. A configuration with no command left to execute can be represented by a state standing alone. So continuing the execution above we obtain

$$\langle X := 5; Y := 1, \sigma \rangle \rightarrow_1 \langle Y := 1, \sigma[5/X] \rangle \rightarrow_1 \sigma[5/X][1/Y].$$

We leave the detailed presentation of rules for the definition of this one-step execution relation to an exercise. But note there is some choice in what is regarded as a single step. If

$$\langle b, \sigma \rangle \rightarrow_1 \langle \mathbf{true}, \sigma \rangle$$

do we wish

$$\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \rightarrow_1 \langle c_0, \sigma \rangle$$

or

$$\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \rightarrow_1 \langle \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle$$

to be a single step? For the language **IMP** these issues are not critical, but they become so in languages where commands can be executed in parallel; then different choices can effect the final states of execution sequences.

Exercise 2.10 Write down a full set of rules for $\rightarrow_1$ on command configurations, so $\rightarrow_1$ stands for a single step in the execution of a command from a particular state, as discussed above. Use command configurations of the form $\langle c, \sigma \rangle$ and σ when there is no more command left to execute. Point out where you have made a choice in the rules between alternative understandings of what constitutes a single step in the execution. (Showing $\langle c, \sigma \rangle \rightarrow_1^* \sigma'$ iff $\langle c, \sigma \rangle \rightarrow \sigma'$ is hard and requires the application of induction principles introduced in the next two chapters.) □

Exercise 2.11 In our language, the evaluation of expressions has no side effects—their evaluation does not change the state. If we were to model side-effects it would be natural to consider instead an evaluation relation of the form

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

where σ' is the state that results from the evaluation of a in original state σ. To introduce side effects into the evaluation of arithmetic expressions of IMP, extend them by adding a construct

$$c \textbf{ resultis } a$$

where c is a command and a is an arithmetic expression. To evaluate such an expression, the command c is first executed and then a evaluated in the changed state. Formalise this idea by first giving the full syntax of the language and then giving it an operational semantics. □

2.7 Further reading

A convincing demonstration of the wide applicability of "structural operational semantics", of which this chapter has given a taste, was first set out by Gordon Plotkin in his lecture notes for a course at Aarhus University, Denmark, in 1981 [81]. A research group under the direction Gilles Kahn at INRIA in Sophia Antipolis, France are currently working on mechanical tools to support semantics in this style; they have focussed on evaluation or execution to a final value or state, so following their lead this particular kind of structural operational semantics is sometimes called "natural semantics" [26, 28, 29]. We shall take up the operational semantics of functional languages, and nondeterminism and parallelism in later chapters, where further references will be presented. More on abstract syntax can be found in Wikström's book [101], Mosses' chapter in [68] and Tennent's book [97].

3 Some principles of induction

Proofs of properties of programs often rely on the application of a proof method, or really a family of proof methods, called induction. The most commonly used forms of induction are mathematical induction and structural induction. These are both special cases of a powerful proof method called well-founded induction.

3.1 Mathematical induction

The natural numbers are built-up by starting from 0 and repeatedly adjoining successors. The natural numbers consist of no more than those elements which are obtained in this way. There is a corresponding proof principle called *mathematical induction.*

Let $P(n)$ be a a property of the natural numbers $n = 0, 1, \cdots$. The principle of mathematical induction says that in order to show $P(n)$ holds for all natural numbers n it is sufficient to show

- $P(0)$ is true
- If $P(m)$ is true then so is $P(m+1)$ for any natural number m.

We can state it more succinctly, using some logical notation, as

$$(P(0)\ \&\ (\forall m \in \omega.\ P(m) \Rightarrow P(m+1)) \Rightarrow \forall n \in \omega.\ P(n).$$

The principle of mathematical induction is intuitively clear: If we know $P(0)$ and we have a method of showing $P(m+1)$ from the assumption $P(m)$ then from $P(0)$ we know $P(1)$, and applying the method again, $P(2)$, and then $P(3)$, and so on. The assertion $P(m)$ is called the *induction hypothesis*, $P(0)$ the *basis* of the induction and $(\forall m \in \omega.\ P(m) \Rightarrow P(m+1))$ the *induction step.*

Mathematical induction shares a feature with all other methods of proof by induction, that the first most obvious choice of induction hypothesis may not work in a proof. Imagine it is required to prove that a property P holds of all the natural numbers. Certainly it is sensible to try to prove this with $P(m)$ as induction hypothesis. But quite often proving the induction step $\forall m \in \omega.\ (P(m) \Rightarrow P(m+1))$ is impossible. The rub can come in proving $P(m+1)$ from the assumption $P(m)$ because the assumption $P(m)$ is not strong enough. The way to tackle this is to strengthen the induction hypothesis to a property $P'(m)$ which implies $P(m)$. There is an art in finding $P'(m)$ however, because in proving the induction step, although we have a stronger assumption $P'(m)$, it is at the cost of having more to prove in $P'(m+1)$ which may be unnecessarily difficult, or impossible.

In showing a property $Q(m)$ holds inductively of all numbers m, it might be that the property's truth at $m+1$ depends not just on its truth at the predecessor m but on

its truth at other numbers preceding m as well. It is sensible to strengthen $Q(m)$ to an induction hypothesis $P(m)$ standing for $\forall k < m.\ Q(k)$. Taking $P(m)$ to be this property in the statement of ordinary mathematical induction we obtain

$$\forall k < 0.\ Q(k)$$

for the basis, and

$$\forall m \in \omega.((\forall k < m.\ Q(k)) \Rightarrow (\forall k < m+1.\ Q(k)))$$

for the induction step. However, the basis is vacuously true—there are no natural numbers strictly below 0, and the step is equivalent to

$$\forall m \in \omega.(\forall k < m.\ Q(k)) \Rightarrow Q(m).$$

We have obtained *course-of-values induction* as a special form of mathematical induction:

$$(\forall m \in \omega.(\forall k < m.\ Q(k)) \Rightarrow Q(m)) \Rightarrow \forall n \in \omega.\ Q(n).$$

Exercise 3.1 Prove by mathematical induction that the following property P holds for all natural numbers:

$$P(n) \iff_{def} \Sigma_{i=1}^{n}(2i-1) = n^2.$$

(The notation $\Sigma_{i=k}^{l} s_i$ abbreviates $s_k + s_{k+1} + \cdots + s_l$ when k, l are integers with $k < l$.) □

Exercise 3.2 A string is a sequence of symbols. A string $a_1 a_2 \cdots a_n$ with n positions occupied by symbols is said to have *length* n. A string can be empty in which case it is said to have length 0. Two strings s and t can be concatenated to form the string st. Use mathematical induction to show there is no string u which satisfies $au = ub$ for two distinct symbols a and b. □

3.2 Structural induction

We would like a technique to prove "obvious" facts like

$$\langle a, \sigma \rangle \to m \ \&\ \langle a, \sigma \rangle \to m' \Rightarrow m = m'$$

for all arithmetic expressions a, states σ and numbers m, m'. It says the evaluation of arithmetic expressions in **IMP** is *deterministic*. The standard tool is the principle of *structural induction*. We state it for arithmetic expressions but of course it applies more generally to all the syntactic sets of our language **IMP**.

Let $P(a)$ be a property of arithmetic expressions a. To show $P(a)$ holds for all arithmetic expressions a it is sufficient to show:

- For all numerals m it is the case that $P(m)$ holds.
- For all locations X it is the case that $P(X)$ holds.
- For all arithmetic expressions a_0 and a_1, if $P(a_0)$ and $P(a_1)$ hold then so does $P(a_0 + a_1)$.
- For all arithmetic expressions a_0 and a_1, if $P(a_0)$ and $P(a_1)$ hold then so does $P(a_0 - a_1)$.
- For all arithmetic expressions a_0 and a_1, if $P(a_0)$ and $P(a_1)$ hold then so does $P(a_0 \times a_1)$.

The assertion $P(a)$ is called the *induction hypothesis*. The principle says that in order to show the induction hypothesis is true of all arithmetic expressions it suffices to show that it is true of atomic expressions and is preserved by all the methods of forming arithmetic expressions. Again this principle is intuitively obvious as arithmetic expressions are precisely those built-up according to the cases above. It can be stated more compactly using logical notation:

$$
\begin{array}{l}
(\forall m \in \mathbf{N}.\ P(m)) \ \&\ (\forall X \in \mathbf{Loc}.P(X)) \ \&\\
(\forall a_0, a_1 \in \mathbf{Aexp}.\ P(a_0)\ \&\ P(a_1) \Rightarrow P(a_0 + a_1)) \ \&\\
(\forall a_0, a_1 \in \mathbf{Aexp}.\ P(a_0)\ \&\ P(a_1) \Rightarrow P(a_0 - a_1)) \ \&\\
(\forall a_0, a_1 \in \mathbf{Aexp}.\ P(a_0)\ \&\ P(a_1) \Rightarrow P(a_0 \times a_1))\\
\Rightarrow\\
\forall a \in \mathbf{Aexp}.\ P(a).
\end{array}
$$

In fact, as is clear, the conditions above not only imply $\forall a \in \mathbf{Aexp}.\ P(a)$ but also are equivalent to it.

Sometimes a degenerate form of structural induction is sufficient. An argument by cases on the structure of expressions will do when a property is true of all expressions simply by virtue of the different forms expressions can take, without having to use the fact that the property holds for subexpressions. An argument by cases on arithmetic expressions uses the fact that if

$$
\begin{array}{l}
(\forall m \in \mathbf{N}.\ P(m))\&\\
(\forall X \in \mathbf{Loc}.P(X)) \ \&\\
(\forall a_0, a_1 \in \mathbf{Aexp}.\ P(a_0 + a_1)) \ \&\\
(\forall a_0, a_1 \in \mathbf{Aexp}.\ P(a_0 - a_1)) \ \&\\
(\forall a_0, a_1 \in \mathbf{Aexp}.\ P(a_0 \times a_1))
\end{array}
$$

then $\forall a \in \mathbf{Aexp}.\ P(a)$.

As an example of how to do proofs by structural induction we prove that the evaluation of arithmetic expression is deterministic.

Proposition 3.3 *For all arithmetic expressions a, states σ and numbers m, m'*

$$\langle a, \sigma\rangle \to m \ \&\ \langle a, \sigma\rangle \to m' \Rightarrow m = m'.$$

Proof: We proceed by structural induction on arithmetic expressions a using the induction hypothesis $P(a)$ where

$$P(a) \text{ iff } \forall \sigma, m, m'.\ (\langle a, \sigma\rangle \to m \ \&\ \langle a, \sigma\rangle \to m' \Rightarrow m = m').$$

For brevity we shall write $\langle a, \sigma\rangle \to m, m'$ for $\langle a, \sigma\rangle \to m$ and $\langle a, \sigma\rangle \to m'$. Using structural induction the proof splits into cases according to the structure of a:
$a \equiv n$: If $\langle a, \sigma\rangle \to m, m'$ then there is only one rule for the evaluation of numbers so $m = m' = n$.
$a \equiv a_0 + a_1$: If $\langle a, \sigma\rangle \to m, m'$ then considering the form of the single rule for the evaluation of sums there must be m_0, m_1 so

$$\langle a_0, \sigma\rangle \to m_0 \text{ and } \langle a_1, \sigma\rangle \to m_1 \text{ with } m = m_0 + m_1$$

as well as m_0', m_1' so

$$\langle a_0, \sigma\rangle \to m_0' \text{ and } \langle a_1, \sigma\rangle \to m_1' \text{ with } m' = m_0' + m_1'$$

By the induction hypothesis applied to a_0 and a_1 we obtain $m_0 = m_0'$ and $m_1 = m_1'$. Thus $m = m_0 + m_1 = m_0' + m_1' = m'$.

The remaining cases follow in a similar way. We can conclude, by the principle of structural induction, that $P(a)$ holds for all $a \in \mathbf{Aexp}$. □

One can prove the evaluation of expressions always terminates by structural induction, and corresponding facts about boolean expressions.

Exercise 3.4 Prove by structural induction that the evaluation of arithmetic expressions always terminates, *i.e.*, for all arithmetic expression a and states σ there is some m such that $\langle a, \sigma\rangle \to m$. □

Exercise 3.5 Using these facts about arithmetic expressions, by structural induction, prove the evaluation of boolean expressions is firstly deterministic, and secondly total. □

Exercise 3.6 What goes wrong when you try to prove the execution of commands is deterministic by using structural induction on commands? (Later, in Section 3.4, we shall give a proof using "structural induction" on derivations.) □

3.3 Well-founded induction

Mathematical and structural induction are special cases of a general and powerful proof principle called well-founded induction. In essence structural induction works because breaking down an expression into subexpressions can not go on forever, eventually it must lead to atomic expressions which can not be broken down any further. If a property fails to hold of any expression then it must fail on some minimal expression which when it is broken down yields subexpressions, all of which satisfy the property. This observation justifies the principle of structural induction: to show a property holds of all expressions it is sufficient to show that a property holds of an arbitrary expression if it holds of all its subexpressions. Similarly with the natural numbers, if a property fails to hold of all natural numbers then there has to be a smallest natural number at which it fails. The essential feature shared by both the subexpression relation and the predecessor relation on natural numbers is that do not give rise to infinite descending chains. This is the feature required of a relation if it is to support well-founded induction.

Definition: A *well-founded relation* is a binary relation $\prec$ on a set A such that there are no infinite descending chains $\cdots \prec a_i \prec \cdots \prec a_1 \prec a_0$. When $a \prec b$ we say a is a *predecessor* of b.

Note a well-founded relation is necessarily *irreflexive i.e.*, for no a do we have $a \prec a$, as otherwise there would be the infinite decending chain $\cdots \prec a \prec \cdots \prec a \prec a$. We shall generally write $\preceq$ for the reflexive closure of the relation $\prec$, *i.e.*

$$a \preceq b \iff a = b \text{ or } a \prec b.$$

Sometimes one sees an alternative definition of well-founded relation, in terms of minimal elements.

Proposition 3.7 *Let $\prec$ be a binary relation on a set A. The relation $\prec$ is well-founded iff any nonempty subset Q of A has a minimal element, i.e. an element m such that*

$$m \in Q \;\&\; \forall b \prec m.\, b \notin Q.$$

Proof:
"if": Suppose every nonempty subset of A has a minimal element. If $\cdots \prec a_i \prec \cdots \prec a_1 \prec a_0$ were an infinite descending chain then the set $Q = \{a_i \mid i \in \omega\}$ would be nonempty without a minimal element, a contradiction. Hence $\prec$ is well-founded.
"only if": To see this, suppose Q is a nonempty subset of A. Construct a chain of elements as follows. Take a_0 to be any element of Q. Inductively, assume a chain of

elements $a_n \prec \cdots \prec a_0$ has been constructed inside Q. Either there is some $b \prec a_n$ such that $b \in Q$ or there is not. If not stop the construction. Otherwise take $a_{n+1} = b$. As $\prec$ is well-founded the chain $\cdots \prec a_i \prec \cdots \prec a_1 \prec a_0$ cannot be infinite. Hence it is finite, of the form $a_n \prec \cdots \prec a_0$ with $\forall b \prec a_n.\ b \notin Q$. Take the required minimal element m to be a_n. □

Exercise 3.8 Let $\prec$ be a well-founded relation on a set B. Prove

1. its transitive closure $\prec^+$ is also well-founded,
2. its reflexive, transitive closure $\prec^*$ is a partial order.

□

The principle of well-founded induction.

Let $\prec$ be a well founded relation on a set A. Let P be a property. Then $\forall a \in A.\ P(a)$ iff

$$\forall a \in A.\ ([\forall b \prec a.\ P(b)] \Rightarrow P(a)).$$

The principle says that to prove a property holds of all elements of a well-founded set it suffices to show that if the property holds of all predecessors of an arbitrary element a then the property holds of a.

We now prove the principle. The proof rests on the observation that any nonempty subset Q of a set A with a well-founded relation $\prec$ has a minimal element. Clearly if $P(a)$ holds for all elements of A then $\forall a \in A.\ ([\forall b \prec a.\ P(b)] \Rightarrow P(a))$. To show the converse, we assume $\forall a \in A.\ ([\forall b \prec a.\ P(b)] \Rightarrow P(a))$ and produce a contradiction by supposing $\neg P(a)$ for some $a \in A$. Then, as we have observed, there must be a minimal element m of the set $\{a \in A \mid \neg P(a)\}$. But then $\neg P(m)$ and yet $\forall b \prec m.\ P(b)$, which contradicts the assumption.

In mathematics this principle is sometimes called *Noetherian induction* after the algebraist Emmy Noether. Unfortunately, in some computer science texts (*e.g.* [59]) it is misleadingly called "structural induction".

Example: If we take the relation $\prec$ to be the successor relation

$$n \prec m \text{ iff } m = n + 1$$

on the non-negative integers the principle of well-founded induction specialises to mathematical induction. □

Example: If we take $\prec$ to be the "strictly less than" relation $<$ on the non-negative integers, the principle specialises to course-of-values induction. □

Example: If we take $\prec$ to be the relation between expressions such that $a \prec b$ holds iff a is an immediate subexpression of b we obtain the principle of structural induction as a special case of well-founded induction. □

Proposition 3.7 provides an alternative to proofs by well-founded induction. Suppose A is a well-founded set. Instead of using well-founded induction to show every element of A satisfies a property P, we can consider the subset of A for which the property P fails, *i.e.* the subset F of counterexamples. By Proposition 3.7, to show F is $\emptyset$ it is sufficient to show that F cannot have a minimal element. This is done by obtaining a contradiction from the assumption that there is a minimal element in F. (See the proof of Proposition 3.12 for an example of this approach.) Whether to use this approach or the principle of well-founded induction is largely a matter of taste, though sometimes, depending on the problem, one approach can be more direct than the other.

Exercise 3.9 For suitable well-founded relation on strings, use the "no counterexample" approach described above to show there is no string u which satisfies $au = ub$ for two distinct symbols a and b. Compare your proof with another by well-founded induction (and with the proof by mathematical induction asked for in Section 3.1). □

Proofs can often depend on a judicious choice of well-founded relation. In Chapter 10 we shall give some useful ways of constructing well-founded relations.

As an example of how the operational semantics supports proofs we show that Euclid's algorithm for the gcd (greatest common divisor) of two non-negative numbers terminates. Though such proofs are often less clumsy when based on a denotational semantics. (Later, Exercise 6.16 will show its correctness.) Euclid's algorithm for the greatest common divisor of two positive integers can be written in **IMP** as:

$$\begin{aligned}\text{Euclid} \equiv \textbf{while } \neg(M = N) \textbf{ do}\\ \textbf{if } M \leq N\\ \textbf{then } N := N - M\\ \textbf{else } M := M - N\end{aligned}$$

Theorem 3.10 *For all states* σ

$$\sigma(M) \geq 1 \;\&\; \sigma(N) \geq 1 \Rightarrow \exists \sigma'.\ \langle \text{Euclid}, \sigma\rangle \rightarrow \sigma'.$$

Proof: We wish to show the property

$$P(\sigma) \iff \exists \sigma'. \langle \text{Euclid}, \sigma\rangle \rightarrow \sigma'.$$

holds for all states σ in $S = \{\sigma \in \Sigma \mid \sigma(M) \geq 1 \ \& \ \sigma(N) \geq 1\}$.

We do this by well-founded induction on the relation $\prec$ on S where

$$\sigma' \prec \sigma \text{ iff } (\sigma'(M) \leq \sigma(M) \ \& \ \sigma'(N) \leq \sigma(N)) \ \& \\ (\sigma'(M) \neq \sigma(M) \text{ or } \sigma'(N) \neq \sigma(N))$$

for states σ', σ in S. Clearly $\prec$ is well-founded as the values in M and N cannot be decreased indefinitely and remain positive.

Let $\sigma \in S$. Suppose $\forall \sigma' \prec \sigma.\ P(\sigma')$. Abbreviate $\sigma(M) = m$ and $\sigma(N) = n$.

If $m = n$ then $\langle \neg(M = N), \sigma \rangle \rightarrow \mathbf{false}$. Using its derivation we construct the derivation

$$\dfrac{\dfrac{\vdots}{\langle \neg(M = N), \sigma \rangle \rightarrow \mathbf{false}}}{\langle \text{Euclid}, \sigma \rangle \rightarrow \sigma}$$

using the rule for while-loops which applies when the boolean condition evaluates to false. In the case where $m = n$, $\langle \text{Euclid}, \sigma \rangle \rightarrow \sigma$.

Otherwise $m \neq n$. In this case $\langle \neg(M = N), \sigma \rangle \rightarrow \mathbf{true}$. From the rules for the execution of commands we derive

$$\langle \mathbf{if}\ M \leq N\ \mathbf{then}\ N := N - M\ \mathbf{else}\ M := M - N,\ \sigma \rangle \rightarrow \sigma''$$

where

$$\sigma'' = \begin{cases} \sigma[n - m/N] & \text{if } m < n \\ \sigma[m - n/M] & \text{if } n < m. \end{cases}$$

In either case $\sigma'' \prec \sigma$. Hence $P(\sigma'')$ so $\langle \text{Euclid}, \sigma'' \rangle \rightarrow \sigma'$ for some σ'. Thus applying the other rule for while-loops we obtain

$$\dfrac{\dfrac{\vdots}{\langle \neg(M = N), \sigma \rangle \rightarrow \mathbf{true}} \qquad \dfrac{\dfrac{\vdots}{\langle \mathbf{if}\ M \leq N\ \mathbf{then}\ N := N - M\ \mathbf{else}\ M := M - N, \sigma \rangle \rightarrow \sigma''} \qquad \dfrac{\vdots}{\langle \text{Euclid}, \sigma'' \rangle \rightarrow \sigma'}}{\langle \text{Euclid}, \sigma \rangle \rightarrow \sigma'}}$$

a derivation of $\langle \text{Euclid}, \sigma \rangle \rightarrow \sigma'$. Therefore $P(\sigma)$.

By well-founded induction we conclude $\forall \sigma \in S.\ P(\sigma)$, as required. □

Well-founded induction is the most important principle in proving the termination of programs. Uncertainties about termination arise because of loops or recursions in a program. If it can be shown that execution of a loop or recursion in a program decreases the value in a well-founded set then it must eventually terminate.

3.4 Induction on derivations

Structural induction alone is often inadequate to prove properties of operational semantics. Often it is useful to do induction on the structure of derivations. Putting this on a firm basis involves formalising some of the ideas met in the last chapter.

Possible derivations are determined by means of rules. Instances of rules have the form

$$\frac{}{x} \quad \text{or} \quad \frac{x_1, \ldots, x_n}{x},$$

where the former is an axiom with an empty set of premises and a conclusion x, while the latter has $\{x_1, \ldots, x_n\}$ as its set of premises and x as its conclusion. The rules specify how to construct derivations, and through these define a set. The set defined by the rules consists precisely of those elements for which there is a derivation. A derivation of an element x takes the form of a tree which is either an instance of an axiom

$$\frac{}{x}$$

or of the form

$$\frac{\begin{array}{c}\vdots\\ x_1\end{array}, \ldots, \begin{array}{c}\vdots\\ x_n\end{array}}{x}$$

which includes derivations of $x_1, \ldots, x_n$, the premises of a rule instance with conclusion x. In such a derivation we think of $\begin{array}{c}\vdots\\ x_1\end{array}, \cdots, \begin{array}{c}\vdots\\ x_n\end{array}$ as subderivations of the larger derivation of x.

Rule instances are got from rules by substituting actual terms or values for metavariables in them. All the rules we are interested in are *finitary* in that their premises are finite. Consequently, all rule instances have a finite, possibly empty set of premises and a conclusion. We start a formalisation of derivations from the idea of a set of rule instances.

A *set of rule instances* R consists of elements which are pairs (X/y) where X is a finite set and y is an element. Such a pair (X/y) is called a *rule instance* with *premises* X and conclusion y.

We are more used to seeing rule instances (X/y) as

$$\frac{}{y} \text{ if } X = \emptyset, \text{ and as } \frac{x_1, \cdots, x_n}{y} \text{ if } X = \{x_1, \cdots, x_n\}.$$

Assume a set of rule instances R. An *R-derivation* of y is either a rule instance $(\emptyset/y)$ or a pair $(\{d_1, \cdots, d_n\}/y)$ where $(\{x_1, \cdots, x_n\}/y)$ is a rule instance and d_1 is an R-derivation

of $x_1, \ldots, d_n$ is an R-derivation of x_n. We write $d \Vdash_R y$ to mean d is an R-derivation of y. Thus

$(\emptyset/y) \Vdash_R y$ if $(\emptyset/y) \in R$, and
$(\{d_1, \cdots, d_n\}/y) \Vdash_R y$ if $(\{x_1, \cdots, x_n\}/y) \in R$ & $d_1 \Vdash_R x_1$ & $\cdots$ & $d_n \Vdash_R x_n$.

We say y is derived from R if there is an R-derivation of y, *i.e.* $d \Vdash_R y$ for some derivation d. We write $\Vdash_R y$ to mean y is derived from R. When the rules are understood we shall write just $d \Vdash y$ and $\Vdash y$.

In operational semantics the premises and conclusions are tuples. There,

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

meaning $\langle c, \sigma \rangle \rightarrow \sigma'$ is derivable from the operational semantics of commands, is customarily written as just $\langle c, \sigma \rangle \rightarrow \sigma'$. It is understood that $\langle c, \sigma \rangle \rightarrow \sigma'$ includes, as part of its meaning, that it is derivable. We shall only write $\Vdash \langle c, \sigma \rangle \rightarrow \sigma'$ when we wish to emphasise that there is a derivation.

Let d, d' be derivations. Say d' is an *immediate subderivation* of d, written $d' \prec_1 d$, iff d has the form (D/y) with $d' \in D$. Write $\prec$ for the transitive closure of $\prec_1$, *i.e.* $\prec = \prec_1^+$. We say d' is a proper subderivation of d iff $d' \prec d$.

Because derivations are finite, both relations of being an immediate subderivation $\prec_1$ and that of being a proper subderivation are well-founded. This fact can be used to show the execution of commands is deterministic.

Theorem 3.11 *Let c be a command and σ_0 a state. If $\langle c, \sigma_0 \rangle \rightarrow \sigma_1$ and $\langle c, \sigma_0 \rangle \rightarrow \sigma$, then $\sigma = \sigma_1$, for all states σ, σ_1.*

Proof: The proof proceeds by well-founded induction on the proper subderivation relation $\prec$ between derivations for the execution of commands. The property we shall show holds of all such derivations d is the following:

$$P(d) \iff \forall c \in \mathbf{Com}, \sigma_0, \sigma, \sigma_1, \in \Sigma.\ d \Vdash \langle c, \sigma_0 \rangle \rightarrow \sigma \ \& \ \langle c, \sigma_0 \rangle \rightarrow \sigma_1 \Rightarrow \sigma = \sigma_1.$$

By the principle of well-founded induction, it suffices to show $\forall d' \prec d.\ P(d')$ implies $P(d)$.

Let d be a derivation from the operational semantics of commands. Assume $\forall d' \prec d.\ P(d')$. Suppose

$$d \Vdash \langle c, \sigma_0 \rangle \rightarrow \sigma \text{ and } \Vdash \langle c, \sigma_0 \rangle \rightarrow \sigma_1.$$

Then $d_1 \Vdash \langle c, \sigma_0 \rangle \rightarrow \sigma_1$ for some d_1.

Now we show by cases on the structure of c that $\sigma = \sigma_1$.

$c \equiv \textbf{skip}$: In this case

$$d = d_1 = \frac{}{\langle \textbf{skip}, \sigma_0 \rangle \to \sigma_0}.$$

$c \equiv X := a$: Both derivations have a similar form:

$$d = \frac{\vdots}{\dfrac{\langle a, \sigma_0 \rangle \to m}{\langle X := a, \sigma_0 \rangle \to \sigma_0[m/X]}} \qquad d_1 = \frac{\vdots}{\dfrac{\langle a, \sigma_0 \rangle \to m_1}{\langle X := a, \sigma_0 \rangle \to \sigma_0[m_1/X]}}$$

where $\sigma = \sigma_0[m/X]$ and $\sigma_1 = \sigma_0[m_1/X]$. As the evaluation of arithmetic expressions is deterministic $m = m_1$, so $\sigma = \sigma_1$.

$c \equiv c_0; c_1$: In this case

$$d = \frac{\dfrac{\vdots}{\langle c_0, \sigma_0 \rangle \to \sigma'} \quad \dfrac{\vdots}{\langle c_1, \sigma' \rangle \to \sigma}}{\langle c_0; c_1, \sigma_0 \rangle \to \sigma} \qquad d_1 = \frac{\dfrac{\vdots}{\langle c_0, \sigma_0 \rangle \to \sigma_1'} \quad \dfrac{\vdots}{\langle c_1, \sigma_1' \rangle \to \sigma_1}}{\langle c_0; c_1, \sigma_0 \rangle \to \sigma_1}.$$

Let d^0 be the subderivation

$$\frac{\vdots}{\langle c_0, \sigma_0 \rangle \to \sigma'}$$

and d^1 the subderivation

$$\frac{\vdots}{\langle c_1, \sigma' \rangle \to \sigma}$$

in d. Then $d^0 \prec d$ and $d^1 \prec d$, so $P(d^0)$ and $P(d^1)$. It follows that $\sigma' = \sigma_1'$, and $\sigma = \sigma_1$ (why?).

$c \equiv \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1$: The rule for conditionals which applies in this case is determined by how the boolean b evaluates. By the exercises of Section 3.2, its evaluation is deterministic so either $\langle b, \sigma_0 \rangle \to \textbf{true}$ or $\langle b, \sigma_0 \rangle \to \textbf{false}$, but not both.

When $\langle b, \sigma_0 \rangle \to \textbf{true}$ we have:

$$d = \frac{\dfrac{\vdots}{\langle b, \sigma_0 \rangle \to \textbf{true}} \quad \dfrac{\vdots}{\langle c_0, \sigma_0 \rangle \to \sigma}}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma_0 \rangle \to \sigma} \qquad d_1 = \frac{\dfrac{\vdots}{\langle b, \sigma_0 \rangle \to \textbf{true}} \quad \dfrac{\vdots}{\langle c_0, \sigma_0 \rangle \to \sigma_1}}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma_0 \rangle \to \sigma_1}.$$

Let d' be the subderivation of $\langle c_0, \sigma_0\rangle \to \sigma$ in d. Then $d' \prec d$. Hence $P(d')$. Thus $\sigma = \sigma_1$. When $\langle b, \sigma_0\rangle \to \mathbf{false}$ the argument is similar.

$c \equiv \mathbf{while}\ b\ \mathbf{do}\ c$: The rule for while-loops which applies is again determined by how b evaluates. Either $\langle b, \sigma_0\rangle \to \mathbf{true}$ or $\langle b, \sigma\rangle \to \mathbf{false}$, but not both.

When $\langle b, \sigma_0\rangle \to \mathbf{false}$ we have :

$$d = \frac{\dfrac{\vdots}{\langle b, \sigma_0\rangle \to \mathbf{false}}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma_0\rangle \to \sigma_0} \qquad d_1 = \frac{\dfrac{\vdots}{\langle b, \sigma_0\rangle \to \mathbf{false}}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma_0\rangle \to \sigma_0}$$

so certainly $\sigma = \sigma_0 = \sigma_1$.

When $\langle b, \sigma_0\rangle \to \mathbf{true}$ we have:

$$d = \frac{\dfrac{\vdots}{\langle b, \sigma_0\rangle \to \mathbf{true}} \quad \dfrac{\vdots}{\langle c, \sigma_0\rangle \to \sigma'} \quad \dfrac{\vdots}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma'\rangle \to \sigma}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma_0\rangle \to \sigma}$$

$$d_1 = \frac{\dfrac{\vdots}{\langle b, \sigma_0\rangle \to \mathbf{true}} \quad \dfrac{\vdots}{\langle c, \sigma_0\rangle \to \sigma_1'} \quad \dfrac{\vdots}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma_1'\rangle \to \sigma_1}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma_0\rangle \to \sigma_1}$$

Let d' be the subderivation of $\langle c, \sigma_0\rangle \to \sigma'$ and d'' the subderivation of $\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma'\rangle \to \sigma$ in d. Then $d' \prec d$ and $d'' \prec d$ so $P(d')$ and $P(d'')$. It follows that $\sigma' = \sigma_1'$, and subsequently that $\sigma = \sigma_1$.

In all cases of c we have shown $d \Vdash \langle c, \sigma_0\rangle \to \sigma$ and $\langle c, \sigma_0\rangle \to \sigma_1$ implies $\sigma = \sigma_1$.

By the principle of well-founded induction we conclude that $P(d)$ holds for all derivations d for the execution of commands. This is equivalent to

$$\forall c \in \mathbf{Com}, \sigma_0, \sigma, \sigma_1, \in \Sigma.\ \langle c, \sigma_0\rangle \to \sigma\ \&\ \langle c, \sigma_0\rangle \to \sigma_1 \Rightarrow \sigma = \sigma_1,$$

which proves the theorem. □

As was remarked, Proposition 3.7 provides an alternative to proofs by well-founded induction. Induction on derivations is a special kind of well-founded induction used to prove a property holds of all derivations. Instead, we can attempt to produce a contradiction from the assumption that there is a minimal derivation for which the property is false. The approach is illustrated below:

Proposition 3.12 *For all states σ, σ',*

$$\langle \textbf{while true do skip}, \sigma \rangle \not\to \sigma'.$$

Proof: Abbreviate $w \equiv$ **while true do skip**. Suppose $\langle w, \sigma \rangle \to \sigma'$ for some states σ, σ'. Then there is a minimal derivation d such that $\exists \sigma, \sigma' \in \Sigma.\ d \Vdash \langle w, \sigma \rangle \to \sigma'$. Only one rule can be the final rule of d, making d of the form:

$$d = \dfrac{\dfrac{\vdots}{\langle \textbf{true}, \sigma \rangle \to \textbf{true}} \quad \dfrac{\vdots}{\langle c, \sigma \rangle \to \sigma''} \quad \dfrac{\vdots}{\langle \textbf{while true do } c, \sigma'' \rangle \to \sigma'}}{\langle \textbf{while true do } c, \sigma \rangle \to \sigma'}$$

But this contains a proper subderivation $d' \Vdash \langle w, \sigma \rangle \to \sigma'$, contradicting the minimality of d. □

3.5 Definitions by induction

Techniques like structural induction are often used to define operations on the set defined. Integers and arithmetic expressions share a common property, that of being built-up in a unique way. An integer is either zero or the successor of a unique integer, while an arithmetic expression is either atomic or a sum, or product *etc.* of a unique pair of expressions. It is by virtue of their being built up in a unique way that we can can make definitions by induction on integers and expressions. For example to define the length of an expression it is natural to define it in terms of the lengths of its components. For arithmetic expressions we can define

$$\begin{aligned} &\mathrm{length}(n) = \mathrm{length}(X) = 1, \\ &\mathrm{length}(a_0 + a_1) = 1 + \mathrm{length}(a_0) + \mathrm{length}(a_1), \\ &\ldots \end{aligned}$$

For future reference we define $\mathrm{loc}_L(c)$, the set of those locations which appear on the left of an assignment in a command. For a command c, the function $\mathrm{loc}_L(c)$ is defined by structural induction by taking

$$\begin{aligned} \mathrm{loc}_L(\textbf{skip}) &= \emptyset, & \mathrm{loc}_L(X := a) &= \{X\}, \\ \mathrm{loc}_L(c_0; c_1) &= \mathrm{loc}_L(c_0) \cup \mathrm{loc}_L(c_1), & \mathrm{loc}_L(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) &= \mathrm{loc}_L(c_0) \cup \mathrm{loc}_L(c_1), \\ \mathrm{loc}_L(\textbf{while } b \textbf{ do } c) &= \mathrm{loc}_L(c). \end{aligned}$$

In a similar way one defines operations on the natural numbers by mathematical induction and operations defined on sets given by rules. In fact the proof of Proposition 3.7,

that every nonempty subset of a well-founded set has a minimal element, contains an implicit use of definition by induction on the natural numbers to construct a chain with a minimal element in the nonempty set.

Both definition by structural induction and definition by mathematical induction are special cases of definition by well-founded induction, also called *well-founded recursion*. To understand this name, notice that both definition by induction and structural induction allow a form of recursive definition. For example, the length of an arithmetic expression could have been defined in this manner:

$$\text{length}(a) = \begin{cases} 1 & \text{if } a \equiv n, \text{ a number} \\ \text{length}(a_0) + \text{length}(a_1) & \text{if } a \equiv (a_0 + a_1), \\ \vdots & \end{cases}$$

How the length function acts on a particular argument, like $(a_0 + a_1)$ is specified in terms of how the length function acts on other arguments, like a_0 and a_1. In this sense the definition of the length function is defined recursively in terms of itself. However this recursion is done in such a way that the value on a particular argument is only specified in terms of strictly smaller arguments. In a similar way we are entitled to define functions on an arbitrary well-founded set. The general principle is more difficult to understand, resting as it does on some relatively sophisticated constructions on sets, and for this reason its full treatment is postponed to Section 10.4. (Although the material won't be needed until then, the curious or impatient reader might care to glance ahead. Despite its late appearance that section does not depend on any additional concepts.)

Exercise 3.13 Give definitions by structural induction of $\text{loc}(a)$, $\text{loc}(b)$ and $\text{loc}_R(c)$, the sets of locations which appear in arithmetic expressions a, boolean expressions b and the right-hand sides of assignments in commands c. □

3.6 Further reading

The techniques and ideas discussed in this chapter are well-known, basic techniques within mathematical logic. As operational semantics follows the lines of natural deduction, it is not surprising that it shares basic techniques with proof theory, as presented in [84] for example—derivations are really a simple kind of proof. For a fairly advanced, though accessible, account of proof theory with a computer science slant see [51, 40], which contains much more on notations for proofs (and so derivations). Further explanation and uses of well-founded induction can be found in [59] and [21], where it is called "structural induction", in [58] and [73]), and here, especially in Chapter 10.

4 Inductive definitions

This chapter is an introduction to the theory of inductively defined sets, of which presentations of syntax and operational semantics are examples. Sets inductively defined by rules are shown to be the least sets closed under the rules. As such, a principle of induction, called rule induction, accompanies the constructions. It specialises to proof rules for reasoning about the operational semantics of IMP.

4.1 Rule induction

We defined the syntactic set of arithmetic expressions **Aexp** as the set obtained from the formation rules for arithmetic expressions. We have seen there is a corresponding induction principle, that of structural induction on arithmetic expressions. We have defined the operational semantics of while-programs by defining evaluation and execution relations as relations given by rules which relate evaluation or execution of terms to the evaluation or execution of their components. For example, the evaluation relation on arithmetic expressions was defined by the rules of Section 2.2 as a ternary relation which is the set consisting of triples (a, σ, n) of $\mathbf{Aexp} \times \Sigma \times \mathrm{N}$ such that $\langle a, \sigma \rangle \to n$. There is a corresponding induction principle which we can see as a special case of a principle we call rule induction.

We are interested in defining a set by rules. Viewed abstractly, instances of rules have the form $(\emptyset/x)$ or $(\{x_1, \ldots, x_n\}/x)$. Given a set of rule instances R, we write I_R for the set defined by R consisting of precisely of those elements x for which there is a derivation. Put another way

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

The principle of rule induction is useful to show a property is true of all the elements in a set defined by some rules. It is based on the idea that if a property is preserved in moving from the premises to the conclusion of all rule instances in a derivation then the conclusion of the derivation has the property, so the property is true of all elements in the set defined by the rules.

The general principle of rule induction

Let I_R be defined by rule instances R. Let P be a property. Then $\forall x \in I_R.\ P(x)$ iff for all rule instances (X/y) in R for which $X \subseteq I_R$

$$(\forall x \in X.\ P(x)) \Rightarrow P(y).$$

Notice for rule instances of the form (X/y), with $X = \emptyset$, the last condition is equivalent to $P(y)$. Certainly then $\forall x \in X.\ x \in I_R\ \&\ P(x)$ is vacuously true because any x in $\emptyset$

satisfies P—there are none. The statement of rule induction amounts to the following. For rule instances R, we have $\forall y \in I_R.\ P(y)$ iff for all instances of axioms

$$\frac{}{x}$$

$P(x)$ is true, and for all rule instances

$$\frac{x_1, \ldots, x_n}{x}$$

if $x_k \in I_R$ & $P(x_k)$ is true for all the premises, when k ranges from 1 to n, then $P(x)$ is true of the conclusion.

The principle of rule induction is fairly intuitive. It corresponds to a superficially different, but equivalent method more commonly employed in mathematics. (This observation will also lead to a proof of the validity of rule induction.) We say a set Q is *closed* under rule instances R, or simply *R-closed*, iff for all rule instances (X/y)

$$X \subseteq Q \Rightarrow y \in Q.$$

In other words, a set is closed under the rule instances if whenever the premises of any rule instance lie in the set so does its conclusion. In particular, an R-closed set must contain all the instances of axioms. The set I_R is the least set closed under R in this sense:

Proposition 4.1 *With respect to rule instances R*
(i) *I_R is R-closed, and*
(ii) *if Q is an R-closed set then $I_R \subseteq Q$.*

Proof:
(i) It is easy to see I_R is closed under R. Suppose (X/y) is an instance of a rule in R and that $X \subseteq I_R$. Then from the definition of I_R there are derivations of each element of X. If X is nonempty these derivations can be combined with the rule instance (X/y) to provide a derivation of y, and, otherwise, $(\emptyset/y)$ provides a derivation immediately. In either case we obtain a derivation of y which must therefore be in I_R too. Hence I_R is closed under R.
(ii) Suppose that Q is R-closed. We want to show $I_R \subseteq Q$. Any element of I_R is the conclusion of some derivation. But any derivation is built out of rule instances (X/y). If the premises X are in Q then so is the conclusion y (in particular, the conclusion of any axiom will be in Q). Hence we can work our way down any derivation, starting at

axioms, to show its conclusion is in Q. More formally, we can do an induction on the proper subderivation relation $\prec$ to show

$$\forall y \in I_R.\ d \Vdash_R y \Rightarrow y \in Q$$

for all R-derivations d. Therefore $I_R \subseteq Q$. □

Exercise 4.2 Do the induction on derivations mentioned in the proof above. □

Suppose we wish to show a property P is true of all elements of I_R, the set defined by rules R. The conditions (i) and (ii) in the proposition above furnish a method. Defining the set

$$Q = \{x \in I_R \mid P(x)\},$$

the property P is true of all elements of I_R iff $I_R \subseteq Q$. By condition (ii), to show $I_R \subseteq Q$ it suffices to show that Q is R-closed. This will follow if for all rule instances (X/y)

$$(\forall x \in X.\ x \in I_R\ \&\ P(x)) \Rightarrow P(y)$$

But this is precisely what is required by rule induction to prove the property P holds for all elements of I_R. The truth of this statement is not just sufficient but also necessary to show the property P of all elements of I_R. Suppose $P(x)$ for all $x \in I_R$. Let (X/y) be a rule instance such that

$$\forall x \in X.\ x \in I_R\ \&\ P(x).$$

By (i), saying I_R is R-closed, we get $y \in I_R$, and so that $P(y)$. And in this way we have derived the principle of rule induction from (i) and (ii), saying that I_R is the least R-closed set.

Exercise 4.3 For rule instances R, show

$$\bigcap \{Q \mid Q \text{ is } R\text{-closed}\}$$

is R-closed. What is this set? □

Exercise 4.4 Let the rules consist of $(\emptyset/0)$ and $(\{n\}/(n+1))$ where n is a natural number. What is the set defined by the rules and what is rule induction in this case? □

In presenting rules we have followed the same style as that used in giving operational semantics. When it comes to defining syntactic sets by rules, BNF is the traditional way though it can be done differently. For instance, what is traditionally written as

$$a ::= \cdots \mid a_0 + a_1 \mid \cdots,$$

saying that if a_0 and a_1 are well-formed expressions arithmetic expressions then so is $a_0 + a_1$, could instead be written as

$$\frac{a_0 : \mathbf{Aexp} \quad a_1 : \mathbf{Aexp}}{a_0 + a_1 : \mathbf{Aexp}}.$$

This way of presenting syntax is becoming more usual.

Exercise 4.5 What is rule induction in the case where the rules are the formation rules for **Aexp**? What about when the rules are those for boolean expressions? (Careful! See the next section.) □

4.2 Special rule induction

Thinking of the syntactic sets of boolean expressions and commands it is clear that sometimes a syntactic set is given by rules which involve elements from another syntactic set. For example, the formation rules for commands say how commands can be formed from arithmetic and boolean expressions, as well as other commands. The formation rules

$$c ::= \cdots \mid X := a \mid \cdots \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \cdots,$$

can, for the sake of uniformity, be written as

$$\frac{X : \mathbf{Loc} \quad a : \mathbf{Aexp}}{X := a : \mathbf{Com}} \quad \text{and} \quad \frac{b : \mathbf{Bexp} \quad c_0 : \mathbf{Com} \quad c_1 : \mathbf{Com}}{\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 : \mathbf{Com}}.$$

Rule induction works by showing properties are preserved by the rules. This means that if we are to use rule induction to prove a property of all commands we must make sure that the property covers all arithmetic and boolean expressions as well. As it stands, the principle of rule induction does not instantiate to structural induction on commands, but to a considerably more awkward proof principle, simultaneously combining structural induction on commands with that on arithmetic and boolean expressions. A modified principle of rule induction is required for establishing properties of *subsets* of the set defined by rules.

The special principle of rule induction

Let I_R be defined by rule instances R. Let $A \subseteq I_R$. Let Q be a property. Then $\forall a \in A.\ Q(a)$ iff for all rule instances (X/y) in R, with $X \subseteq I_R$ and $y \in A$,

$$(\forall x \in X \cap A.\ Q(x)) \Rightarrow Q(y).$$

The special principle of rule induction actually follows from the general principle. Let R be a set of rule instances. Let A be a subset of I_R, the set defined by R. Suppose $Q(x)$ is a property we are interested in showing is true of all elements of A. Define a corresponding property $P(x)$ by

$$P(x) \iff (x \in A \Rightarrow Q(x)).$$

Showing $Q(a)$ for all $a \in A$ is equivalent to showing that $P(x)$ is true for all $x \in I_R$. By the general principle of rule induction the latter is equivalent to

$$\forall (X/y) \in R. \quad X \subseteq I_R \;\&\; (\forall x \in X.(x \in A \Rightarrow Q(x))) \Rightarrow (y \in A \Rightarrow Q(y)).$$

But this is logically equivalent to

$$\forall (X/y) \in R. \quad (X \subseteq I_R \;\&\; y \in A \;\&\; (\forall x \in X.(x \in A \Rightarrow Q(x)))) \Rightarrow Q(y).$$

This is equivalent to the condition required by the special principle of rule induction.

Exercise 4.6 Explain how structural induction for commands and booleans follows from the special principle of rule induction. □

Because the special principle follows from the general, any proof using the special principle can be replaced by one using the principle of general rule induction. But in practice use of the special principle can drastically cut down the number of rules to consider, a welcome feature when it comes to considering rule induction for operational semantics.

4.3 Proof rules for operational semantics

Not surprisingly, rule induction can be a useful tool for proving properties of operational semantics presented by rules, though then it generally takes a superficially different form because the sets defined by the rules are sets of tuples. This section presents the special cases of rule induction which we will use later in reasoning about the operational behaviour of **IMP** programs.

4.3.1 Rule induction for arithmetic expressions

The principle of rule induction for the evaluation of arithmetic expressions is got from the rules for their operational semantics. It is an example of rule induction; a property $P(a, \sigma, n)$ is true of all evaluations $\langle a, \sigma \rangle \rightarrow n$ iff it is preserved by the rules for building

up the evaluation relation.

$$\forall a \in \mathbf{Aexp}, \sigma \in \Sigma, n \in \mathbf{N}. \quad \langle a, \sigma \rangle \to n \Rightarrow P(a, \sigma, n)$$
iff
$$[\forall n \in \mathbf{N}, \sigma \in \Sigma.\ P(n, \sigma, n)$$
$$\&$$
$$\forall X \in \mathbf{Loc}, \sigma \in \Sigma.\ P(X, \sigma, \sigma(X))$$
$$\&$$
$$\forall a_0, a_1 \in \mathbf{Aexp}, \sigma \in \Sigma, n_0, n_1 \in \mathbf{N}.$$
$$\langle a_0, \sigma \rangle \to n_0\ \&\ P(a_0, \sigma, n_0)\ \&\ \langle a_1, \sigma \rangle \to n_1\ \&\ P(a_1, \sigma, n_1)$$
$$\Rightarrow P(a_0 + a_1, \sigma, n_0 + n_1)$$
$$\&$$
$$\forall a_0, a_1 \in \mathbf{Aexp}, \sigma \in \Sigma, n_0, n_1 \in \mathbf{N}.$$
$$\langle a_0, \sigma \rangle \to n_0\ \&\ P(a_0, \sigma, n_0)\ \&\ \langle a_1, \sigma \rangle \to n_1\ \&\ P(a_1, \sigma, n_1)$$
$$\Rightarrow P(a_0 - a_1, \sigma, n_0 - n_1)$$
$$\&$$
$$\forall a_0, a_1 \in \mathbf{Aexp}, \sigma \in \Sigma, n_0, n_1 \in \mathbf{N}.$$
$$\langle a_0, \sigma \rangle \to n_0\ \&\ P(a_0, \sigma, n_0)\ \&\ \langle a_1, \sigma \rangle \to n_1\ \&\ P(a_1, \sigma, n_1)$$
$$\Rightarrow P(a_0 \times a_1, \sigma, n_0 \times n_1)].$$

Compare this specific principle with that for general rule induction. Notice how all possible rule instances are covered by considering one evaluation rule at a time.

4.3.2 Rule induction for boolean expressions

The rules for the evaluation of boolean expressions involve those for the evaluation of arithmetic expressions. Together the rules define a subset of

$$(\mathbf{Aexp} \times \Sigma \times \mathbf{N}) \cup (\mathbf{Bexp} \times \Sigma \times \mathbf{T}).$$

A principle useful for reasoning about the operational semantics of boolean expressions is got from the special principle of rule induction for properties $P(b, \sigma, t)$ on the subset $\mathbf{Bexp} \times \Sigma \times \mathbf{T}$.

$$\forall b \in \mathbf{Bexp}, \sigma \in \Sigma, t \in \mathbf{T}. \quad \langle b, \sigma \rangle \to t \Rightarrow P(b, \sigma, t)$$

iff

$$[\forall \sigma \in \Sigma.\ P(\mathbf{false}, \sigma, \mathbf{false}) \ \&\ \forall \sigma \in \Sigma.\ P(\mathbf{true}, \sigma, \mathbf{true})$$

&

$$\forall a_0, a_1 \in \mathbf{Aexp}, \sigma \in \Sigma, m, n \in \mathbf{N}.$$

$$\langle a_0, \sigma \rangle \to m \ \&\ \langle a_1, \sigma \rangle \to n \ \&\ m = n \Rightarrow P(a_0 = a_1, \sigma, \mathbf{true})$$

&

$$\forall a_0, a_1 \in \mathbf{Aexp}, \sigma \in \Sigma, m, n \in \mathbf{N}.$$

$$\langle a_0, \sigma \rangle \to m \ \&\ \langle a_1, \sigma \rangle \to n \ \&\ m \neq n \Rightarrow P(a_0 = a_1, \sigma, \mathbf{false})$$

&

$$\forall a_0, a_1 \in \mathbf{Aexp}, \sigma \in \Sigma, m, n \in \mathbf{N}.$$

$$\langle a_0, \sigma \rangle \to m \ \&\ \langle a_1, \sigma \rangle \to n \ \&\ m \leq n \Rightarrow P(a_0 \leq a_1, \sigma, \mathbf{true})$$

&

$$\forall a_0, a_1 \in \mathbf{Aexp}, \sigma \in \Sigma, m, n \in \mathbf{N}.$$

$$\langle a_0, \sigma \rangle \to m \ \&\ \langle a_1, \sigma \rangle \to n \ \&\ m \not\leq n \Rightarrow P(a_0 \leq a_1, \sigma, \mathbf{false})$$

&

$$\forall b \in \mathbf{Bexp}, \sigma \in \Sigma, t \in \mathbf{T}.$$

$$\langle b, \sigma \rangle \to t \ \&\ P(b, \sigma, t) \Rightarrow P(\neg b, \sigma, \neg t)$$

&

$$\forall b_0, b_1 \in \mathbf{Bexp}, \sigma \in \Sigma, t_0, t_1 \in \mathbf{T}.$$

$$\langle b_0, \sigma \rangle \to t_0 \ \&\ P(b_0, \sigma, t_0) \ \&\ \langle b_1, \sigma \rangle \to t_1 \ \&\ P(b_1, \sigma, t_1) \Rightarrow P(b_0 \wedge b_1, \sigma, t_0 \wedge t_1)$$

&

$$\forall b_0, b_1 \in \mathbf{Bexp}, \sigma \in \Sigma, t_0, t_1 \in \mathbf{T}.$$

$$\langle b_0, \sigma \rangle \to t_0 \ \&\ P(b_0, \sigma, t_0) \ \&\ \langle b_1, \sigma \rangle \to t_1 \ \&\ P(b_1, \sigma, t_1) \Rightarrow P(b_0 \vee b_1, \sigma, t_0 \vee t_1)].$$

4.3.3 Rule induction for commands

The principle of rule induction we use for reasoning about the operational semantics of commands is an instance of the special principle of rule induction. The rules for the execution of commands involve the evaluation of arithmetic and boolean expressions. The rules for the operational semantics of the different syntactic sets taken together

define a subset of

$$(\mathbf{Aexp} \times \Sigma \times \mathbf{N}) \cup (\mathbf{Bexp} \times \Sigma \times \mathbf{T}) \cup (\mathbf{Com} \times \Sigma \times \Sigma).$$

We use the special principle for properties $P(c, \sigma, \sigma')$ on the subset $\mathbf{Com} \times \Sigma \times \Sigma$.
(Try to write it down and compare your result with the following.)

$$\forall c \in \mathbf{Com}, \sigma, \sigma' \in \Sigma. \quad \langle c, \sigma \rangle \to \sigma' \Rightarrow P(c, \sigma, \sigma')$$

iff

$$[\forall \sigma \in \Sigma.\ P(\mathbf{skip}, \sigma, \sigma)$$

&

$$\forall X \in \mathbf{Loc}, a \in \mathbf{Aexp}, \sigma \in \Sigma, m \in \mathbf{N}.\ \langle a, \sigma \rangle \to m \Rightarrow P(X := a, \sigma, \sigma[m/X])$$

&

$$\forall c_0, c_1 \in \mathbf{Com}, \sigma, \sigma', \sigma'' \in \Sigma.$$
$$\langle c_0, \sigma \rangle \to \sigma'' \ \&\ P(c_0, \sigma, \sigma'') \ \&\ \langle c_1, \sigma'' \rangle \to \sigma' \ \&\ P(c_1, \sigma'', \sigma') \Rightarrow P(c_0; c_1, \sigma, \sigma')$$

&

$$\forall c_0, c_1 \in \mathbf{Com}, b \in \mathbf{Bexp}, \sigma, \sigma' \in \Sigma.$$
$$\langle b, \sigma \rangle \to \mathbf{true}\ \&\ \langle c_0, \sigma \rangle \to \sigma' \ \&\ P(c_0, \sigma, \sigma') \Rightarrow P(\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma, \sigma')$$

&

$$\forall c_0, c_1 \in \mathbf{Com}, b \in \mathbf{Bexp}, \sigma, \sigma' \in \Sigma.$$
$$\langle b, \sigma \rangle \to \mathbf{false}\ \&\ \langle c_1, \sigma \rangle \to \sigma' \ \&\ P(c_1, \sigma, \sigma') \Rightarrow P(\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma, \sigma')$$

&

$$\forall c \in \mathbf{Com}, b \in \mathbf{Bexp}, \sigma \in \Sigma.$$
$$\langle b, \sigma \rangle \to \mathbf{false} \Rightarrow P(\mathbf{while}\ b\ \mathbf{do}\ c, \sigma, \sigma)$$

&

$$\forall c \in \mathbf{Com}, b \in \mathbf{Bexp}, \sigma, \sigma', \sigma'' \in \Sigma.$$
$$\langle b, \sigma \rangle \to \mathbf{true}\ \&\ \langle c, \sigma \rangle \to \sigma''\ \&\ P(c, \sigma, \sigma'')\ \&$$
$$\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma'' \rangle \to \sigma'\ \&\ P(\mathbf{while}\ b\ \mathbf{do}\ c, \sigma'', \sigma')$$
$$\Rightarrow P(\mathbf{while}\ b\ \mathbf{do}\ c, \sigma, \sigma')].$$

As an example, we apply rule induction to show the intuitively obvious fact that if a location Y does not occur in the left hand side of an assignment in a command c then execution of c cannot affect its value. Recall the definition of the locations $\mathrm{loc}_L(c)$ of a command c given in Section 3.5.

Proposition 4.7 *Let* $Y \in \mathbf{Loc}$. *For all commands* c *and states* σ, σ',

$$Y \notin loc_L(c) \;\&\; \langle c, \sigma\rangle \to \sigma' \Rightarrow \sigma(Y) = \sigma'(Y).$$

Proof: Let P be the property given by:

$$P(c,\sigma,\sigma') \iff (Y \notin \mathrm{loc}_L(c) \Rightarrow \sigma(Y) = \sigma'(Y)).$$

We use rule induction on commands to show that

$$\forall c \in \mathbf{Com}, \sigma, \sigma' \in \Sigma. \quad \langle c, \sigma\rangle \to \sigma' \Rightarrow P(c,\sigma,\sigma').$$

Clearly $P(\mathbf{skip}, \sigma, \sigma)$ for any $\sigma \in \Sigma$.

Let $X \in \mathbf{Loc}, a \in \mathbf{Aexp}, \sigma \in \Sigma, m \in \mathbf{N}$. Assume $\langle a, \sigma\rangle \to m$. If $Y \notin \mathrm{loc}_L(X := a)$ then $Y \not\equiv X$, so $\sigma(Y) = \sigma[m/X](Y)$. Hence $P(X := a, \sigma, \sigma[m/X])$.

Let $c_0, c_1 \in \mathbf{Com}, \sigma, \sigma' \in \Sigma$. Assume

$$\langle c_0, \sigma\rangle \to \sigma'' \;\&\; P(c_0,\sigma,\sigma'') \;\&\; \langle c_1, \sigma''\rangle \to \sigma' \;\&\; P(c_1,\sigma'',\sigma'),$$

i.e., that

$$\langle c_0, \sigma\rangle \to \sigma'' \;\&\; (Y \notin \mathrm{loc}_L(c_0) \Rightarrow \sigma(Y) = \sigma''(Y)) \;\&$$
$$\langle c_1, \sigma''\rangle \to \sigma' \;\&\; (Y \notin \mathrm{loc}_L(c_1) \Rightarrow \sigma''(Y) = \sigma'(Y)).$$

Suppose $Y \notin \mathrm{loc}_L(c_0; c_1)$. Then, as $\mathrm{loc}_L(c_0; c_1) = \mathrm{loc}_L(c_0) \cup \mathrm{loc}_L(c_1)$, we obtain $Y \notin \mathrm{loc}_L(c_0)$ and $Y \notin \mathrm{loc}_L(c_1)$. Thus, from the assumption, $\sigma(Y) = \sigma''(Y) = \sigma'(Y)$. Hence $P(c_0; c_1, \sigma, \sigma')$.

We shall only consider one other case of rule instances.

Let $c \in \mathbf{Com}, b \in \mathbf{Bexp}, \sigma, \sigma', \sigma'' \in \Sigma$. Let $w \equiv \mathbf{while}\ b\ \mathbf{do}\ c$. Assume

$$\langle b, \sigma\rangle \to \mathbf{true} \;\&\; \langle c, \sigma\rangle \to \sigma'' \;\&\; P(c,\sigma,\sigma'') \;\&$$
$$\langle w, \sigma''\rangle \to \sigma' \;\&\; P(w,\sigma'',\sigma')$$

i.e.,

$$\langle b, \sigma\rangle \to \mathbf{true} \;\&\; \langle c, \sigma\rangle \to \sigma'' \;\&\; (Y \notin \mathrm{loc}_L(c) \Rightarrow \sigma(Y) = \sigma''(Y)) \;\&$$
$$\langle w, \sigma''\rangle \to \sigma' \;\&\; (Y \notin \mathrm{loc}_L(w) \Rightarrow \sigma''(Y) = \sigma'(Y)).$$

Suppose $Y \notin \mathrm{loc}_L(w)$. By the assumption $\sigma''(Y) = \sigma'(Y)$. Also, as $\mathrm{loc}_L(w) = \mathrm{loc}_L(c)$, we see $Y \notin \mathrm{loc}_L(c)$, so by the assumption $\sigma(Y) = \sigma''(Y)$. Thus $\sigma(Y) = \sigma'(Y)$. Hence $P(w, \sigma, \sigma')$.

The other cases are very similar and left as an exercise. □

We shall see many more proofs by rule inductiсn in subsequent chapters. In general they will be smooth and direct arguments. Here are some more difficult exercises on using rule induction. As the first two exercises indicate applications of rule induction can sometimes be tricky.

Exercise 4.8 Let $w \equiv$ **while true do skip**. Prove by special rule induction that

$$\forall \sigma, \sigma'.\ \langle w, \sigma \rangle \not\to \sigma'.$$

(Hint: Apply the special principle of rule induction restricting to the set

$$\{(w, \sigma, \sigma') \mid \sigma, \sigma' \in \Sigma\}$$

and take the property $P(w, \sigma, \sigma')$ to be constantly false.
It is interesting to compare the proof for this exercise with that of Proposition 3.12 in Section 3.4—proofs by rule induction can sometimes be less intuitive than proofs in which the form of derivations is considered.) □

Although rule induction can be used in place of induction on derivations it is no panacea; exclusive use of rule induction can sometimes make proofs longer and more confusing, as will probably become clear on trying the following exercise:

Exercise 4.9 Take a simplified syntax of arithmetic expressions:

$$a ::= n \mid X \mid a_0 + a_1.$$

The evaluation rules of the simplified expressions are as before:

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

$$\langle X, \sigma \rangle \to \sigma(X)$$

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n}$$

where n is the number which is the sum of n_0 and n_1.

By considering the unique form of derivations it is easy to see that $\langle n, \sigma \rangle \to m$ implies $m \equiv n$. Can you see how this follows by special rule induction? Use rule induction on the operational semantics (and not induction on derivations) to show that the evaluation

of expressions is deterministic.
(Hint: For the latter, take

$$P(a,\sigma,m) \iff_{def} \forall m' \in \mathbf{N}.\ \langle a,\sigma\rangle \to m' \Rightarrow m = m'$$

as induction hypothesis, and be prepared for a further use of (special) rule induction.)
An alternative proof, of Proposition 3.3 in Section 3.2, uses structural induction and considers the forms that derivations could take. How does the proof compare with that of Proposition 3.3?
□

The next, fairly long, exercise proves the equivalence of two operational semantics.

Exercise 4.10 (Long) One operational semantics is that of Chapter 2, based on the relation $\langle c,\sigma\rangle \to \sigma'$. The other is the one-step execution relation $\langle c,\sigma\rangle \to_1 \langle c',\sigma'\rangle$ mentioned previously in Section 2.6, but where, for simplicity, evaluation of expressions is treated in exactly the same way as in Chapter 2. For instance, for the sequencing of two commands there are the rules:

$$\frac{\langle c_0,\sigma\rangle \to_1 \langle c_0',\sigma'\rangle}{\langle c_0;c_1,\sigma\rangle \to_1 \langle c_0';c_1,\sigma'\rangle} \qquad \frac{\langle c_0,\sigma\rangle \to_1 \sigma'}{\langle c_0;c_1,\sigma\rangle \to_1 \langle c_1,\sigma'\rangle}$$

Start by proving the lemma

$$\langle c_0;c_1,\sigma\rangle \to_1^* \sigma' \text{ iff } \exists\sigma''.\ \langle c_0,\sigma\rangle \to_1^* \sigma'' \ \&\ \langle c_1,\sigma''\rangle \to_1^* \sigma',$$

for all commands c_0, c_1 and all states σ, σ'. Prove this in two stages. Firstly prove

$$\forall\sigma,\sigma'.\ [\langle c_0;c_1,\sigma\rangle \to_1^n \sigma' \Rightarrow \exists\sigma''.\ \langle c_0,\sigma\rangle \to_1^* \sigma'' \ \&\ \langle c_1,\sigma''\rangle \to_1^* \sigma']$$

by mathematical induction on n, the length of computation. Secondly prove

$$\forall\sigma,\sigma',\sigma''.\ [\langle c_0,\sigma\rangle \to_1^n \sigma'' \ \&\ \langle c_1,\sigma''\rangle \to_1^* \sigma' \Rightarrow \langle c_0;c_1,\sigma\rangle \to_1^* \sigma']$$

by mathematical induction on n, this time the length of the execution of c_0 from state σ. Conclude that the lemma holds. Now proceed to the proof of the theorem:

$$\forall\sigma,\sigma'.\ [\langle c,\sigma\rangle \to_1^* \sigma' \text{ iff } \langle c,\sigma\rangle \to \sigma'].$$

The "only if" direction of the proof can be done by structural induction on c, with an induction on the length of the computation in the case where c is a while-loop. The "if" direction of the proof can be done by rule induction (or by induction on derivations). □

4.4 Operators and their least fixed points

There is another way to view a set defined by rules. A set of rule instances R determines an operator $\widehat{R}$ on sets, which given a set B results in a set

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

Use of the operator $\widehat{R}$ gives another way of saying a set is R-closed.

Proposition 4.11 *A set B is closed under R iff $\widehat{R}(B) \subseteq B$.*

Proof: The fact follows directly from the definitions. □

The operator $\widehat{R}$ provides a way of building up the set I_R. The operator $\widehat{R}$ is *monotonic* in the sense that

$$A \subseteq B \Rightarrow \widehat{R}(A) \subseteq \widehat{R}(B).$$

If we repeatedly apply $\widehat{R}$ to the empty set $\emptyset$ we obtain the sequence of sets:

$$\begin{aligned} A_0 &= \widehat{R}^0(\emptyset) = \emptyset, \\ A_1 &= \widehat{R}^1(\emptyset) = \widehat{R}(\emptyset), \\ A_2 &= \widehat{R}(\widehat{R}(\emptyset)) = \widehat{R}^2(\emptyset), \\ &\vdots \\ A_n &= \widehat{R}^n(\emptyset), \\ &\vdots \end{aligned}$$

The set A_1 consists of all the conclusions of instances of axioms, and in general the set A_{n+1} is all things which immediately follow by rule instances with premises in A_n. Clearly $\emptyset \subseteq \widehat{R}(\emptyset)$, *i.e.* $A_0 \subseteq A_1$. By the monotonicity of $\widehat{R}$ we obtain $\widehat{R}(A_0) \subseteq \widehat{R}(A_1)$, *i.e.* $A_1 \subseteq A_2$. Similarly we obtain $A_2 \subseteq A_3$ *etc.*. Thus the sequence forms a chain

$$A_0 \subseteq A_1 \subseteq \cdots \subseteq A_n \subseteq \cdots.$$

Taking $A = \bigcup_{n \in \omega} A_n$, we have:

Proposition 4.12

(i) A is R-closed.

(ii) $\widehat{R}(A) = A$.

(iii) A is the least R-closed set.

Proof:
(i) Suppose $(X/y) \in R$ with $X \subseteq A$. Recall $A = \bigcup_n A_n$ is the union of an increasing chain of sets. As X is a finite set there is some n such that $X \subseteq A_n$. (The set X is either empty, whence $X \subseteq A_0$, or of the form $\{x_1, \ldots, x_k\}$. In the latter case, we have $x_1 \in A_{n_1}, \cdots, x_k \in A_{n_k}$ for some $n_1, \ldots, n_k$. Taking n bigger than all of $n_1, \ldots, n_k$ we must have $X \subseteq A_n$ as the sequence $A_0, A_1, \ldots, A_n, \ldots$ is increasing.) As $X \subseteq A_n$ we obtain $y \in \widehat{R}(A_n) = A_{n+1}$. Hence $y \in \bigcup_n A_n = A$. Thus A is closed under R.
(ii) By Proposition 4.11 the set A is R-closed, so we already know that $\widehat{R}(A) \subseteq A$. We require the converse inclusion. Suppose $y \in A$. Then $y \in A_n$ for some $n > 0$. Thus $y \in \widehat{R}(A_{n-1})$. This means there is some $(X/y) \in R$ with $X \subseteq A_{n-1}$. But $A_{n-1} \subseteq A$ so $X \subseteq A$ with $(X/y) \in R$, giving $y \in \widehat{R}(A)$. We have established the required converse inclusion, $A \subseteq \widehat{R}(A)$. Hence $\widehat{R}(A) = A$.
(iii) We need to show that if B is another R-closed set then $A \subseteq B$. Suppose B is closed under R. Then $\widehat{R}(B) \subseteq B$. We show by mathematical induction that for all natural numbers $n \in \omega$

$$A_n \subseteq B.$$

The basis of the induction $A_0 \subseteq B$ is obviously true as $A_0 = \emptyset$. To show the induction step, assume $A_n \subseteq B$. Then

$$A_{n+1} = \widehat{R}(A_n) \subseteq \widehat{R}(B) \subseteq B,$$

using the facts that $\widehat{R}$ is monotonic and that B is R-closed. □

Notice the essential part played in the proof of (i) by the fact that rule instances are finitary, *i.e.* in a rule instance (X/y), the set of premises X is finite.

It follows from (i) and (iii) that $A = I_R$, the set of elements for which there are R-derivations. Now (ii) says precisely that I_R is a fixed point of $\widehat{R}$. Moreover, (iii) implies that I_R is the *least fixed point* of $\widehat{R}$, *i.e.*

$$\widehat{R}(B) = B \Rightarrow I_R \subseteq B,$$

because if any other set B is a fixed point it is closed under R, so $I_R \subseteq B$ by Proposition 4.1. The set I_R, defined by the rule instances R, is the least fixed point, $fix(\widehat{R})$, obtained by the construction

$$fix(\widehat{R}) =_{def} \bigcup_{n \in \omega} \widehat{R}^n(\emptyset).$$

Least fixed points will play a central role in the next chapter.

Exercise 4.13 Given a set of rules R define a different operator $\overline{R}$ by

$$\overline{R}(A) = A \cup \{y \mid \exists X \subseteq A.\ (X/y) \in R\}.$$

Clearly $\overline{R}$ is monotonic and in addition satisfies the property

$$A \subseteq \overline{R}(A).$$

An operator satisfying such a property is called *increasing*. Exhibit a monotonic operator which is not increasing. Show that given any set A there is a least fixed point of $\overline{R}$ which includes A, and that this property can fail for monotonic operations. □

Exercise 4.14 Let R be a set of rule instances. Show that $\widehat{R}$ is *continuous* in the sense that

$$\bigcup_{n\in\omega} \widehat{R}(B_n) = \widehat{R}(\bigcup_{n\in\omega} B_n)$$

for any increasing chain of sets $B_0 \subseteq \cdots \subseteq B_n \subseteq \cdots$.
(The solution to this exercise is contained in the next chapter.) □

4.5 Further reading

This chapter has provided an elementary introduction to the mathematical theory of *inductive definitions*. A detailed, though much harder, account can be found in Peter Aczel's handbook chapter [4]—our treatment, with just finitary rules, avoids the use of ordinals. The term "rule induction" originates with the author's Cambridge lecture notes of 1984, and seems be catching on (the principle is well-known and, for instance, is called simply R-induction, for rules R, in [4]). This chapter has refrained from any recommendations about which style of argument to use in reasoning about operational semantics; whether to use rule induction or the often clumsier, but conceptually more straightforward, induction on derivations. In many cases it is a matter of taste.

5 The denotational semantics of IMP

This chapter provides a denotational semantics for **IMP**, and a proof of its equivalence with the previously given operational semantics. The chapter concludes with an introduction to the foundations of denotational semantics (complete partial orders, continuous functions and least fixed points) and the Knaster-Tarski Theorem.

5.1 Motivation

We have described the behaviour of programs in **IMP** in an operational manner by inductively defining transition relations to express evaluation and execution. There was some arbitrariness in the choice of rules, for example, in the size of transition steps we chose. Also note that in the description of the behaviour the syntax was mixed-up in the description. This style of semantics, in which the transitions are built out of the syntax, makes it hard to compare two programs written in different programming languages. Still, the style of semantics was fairly close to an implemention of the language, the description can be turned into an interpreter for **IMP** written for example in Prolog, and it led to firm definitions of equivalence between arithmetic expressions, boolean expressions and commands. For example we defined

$$c_0 \sim c_1 \text{ iff } (\forall \sigma, \sigma'.\ \langle c_0, \sigma\rangle \to \sigma' \iff \langle c_1, \sigma\rangle \to \sigma').$$

Perhaps it has already occurred to the reader that there is a more direct way to capture the semantics of **IMP** if we are only interested in commands to within the equivalence $\sim$. Notice $c_0 \sim c_1$ iff

$$\{(\sigma, \sigma') \mid \langle c_0, \sigma\rangle \to \sigma'\} = \{(\sigma, \sigma') \mid \langle c_1, \sigma\rangle \to \sigma'\}.$$

In other words, $c_0 \sim c_1$ iff c_0 and c_1 determine the same partial function on states. This suggests we should define the meaning, or semantics, of **IMP** at a more abstract level in which we take the *denotation* of a command to be a partial function on states. The style we adopt in giving this new description of the semantics of **IMP** is that from *denotational semantics.* Denotational semantics is much more widely applicable than to simple programming languages like **IMP** —it can handle virtually all programming languages. though the standard framework appears inadequate for parallelism and "fairness" (see Chapter 14 on parallelism). The approach was pioneered by Christopher Strachey, and Dana Scott who supplied the mathematical foundations. Our denotational semantics of **IMP** is really just an introductory example. We shall see more on the applications and foundations of denotational semantics in later chapters.

An arithmetic expression $a \in \mathbf{Aexp}$ will denote a function $\mathcal{A}[\![a]\!] : \Sigma \to \mathbf{N}$.

A boolean expression $b \in \mathbf{Bexp}$ will denote a function $\mathcal{B}[\![b]\!] : \Sigma \to \mathbf{T}$, from the set of states to the set of truth values.

A command c will denote a partial function $\mathcal{C}[\![c]\!] : \Sigma \rightharpoonup \Sigma$.

The brackets $[\![\ ]\!]$ are traditional in denotational semantics. You see $\mathcal{A}$ is really a function from arithmetic expressions of the type $\mathbf{Aexp} \to (\Sigma \to \mathbf{N})$, and our first thought in ordinary mathematics, when we see an expression, is to evaluate it. The square brackets $[\![a]\!]$ put the arithmetic expression a in quotes so we don't evaluate a. We could have written *e.g.* $\mathcal{A}(\text{“}3+5\text{”})\sigma = 8$ instead of $\mathcal{A}[\![3+5]\!]\sigma = 8$. The quotes tell that it is the piece of syntax “3+5” which is being mapped. The full truth is a little more subtle as we shall sometimes write denotations like $\mathcal{A}[\![a_0 + a_1]\!]$, where a_0 and a_1 are metavariables which stand for arithmetic expressions. It is the syntactic object got by placing the sign “+” between the syntactic objects a_0 and a_1 that is put in quotes. So the brackets $[\![\ ]\!]$ do not represent true and complete quotation. We shall use the brackets $[\![\ ]\!]$ round an argument of a semantic function to show that the argument is a piece of syntax.

5.2 Denotational semantics

We define the semantic functions

$$\mathcal{A} : \mathbf{Aexp} \to (\Sigma \to \mathbf{N})$$
$$\mathcal{B} : \mathbf{Bexp} \to (\Sigma \to \mathbf{T})$$
$$\mathcal{C} : \mathbf{Com} \to (\Sigma \rightharpoonup \Sigma)$$

by structural induction. For example, for commands, for each command c we define the partial function $\mathcal{C}[\![c]\!]$ assuming the previous definition of c' for subcommands c' of c. The command c is said to *denote* $\mathcal{C}[\![c]\!]$, and $\mathcal{C}[\![c]\!]$ is said to be a *denotation* of c.

Denotations of Aexp:

Firstly, we define the denotation of an arithmetic expression, by structural induction, as a relation between states and numbers:

$$\mathcal{A}[\![n]\!] = \{(\sigma, n) \mid \sigma \in \Sigma\}$$
$$\mathcal{A}[\![X]\!] = \{(\sigma, \sigma(X)) \mid \sigma \in \Sigma\}$$
$$\mathcal{A}[\![a_0 + a_1]\!] = \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \ \&\ (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$$
$$\mathcal{A}[\![a_0 - a_1]\!] = \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \ \&\ (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$$
$$\mathcal{A}[\![a_0 \times a_1]\!] = \{(\sigma, n_0 \times n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \ \&\ (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}.$$

An obvious structural induction on arithmetic expressions a shows that each denotation $\mathcal{A}[\![a]\!]$ is in fact a function. Notice that the signs “+”, “−”, “×” on the left-hand sides represent syntactic signs in **IMP** whereas the signs on the right represent operations on

numbers, so *e.g.*, for any state σ,

$$\mathcal{A}[\![3+5]\!]\sigma = \mathcal{A}[\![3]\!]\sigma + \mathcal{A}[\![5]\!]\sigma = 3+5 = 8,$$

as is to be expected. Note that using λ-notation we can present the definition of the semantics in the following equivalent way:

$$\begin{aligned}
\mathcal{A}[\![n]\!] &= \lambda\sigma \in \Sigma.n \\
\mathcal{A}[\![X]\!] &= \lambda\sigma \in \Sigma.\sigma(X) \\
\mathcal{A}[\![a_0 + a_1]\!] &= \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_0]\!]\sigma + \mathcal{A}[\![a_1]\!]\sigma) \\
\mathcal{A}[\![a_0 - a_1]\!] &= \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_0]\!]\sigma - \mathcal{A}[\![a_1]\!]\sigma) \\
\mathcal{A}[\![a_0 \times a_1]\!] &= \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_0]\!]\sigma \times \mathcal{A}[\![a_1]\!]\sigma).
\end{aligned}$$

Denotations of Bexp:

The semantic function for booleans is given in terms of logical operations conjunction $\wedge_T$, disjunction $\vee_T$ and negation $\neg_T$, on the set of truth values $\mathbf{T}$. The denotation of a boolean expression is defined by structural induction to be a relation between states and truth values.

$$\mathcal{B}[\![\mathbf{true}]\!] = \{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma\}$$

$$\mathcal{B}[\![\mathbf{false}]\!] = \{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma\}$$

$$\begin{aligned}
\mathcal{B}[\![a_0 = a_1]\!] = \{&(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \;\&\; \mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma\} \cup \\
&\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \;\&\; \mathcal{A}[\![a_0]\!]\sigma \neq \mathcal{A}[\![a_1]\!]\sigma\},
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}[\![a_0 \leq a_1]\!] = \{&(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \;\&\; \mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma\} \cup \\
&\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \;\&\; \mathcal{A}[\![a_0]\!]\sigma \not\leq \mathcal{A}[\![a_1]\!]\sigma\},
\end{aligned}$$

$$\mathcal{B}[\![\neg b]\!] = \{(\sigma, \neg_T t) \mid \sigma \in \Sigma \;\&\; (\sigma, t) \in \mathcal{B}[\![b]\!]\},$$

$$\mathcal{B}[\![b_0 \wedge b_1]\!] = \{(\sigma, t_0 \wedge_T t_1) \mid \sigma \in \Sigma \;\&\; (\sigma, t_0) \in \mathcal{B}[\![b_0]\!] \;\&\; (\sigma, t_1) \in \mathcal{B}[\![b_1]\!]\},$$

$$\mathcal{B}[\![b_0 \vee b_1]\!] = \{(\sigma, t_0 \vee_T t_1) \mid \sigma \in \Sigma \;\&\; (\sigma, t_0) \in \mathcal{B}[\![b_0]\!] \;\&\; (\sigma, t_1) \in \mathcal{B}[\![b_1]\!]\}.$$

A simple structural induction shows that each denotation is a function. For example,

$$\mathcal{B}[\![a_0 \leq a_1]\!]\sigma = \begin{cases} \textbf{true} & \text{if } \mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma, \\ \textbf{false} & \text{if } \mathcal{A}[\![a_0]\!]\sigma \not\leq \mathcal{A}[\![a_1]\!]\sigma \end{cases}$$

for all $\sigma \in \Sigma$.

Denotations of Com:

The definition of $\mathcal{C}[\![c]\!]$ for commands c is more complicated. We will first give denotations as relations between states; afterwards a straightforward structural induction will show that they are, in fact, partial functions. It is fairly obvious that we should take

$$\mathcal{C}[\![\textbf{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

$$\mathcal{C}[\![X := a]\!] = \{(\sigma, \sigma[n/X]) \mid \sigma \in \Sigma \ \&\ n = \mathcal{A}[\![a]\!]\sigma\}$$

$\mathcal{C}[\![c_0; c_1]\!] = \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!]$, a composition of relations,
the definition of which explains the order-reversal in c_0 and c_1,

$$\mathcal{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!] =$$
$$\{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{true} \ \&\ (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\} \cup \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{false} \ \&\ (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\}.$$

But there are difficulties when we consider the denotation of a while-loop. Write

$$w \equiv \textbf{while } b \textbf{ do } c.$$

We have noted the equivalence

$$w \sim \textbf{if } b \textbf{ then } c; w \textbf{ else skip}$$

so the partial function $\mathcal{C}[\![w]\!]$ should equal the partial function $\mathcal{C}[\![\textbf{if } b \textbf{ then } c; w \textbf{ else skip}]\!]$. Thus we should have :

$$\begin{aligned} \mathcal{C}[\![w]\!] =& \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{true} \ \&\ (\sigma, \sigma') \in \mathcal{C}[\![c; w]\!]\} \cup \\ & \{(\sigma, \sigma) \mid \mathcal{B}[\![b]\!]\sigma = \textbf{false}\} \\ =& \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{true} \ \&\ (\sigma, \sigma') \in \mathcal{C}[\![w]\!] \circ \mathcal{C}[\![c]\!]\} \cup \\ & \{(\sigma, \sigma) \mid \mathcal{B}[\![b]\!]\sigma = \textbf{false}\}. \end{aligned}$$

Writing φ for $\mathcal{C}[\![w]\!]$, β for $\mathcal{B}[\![b]\!]$ and γ for $\mathcal{C}[\![c]\!]$ we require a partial function φ such that

$$\begin{aligned} \varphi =& \{(\sigma, \sigma') \mid \beta(\sigma) = \textbf{true} \ \&\ (\sigma, \sigma') \in \varphi \circ \gamma\} \cup \\ & \{(\sigma, \sigma) \mid \beta(\sigma) = \textbf{false}\}. \end{aligned}$$

But this involves φ on both sides of the equation. How can we solve it to find φ? We clearly require some technique for solving a recursive equation of this form (it is called "recursive" because the value we wish to know on the left recurs on the right). Looked at in another way we can regard Γ, where

$$\begin{aligned}\Gamma(\varphi) =&\{(\sigma,\sigma') \mid \beta(\sigma) = \mathbf{true} \;\&\; (\sigma,\sigma') \in \varphi \circ \gamma\} \cup \\ &\{(\sigma,\sigma) \mid \beta(\sigma) = \mathbf{false}\} \\ =&\{(\sigma,\sigma') \mid \exists\sigma''.\ \beta(\sigma) = \mathbf{true} \;\&\; (\sigma,\sigma'') \in \gamma \;\&\; (\sigma'',\sigma') \in \varphi\} \cup \\ &\{(\sigma,\sigma) \mid \beta(\sigma) = \mathbf{false}\},\end{aligned}$$

as a function which given φ returns $\Gamma(\varphi)$. We want a fixed point φ of Γ in the sense that

$$\varphi = \Gamma(\varphi).$$

The last chapter provides the clue to finding such a solution in Section 4.4. It is not hard to check that the function Γ is equal to $\widehat{R}$, where $\widehat{R}$ is the operator on sets determined by the rule instances

$$\begin{aligned}R =&\{(\{(\sigma'',\sigma')\}/(\sigma,\sigma')) \mid \beta(\sigma) = \mathbf{true} \;\&\; (\sigma,\sigma'') \in \gamma\} \cup \\ &\{(\emptyset/(\sigma,\sigma)) \mid \beta(\sigma) = \mathbf{false}\}.\end{aligned}$$

As Section 4.4 shows $\widehat{R}$ has a least fixed point

$$\varphi = \mathit{fix}(\widehat{R})$$

where φ is a set—in this case a set of pairs—with the property that

$$\widehat{R}(\theta) = \theta \Rightarrow \varphi \subseteq \theta.$$

We shall take this least fixed point as the denotation of the while program w. Certainly its denotation should be a fixed point. The full justification for taking it to be the *least* fixed point will be given in the next section where we establish that this choice for the semantics agrees with the operational semantics.

Now we can go ahead and define the denotational semantics of commands in the

following way, by structural induction:

$$\mathcal{C}[\![\mathbf{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

$$\mathcal{C}[\![X := a]\!] = \{(\sigma, \sigma[n/X]) \mid \sigma \in \Sigma \;\&\; n = \mathcal{A}[\![a]\!]\sigma\}$$

$$\mathcal{C}[\![c_0; c_1]\!] = \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!]$$

$$\mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!] =$$
$$\{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{true} \;\&\; (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\} \cup$$
$$\{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{false} \;\&\; (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\}$$

$$\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = \mathit{fix}(\Gamma)$$

where

$$\Gamma(\varphi) = \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{true} \;\&\; (\sigma, \sigma') \in \varphi \circ \mathcal{C}[\![c]\!]\} \cup \{(\sigma, \sigma) \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{false}\}.$$

In this way we define a denotation of each command as a relation between states. Notice how the semantic definition is *compositional* in the sense that the denotation of a command is constructed from the denotations of its immediate subcommands, reflected in the fact that the definition is by structural induction. This property is a hallmark of denotational semantics. Notice it is not true of the operational semantics of **IMP** because of the rule for **while**-loops in which the **while**-loop reappears in the premise of the rule.

We have based the definition of the semantic function on while programs by the operational equivalence between while programs and one "unfolding" of them into a conditional. Not surprisingly it is straightforward to check this equivalence holds according to the denotational semantics.

Proposition 5.1 *Write*

$$w \equiv \mathbf{while}\ b\ \mathbf{do}\ c$$

for a command c and boolean expression b. Then

$$\mathcal{C}[\![w]\!] = \mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else}\ \mathbf{skip}]\!].$$

Proof: The denotation of w is a fixed point of Γ, defined above. Hence

$$\begin{aligned}
\mathcal{C}[\![w]\!] =& \Gamma(\mathcal{C}[\![w]\!]) \\
=& \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{true} \ \&\ (\sigma,\sigma') \in \mathcal{C}[\![w]\!] \circ \mathcal{C}[\![c]\!]\} \cup \\
& \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{false}\} \\
=& \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{true} \ \&\ (\sigma,\sigma') \in \mathcal{C}[\![c;w]\!]\} \cup \\
& \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{false} \ \&\ (\sigma,\sigma') \in \mathcal{C}[\![\mathbf{skip}]\!]\} \\
=& \mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c;w\ \mathbf{else}\ \mathbf{skip}]\!].\square
\end{aligned}$$

Exercise 5.2 Show by structural induction on commands that the denotation $\mathcal{C}[\![c]\!]$ is a partial function for all commands c.
(The case for while-loops involves proofs by mathematical induction showing that $\Gamma^n(\emptyset)$ is a partial function between states for all natural numbers n, and that these form an increasing chain, followed by the observation that the union of such a chain of partial functions is itself a partial function.) □

In Section 5.4 we shall introduce a general theory of fixed points, which makes sense when the objects defined recursively are not sets ordered by inclusion.

5.3 Equivalence of the semantics

Although inspired by our understanding of the operational behaviour of **IMP** the denotational semantics has not yet been demonstrated to agree with the operational semantics. We first check the operational and denotational semantics agree on the evaluation of expressions:

Lemma 5.3 *For all* $a \in$ **Aexp**,

$$\mathcal{A}[\![a]\!] = \{(\sigma,n) \mid \langle a,\sigma\rangle \to n\}.$$

Proof: We prove the lemma by structural induction. As induction hypothesis we take

$$P(a) \iff_{def} \mathcal{A}[\![a]\!] = \{(\sigma,n) \mid \langle a,\sigma\rangle \to n\}.$$

Following the scheme of structural induction the proof splits into cases according to the structure of an arithmetic expression a.
$a \equiv m$: From the definition of the semantic function, in the case where a is a number m, we have

$$(\sigma,n) \in \mathcal{A}[\![m]\!] \iff \sigma \in \Sigma \ \&\ n \equiv m.$$

Clearly, if $(\sigma, n) \in \mathcal{A}[\![m]\!]$ then $n \equiv m$ and $\langle m, \sigma \rangle \to n$. Conversely, if $\langle m, \sigma \rangle \to n$ then the only possible derivation is one in which $n \equiv m$ and hence $(\sigma, n) \in \mathcal{A}[\![m]\!]$.
$a \equiv X$: Similarly, if a is a location X,

$$(\sigma, n) \in \mathcal{A}[\![X]\!] \iff (\sigma \in \Sigma \;\&\; n \equiv \sigma(X))$$
$$\iff \langle X, \sigma \rangle \to n.$$

$a \equiv a_0 + a_1$: Assume $P(a_0)$ and $P(a_1)$ for two arithmetic expressions a_0, a_1. We have

$$(\sigma, n) \in \mathcal{A}[\![a_0 + a_1]\!] \iff \exists n_0, n_1.\; n = n_0 + n_1 \;\&\; (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \;\&\; (\sigma, n_1) \in \mathcal{A}[\![a_1]\!].$$

Supposing $(\sigma, n) \in \mathcal{A}[\![a_0 + a_1]\!]$, there are n_0, n_1 such that $n = n_0 + n_1$ and $(\sigma, n_0) \in \mathcal{A}[\![a_0]\!]$ and $(\sigma, n_1) \in \mathcal{A}[\![a_1]\!]$. From the assumptions $P(a_0)$ and $P(a_1)$, we obtain

$$\langle a_0, \sigma \rangle \to n_0 \quad \text{and} \quad \langle a_1, \sigma \rangle \to n_1.$$

Thus we can derive $\langle a_0 + a_1, \sigma \rangle \to n$. Conversely, any derivation of $\langle a_0 + a_1, \sigma \rangle \to n$ must have the form

$$\frac{\dot{\langle a_0, \sigma \rangle \to n_0} \quad \dot{\langle a_1, \sigma \rangle \to n_1}}{\langle a_0 + a_1, \sigma \rangle \to n}$$

for some n_0, n_1 such that $n = n_0 + n_1$. This time, from the assumptions $P(a_0)$ and $P(a_1)$, we obtain $(\sigma, n_0) \in \mathcal{A}[\![a_0]\!]$ and $(\sigma, n_1) \in \mathcal{A}[\![a_1]\!]$. Hence $(\sigma, n) \in \mathcal{A}[\![a]\!]$.

The proofs of the other cases, for arithmetic expressions of the form $a_0 - a_1$ and $a_0 \times a_1$, follow exactly the same pattern. By structural induction on arithmetic expressions we conclude that

$$\mathcal{A}[\![a]\!] = \{(\sigma, n) \mid \langle a, \sigma \rangle \to n\},$$

for all arithmetic expressions a. □

Lemma 5.4 *For $b \in$* **Bexp**,

$$\mathcal{B}[\![b]\!] = \{(\sigma, t) \mid \langle b, \sigma \rangle \to t\}.$$

Proof: The proof for boolean expressions is similar to that for arithmetic expressions. It proceeds by structural induction on boolean expressions with induction hypothesis

$$P(b) \iff_{def} \mathcal{B}[\![b]\!] = \{(\sigma, t) \mid \langle b, \sigma \rangle \to t\}$$

for boolean expression b.

We only do two cases of the induction. They are typical, and the remaining cases are left to the reader.

$b \equiv (a_0 = a_1)$: Let a_0, a_1 be arithmetic expressions. By definition, we have

$$\begin{aligned}\mathcal{B}[\![a_0 = a_1]\!] = &\{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \;\&\; \mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma\} \cup \\ &\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \;\&\; \mathcal{A}[\![a_0]\!]\sigma \neq \mathcal{A}[\![a_1]\!]\sigma\}.\end{aligned}$$

Thus

$$(\sigma, \mathbf{true}) \in \mathcal{B}[\![a_0 = a_1]\!] \iff \sigma \in \Sigma \;\&\; \mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma.$$

If $(\sigma, \mathbf{true}) \in \mathcal{B}[\![a_0 = a_1]\!]$ then $\mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma$, so, by the previous lemma,

$$\langle a_0, \sigma\rangle \to n \quad \text{and} \quad \langle a_1, \sigma\rangle \to n,$$

for some number n. Hence from the operational semantics for boolean expressions we obtain

$$\langle a_0 = a_1, \sigma\rangle \to \mathbf{true}.$$

Conversely, supposing $\langle a_0 = a_1, \sigma\rangle \to \mathbf{true}$, it must have a derivation of the form

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \langle a_0, \sigma\rangle \to n & \langle a_1, \sigma\rangle \to n\end{array}}{\langle a_0 = a_1, \sigma\rangle \to \mathbf{true}}.$$

But then, by the previous lemma, $\mathcal{A}[\![a_0]\!]\sigma = n = \mathcal{A}[\![a_1]\!]\sigma$. Hence $(\sigma, \mathbf{true}) \in \mathcal{B}[\![a_0 = a_1]\!]$. Therefore

$$(\sigma, \mathbf{true}) \in \mathcal{B}[\![a_0 = a_1]\!] \iff \langle a_0 = a_1, \sigma\rangle \to \mathbf{true}.$$

Similarly,

$$(\sigma, \mathbf{false}) \in \mathcal{B}[\![a_0 = a_1]\!] \iff \langle a_0 = a_1, \sigma\rangle \to \mathbf{false}.$$

It follows that

$$\mathcal{B}[\![a_0 = a_1]\!] = \{(\sigma, t) \mid \langle a_0 = a_1, \sigma\rangle \to t\}.$$

$b \equiv b_0 \wedge b_1$: Let b_0, b_1 be boolean expressions. Assume $P(b_0)$ and $P(b_1)$. By definition, we have

$$(\sigma, t) \in \mathcal{B}[\![b_0 \wedge b_1]\!] \iff \sigma \in \Sigma \;\&\; \exists t_0, t_1.\; t = t_0 \wedge_T t_1 \;\&\; (\sigma, t_0) \in \mathcal{B}[\![b_0]\!] \;\&\; (\sigma, t_1) \in \mathcal{B}[\![b_1]\!].$$

Thus, supposing $(\sigma, t) \in \mathcal{B}[\![b_0 \wedge b_1]\!]$, there are t_0, t_1 such that $(\sigma, t_0) \in \mathcal{B}[\![b_0]\!]$ and $(\sigma, t_1) \in \mathcal{B}[\![b_1]\!]$. From the assumptions $P(b_0)$ and $P(b_1)$ we obtain

$$\langle b_0, \sigma\rangle \to t_0 \quad \text{and} \quad \langle b_1, \sigma\rangle \to t_1.$$

Thus we can derive $\langle b_0 \wedge b_1, \sigma\rangle \to t$ where $t = t_0 \wedge_T t_1$. Conversely, any derivation of $\langle b_0 \wedge b_1, \sigma\rangle \to t$ must have the form

$$\frac{\vdots \qquad \vdots}{\langle b_0, \sigma\rangle \to t_0 \quad \langle b_1, \sigma\rangle \to t_1}$$
$$\langle b_0 \wedge b_1, \sigma\rangle \to t$$

for some t_0, t_1 such that $t = t_0 \wedge_T t_1$. From the $P(b_0)$ and $P(b_1)$, we obtain $(\sigma, t_0) \in \mathcal{B}[\![b_0]\!]$ and $(\sigma, t_1) \in \mathcal{B}[\![b_1]\!]$. Hence $(\sigma, t) \in \mathcal{B}[\![b]\!]$.

As remarked the other cases of the induction are similar. □

Exercise 5.5 The proofs above involve considering the form of derivations. Alternative proofs can be obtained by a combination of structural induction and rule induction. For example, show

1. $\{(\sigma, n) \mid \langle a, \sigma\rangle \to n\} \subseteq \mathcal{A}[\![a]\!]$,
2. $\mathcal{A}[\![a]\!] \subseteq \{(\sigma, n) \mid \langle a, \sigma\rangle \to n\}$,

for all arithmetic expressions a by using rule induction on the operational semantics of arithmetic expressions for 1 and structural induction on arithmetic expressions for 2. □

Now we can check that the denotational semantics of commands agrees with their operational semantics:

Lemma 5.6 *For all commands c and states σ, σ',*

$$\langle c, \sigma\rangle \to \sigma' \Rightarrow (\sigma, \sigma') \in C[\![c]\!].$$

Proof: We use rule-induction on the operational semantics of commands, as stated in Section 4.3.3. For $c \in \mathbf{Com}$ and $\sigma, \sigma' \in \Sigma$, define

$$P(c, \sigma, \sigma') \iff_{def} (\sigma, \sigma') \in C[\![c]\!].$$

If we can show P is closed under the rules for the execution of commands, in the sense of Section 4.3.3, then

$$\langle c, \sigma\rangle \to \sigma' \Rightarrow P(c, \sigma, \sigma')$$

for any command c and states σ, σ'. We check only one clause in Section 4.3.3, that associated with while-loops in the case in which the condition evaluates to true. Recall it is:

$$\frac{\langle b, \sigma\rangle \to \mathbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle w, \sigma''\rangle \to \sigma'}{\langle w, \sigma\rangle \to \sigma'}$$

where we abbreviate $w \equiv \textbf{while } b \textbf{ do } c$. Following the scheme of Section 4.3.3, assume

$$\langle b, \sigma \rangle \to \textbf{true} \;\&\; \langle c, \sigma \rangle \to \sigma'' \;\&\; P(c, \sigma, \sigma'') \;\&\; \langle w, \sigma'' \rangle \to \sigma' \;\&\; P(w, \sigma'', \sigma').$$

By Lemma 5.4

$$\mathcal{B}[\![b]\!]\sigma = \textbf{true}.$$

From the meaning of P we obtain directly that

$$\mathcal{C}[\![c]\!]\sigma = \sigma'' \text{ and } \mathcal{C}[\![w]\!]\sigma'' = \sigma'.$$

Now, from the definition of the denotational semantics, we see

$$\mathcal{C}[\![w]\!]\sigma = \mathcal{C}[\![c; w]\!]\sigma = \mathcal{C}[\![w]\!](\mathcal{C}[\![c]\!]\sigma) = \mathcal{C}[\![w]\!]\sigma'' = \sigma'.$$

But $\mathcal{C}[\![w]\!]\sigma = \sigma'$ means $P(w, \sigma, \sigma')$ *i.e.* P holds for the consequence of the rule. Hence P is closed under this rule. By similar arguments, P is closed under the other rules for the execution of commands (Exercise!). Hence by rule induction we have proved the lemma. □

The next theorem, showing the equivalence of operational and denotational semantics for commands, is proved by structural induction with a use of mathematical induction inside one case, that for while-loops.

Theorem 5.7 *For all commands c*

$$\mathcal{C}[\![c]\!] = \{(\sigma, \sigma') \mid \langle c, \sigma \rangle \to \sigma'\}.$$

Proof: The theorem can clearly be restated as: for all commands c

$$(\sigma, \sigma') \in \mathcal{C}[\![c]\!] \iff \langle c, \sigma \rangle \to \sigma'.$$

for all states σ, σ'. Notice Lemma 5.6 gives the "$\Leftarrow$" direction of the equivalence.

We proceed by structural induction on commands c, taking

$$\forall \sigma, \sigma' \in \Sigma. (\sigma, \sigma') \in \mathcal{C}[\![c]\!] \iff \langle c, \sigma \rangle \to \sigma'.$$

as induction hypothesis.

$c \equiv \textbf{skip}$: By definition, $\mathcal{C}[\![\textbf{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$. Thus if $(\sigma, \sigma) \in \mathcal{C}[\![c]\!]$ then $\sigma' = \sigma$ so $\langle \textbf{skip}, \sigma \rangle \to \sigma'$ by the rule for **skip**. The induction hypothesis holds in this case.

$c \equiv X := a$: Suppose $(\sigma, \sigma') \in \mathcal{C}[\![X := a]\!]$. Then $\sigma' = \sigma[n/X]$ where $n = \mathcal{A}[\![a]\!]\sigma$. By Lemma 5.3, $\langle a, \sigma \rangle \to n$. Hence $\langle c, \sigma \rangle \to \sigma'$. The induction hypothesis holds in this case.

$c \equiv c_0; c_1$: Assume the induction hypothesis holds for c_0 and c_1. Suppose $(\sigma, \sigma') \in \mathcal{C}[\![c]\!]$. Then there is some state σ'' for which $(\sigma, \sigma'') \in \mathcal{C}[\![c_0]\!]$ and $(\sigma'', \sigma') \in \mathcal{C}[\![c_1]\!]$. By the induction hypothesis for commands c_0 and c_1 we know

$$\langle c_0, \sigma \rangle \to \sigma'' \text{ and } \langle c_1, \sigma'' \rangle \to \sigma'.$$

Hence $\langle c_0; c_1, \sigma \rangle \to \sigma'$ for the rules for the operational semantics of commands. Thus the induction hypothesis holds for c.

$c \equiv \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1$: Assume the induction hypothesis holds for c_0 and c_1. Recall that

$$\begin{aligned}\mathcal{C}[\![c]\!] =& \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{true} \ \&\ (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\} \cup \\ & \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{false} \ \&\ (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\}.\end{aligned}$$

So, if $(\sigma, \sigma') \in \mathcal{C}[\![c]\!]$ then either

(i) $\mathcal{B}[\![b]\!]\sigma = \textbf{true}$ and $(\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]$, or

(ii) $\mathcal{B}[\![b]\!]\sigma = \textbf{false}$ and $(\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]$.

Suppose (i). Then $\langle b, \sigma \rangle \to \textbf{true}$ by Lemma 5.4, and $\langle c_0, \sigma \rangle \to \sigma'$ because the induction hypothesis holds for c_0. From the rules for conditionals in the operational semantics of commands we obtain $\langle c, \sigma \rangle \to \sigma'$. Supposing (ii), we can arrive at the conclusion in essentially the same way. Thus the induction hypothesis holds for c.

$c \equiv \textbf{while } b \textbf{ do } c_0$: Assume the induction hypothesis holds for c_0. Recall that

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c_0]\!] = \mathit{fix}(\Gamma)$$

where

$$\begin{aligned}\Gamma(\varphi) =& \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{true} \ \&\ (\sigma, \sigma') \in \varphi \circ \mathcal{C}[\![c_0]\!]\} \cup \\ & \{(\sigma, \sigma) \mid \mathcal{B}[\![b]\!]\sigma = \textbf{false}\}.\end{aligned}$$

So, writing θ_n for $\Gamma^n(\emptyset)$, we have

$$\mathcal{C}[\![c]\!] = \bigcup_{n \in \omega} \theta_n$$

where

$$\begin{aligned}\theta_0 =& \emptyset, \\ \theta_{n+1} =& \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{true} \ \&\ (\sigma, \sigma') \in \theta_n \circ \mathcal{C}[\![c_0]\!]\} \cup \\ & \{(\sigma, \sigma) \mid \mathcal{B}[\![b]\!]\sigma = \textbf{false}.\}\end{aligned}$$

We shall show by mathematical induction that

$$\forall \sigma, \sigma' \in \Sigma.\ (\sigma, \sigma') \in \theta_n \Rightarrow \langle c, \sigma \rangle \to \sigma' \tag{1}$$

for all $n \in \omega$. It then follows, of course, that $(\sigma, \sigma') \in \mathcal{C}[\![c]\!] \iff \langle c, \sigma\rangle \to \sigma'$ for states σ, σ'.

We start the mathematical induction on the induction hypothesis (1).
Base case $n = 0$: When $n = 0$, $\theta_0 = \emptyset$ so that induction hypothesis is vacuously true.
Induction Step: We assume (1) holds for an arbitrary $n \in \omega$ and attempt to prove

$$(\sigma, \sigma') \in \theta_{n+1} \Rightarrow \langle c, \sigma\rangle \to \sigma'$$

for any states σ, σ'.

Assume $(\sigma, \sigma') \in \theta_{n+1}$. Then either
(i) $\mathcal{B}[\![b]\!]\sigma = \mathbf{true}$ and $(\sigma, \sigma') \in \theta_n \circ \mathcal{C}[\![c_0]\!]$, or
(ii) $\mathcal{B}[\![b]\!]\sigma = \mathbf{false}$ and $\sigma' = \sigma$.
Assume (i). Then $\langle b, \sigma\rangle \to \mathbf{true}$ by Lemma 5.4. Also $(\sigma, \sigma'') \in \mathcal{C}[\![c_0]\!]$ and $(\sigma'', \sigma') \in \theta_n$ for some state σ''. From the induction hypothesis (1) we obtain $\langle c, \sigma''\rangle \to \sigma'$. By assumption of the structural induction hypothesis for c_0, we have $\langle c_0, \sigma\rangle \to \sigma''$. By the rule for while-loops we obtain $\langle c, \sigma\rangle \to \sigma'$.
Assume (ii). As $\mathcal{B}[\![b]\!] = \mathbf{false}$, by Lemma 5.4, we obtain $\langle b, \sigma\rangle \to \mathbf{false}$. Also $\sigma' = \sigma$ so $\langle c, \sigma\rangle \to \sigma$. In this case the induction hypothesis holds.
This establishes the induction hypothesis (1) for $n + 1$.
By mathematical induction we conclude (1) holds for all n. Consequently:

$$(\sigma, \sigma') \in \mathcal{C}[\![c]\!] \Rightarrow \langle c, \sigma\rangle \to \sigma'$$

for all states σ, σ' in the case where $c \equiv \mathbf{while}\ b\ \mathbf{do}\ c_0$.
Finally, by structural induction, we have proved the theorem. □

Exercise 5.8 Let $w \equiv \mathbf{while}\ b\ \mathbf{do}\ c$. Prove that

$$\mathcal{C}[\![w]\!]\sigma = \sigma' \text{ iff } \mathcal{B}[\![b]\!]\sigma = \mathbf{false}\ \&\ \sigma = \sigma'$$

or

$$\exists \sigma_0, \cdots, \sigma_n \in \Sigma.$$
$$\sigma = \sigma_0\ \&\ \sigma' = \sigma_n\ \&\ \mathcal{B}[\![b]\!]\sigma_n = \mathbf{false}\ \&$$
$$\forall i (0 \le i < n).\ \mathcal{B}[\![b]\!]\sigma_i = \mathbf{true}\ \&\ \mathcal{C}[\![c]\!]\sigma_i = \sigma_{i+1}.$$

(The proof from left to right uses induction on the $\Gamma^n(\emptyset)$ used in building up the denotation of w; the proof from right to left uses induction on the length of the chain of states.) □

Exercise 5.9 The syntax of commands of a simple imperative language with a repeat construct is given by

$$c ::= X := e \mid c_0; c_1 \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{repeat } c \textbf{ until } b$$

where X is a location, e is an arithmetic expression, b is a boolean expression and c, c_0, c_1 range over commands. From your understanding of how such commands behave explain how to change the semantics of while programs to that of repeat programs to give:
(i) an operational semantics in the form of rules to generate transitions of the form $\langle c, \sigma\rangle \to \sigma'$ meaning the execution of c from state σ terminates in state σ';
(ii) a denotational semantics for commands in which each command c is denoted by a partial function $\mathcal{C}[\![c]\!]$ from states to states;
(iii) sketch the proof of the equivalence between the operational and denotational semantics, that $\langle c, \sigma\rangle \to \sigma'$ iff $\mathcal{C}[\![c]\!]\sigma = \sigma'$, concentrating on the case where c is a repeat loop. □

5.4 Complete partial orders and continuous functions

In the last chapter we gave an elementary account of the theory of inductive definitions. We have shown how it can be used to give a denotational semantics for **IMP**. In practice very few recursive definitions can be viewed straightforwardly as least fixed points of operators on sets, and they are best tackled using the more abstract ideas of complete partial orders and continuous functions, the standard tools of denotational semantics. We can approach this framework from that of inductive definitions. In this way it is hoped to make the more abstract ideas of complete partial orders more accessible and show the close tie-up between them and the more concrete notions in operational semantics.

Suppose we have a set of rule instances R of the form (X/y). We saw how R determines an operator $\widehat{R}$ on sets, which given a set B results in a set

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

and how the operator $\widehat{R}$ has a least fixed point

$$fix(\widehat{R}) =_{def} \bigcup_{n \in \omega} \widehat{R}^n(\emptyset)$$

formed by taking the union of the chain of sets

$$\emptyset \subseteq \widehat{R}(\emptyset) \subseteq \cdots \subseteq \widehat{R}^n(\emptyset) \subseteq \cdots.$$

It is a fixed point in the sense that

$$\widehat{R}(\mathit{fix}(\widehat{R})) = \mathit{fix}(\widehat{R}),$$

and it is the least fixed point because $\mathit{fix}(\widehat{R})$ is included in any fixed point B, *i.e.*

$$\widehat{R}(B) = B \Rightarrow \mathit{fix}(\widehat{R}) \subseteq B.$$

In fact Proposition 4.12 of Section 4.4 shows that $\mathit{fix}(\widehat{R})$ was the least R-closed set, where we can characterise an R-closed set as one B for which

$$\widehat{R}(B) \subseteq B.$$

In this way we can obtain, by choosing appropriate rule instances R, a solution to the recursive equation needed for a denotation of the while-loop. However it pays to be more general, and extract from the example above the essential mathematical properties we used to obtain a least fixed point. This leads to the notions of complete partial order and continuous functions.

The very idea of "least" only made sense because of the inclusion, or subset, relation. In its place we take the more general idea of partial order.

Definition: A *partial order* (p.o.) is a set P on which there is a binary relation $\sqsubseteq$ which is:

(i) relexive: $\forall p \in P.\ p \sqsubseteq p$
(ii) transitive: $\forall p, q, r \in P.\ p \sqsubseteq q\ \&\ q \sqsubseteq r \Rightarrow p \sqsubseteq r$
(iii) antisymmetric: $\forall p, q \in P.\ p \sqsubseteq q\ \&\ q \sqsubseteq p \Rightarrow p = q$.

But not all partial orders support the constructions we did on sets. In constructing the least fixed point we formed the union $\bigcup_{n\in\omega} A_n$ of a ω-chain $A_0 \subseteq A_1 \subseteq \cdots A_n \subseteq \cdots$ which started at $\emptyset$—the least set. Union on sets, ordered by inclusion, generalises to the notion of least upper bound on partial orders—we only require them to exist for such increasing chains indexed by ω. Translating these properties to partial orders, we arrive at the definition of a complete partial order.

Definition: For a partial order $(P, \sqsubseteq)$ and subset $X \subseteq P$ say p is an *upper bound* of X iff

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

Say p is a *least upper bound* (lub) of X iff
(i) p is an upper bound of X, and
(ii) for all upper bounds q of X, $p \sqsubseteq q$.

When a subset X of a partial order has a least upper bound we shall write it as $\bigsqcup X$. We write $\bigsqcup \{d_1, \cdots, d_m\}$ as $d_1 \sqcup \cdots \sqcup d_m$.

Definition: Let $(D, \sqsubseteq_D)$ be a partial order.

An *ω-chain* of the partial order is an increasing chain $d_0 \sqsubseteq_D d_1 \sqsubseteq_D \cdots \sqsubseteq_D d_n \sqsubseteq_D \cdots$ of elements of the partial order.

The partial order $(D, \sqsubseteq_D)$ is a *complete partial order* (abbreviated to cpo) if it has lubs of all ω-chains $d_0 \sqsubseteq_D d_1 \sqsubseteq_D \cdots \sqsubseteq_D d_n \sqsubseteq_D \cdots$, *i.e.* any increasing chain $\{d_n \mid n \in \omega\}$ of elements in D has a least upper bound $\bigsqcup\{d_n \mid n \in \omega\}$ in D, often written as $\bigsqcup_{n\in\omega} d_n$.

We say $(D, \sqsubseteq_D)$ is a cpo *with bottom* if it is a cpo which has a least element $\bot_D$ (called "bottom").[1]

Notation: In future we shall often write the ordering of a cpo $(D, \sqsubseteq_D)$ as simply $\sqsubseteq$, and its bottom element, when it has one, as just $\bot$. The context generally makes clear to which cpo we refer.

Notice that any set ordered by the identity relation forms a cpo, certainly without a bottom element. Such cpo's are called *discrete*, or *flat*.

Exercise 5.10 Show $(\mathcal{P}ow(X), \subseteq)$ is a cpo with bottom, for any set X. Show the set of partial functions $\Sigma \rightharpoonup \Sigma$ ordered by $\subseteq$ forms a cpo with bottom. □

The counterpart of an operation on sets is a function $f : D \to D$ from a cpo D back to D. We require such a function to respect the ordering on D in a certain way. To motivate these properties we consider the operator defined from the rule instances R. Suppose

$$B_0 \subseteq B_1 \subseteq \cdots B_n \subseteq \cdots.$$

Then

$$\widehat{R}(B_0) \subseteq \widehat{R}(B_1) \subseteq \cdots \widehat{R}(B_n) \subseteq \cdots$$

is an increasing chain of sets too. This is because $\widehat{R}$ is monotonic in the sense that

$$B \subseteq C \Rightarrow \widehat{R}(B) \subseteq \widehat{R}(C).$$

By monotonicity, as each $B_n \subseteq \bigcup_{n\in\omega} B_n$,

$$\bigcup_{n\in\omega} \widehat{R}(B_n) \subseteq \widehat{R}(\bigcup_{n\in\omega} B_n).$$

In fact, the converse inclusion, and so equality, holds too because of the finitary nature of rule instances. Suppose $y \in \widehat{R}(\bigcup_{n\in\omega} B_n)$. Then $(X/y) \in R$ for some *finite* set

[1] The cpo's here are commonly called (bottomless) ω-cpo's, or predomains.

$X \subseteq \bigcup_{n\in\omega} B_n$. Because X is finite, $X \subseteq B_n$ for some n. Hence $y \in \widehat{R}(B_n)$. Thus $y \in \bigcup_{n\in\omega} \widehat{R}(B_n)$. We have proved that $\widehat{R}$ is *continuous* in the sense that

$$\bigcup_{n\in\omega} \widehat{R}(B_n) = \widehat{R}(\bigcup_{n\in\omega} B_n)$$

for any increasing chain $B_0 \subseteq \cdots \subseteq B_n \subseteq \cdots$. This followed because the rules are *finitary* *i.e.* each rule (X/y) involves only a finite set of premises X.

We can adopt these properties to define the continuous functions between a pair of cpos.

Definition: A function $f : D \to E$ between cpos D and E is *monotonic* iff

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

Such a function is *continuous* iff it is monotonic and for all chains $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots$ in D we have

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

An important consequence of this definition is that any continuous function from a cpo with bottom to itself has a least fixed point, in a way which generalises that of operators on sets in Section 4.4. In fact we can catch the notion of a set closed under rules with the order-theoretic notion of a prefixed point (Recall a set B was closed under rule instances R iff $\widehat{R}(B) \subseteq B$).

Definition: Let $f : D \to D$ be a continuous function on a cpo D. A *fixed point* of f is an element d of D such that $f(d) = d$. A *prefixed point* of f is an element d of D such that $f(d) \sqsubseteq d$.

The following simple, but important, theorem gives an explicit construction $fix(f)$ of the least fixed point of a continuous function f on a cpo D.

Theorem 5.11 *(Fixed-Point Theorem)*
Let $f : D \to D$ be a continuous function on a cpo with bottom D. Define

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

Then $fix(f)$ is a fixed point of f and the least prefixed point of f i.e.
(i) $f(fix(f)) = fix(f)$ and (ii) if $f(d) \sqsubseteq d$ then $fix(f) \sqsubseteq d$. Consequently $fix(f)$ is the least fixed point of f.

Proof:
(i) By continuity

$$\begin{aligned} f(\mathit{fix}(f)) &= f(\bigsqcup_{n\in\omega} f^n(\bot)) \\ &= \bigsqcup_{n\in\omega} f^{n+1}(\bot) \\ &= (\bigsqcup_{n\in\omega} f^{n+1}(\bot)) \sqcup \{\bot\} \\ &= \bigsqcup_{n\in\omega} f^n(\bot) \\ &= \mathit{fix}(f). \end{aligned}$$

Thus $\mathit{fix}(f)$ is a fixed point.
(ii) Suppose d is a prefixed point. Certainly $\bot \sqsubseteq d$. By monotonicity $f(\bot) \sqsubseteq f(d)$. But d is prefixed point, *i.e.* $f(d) \sqsubseteq d$, so $f(\bot) \sqsubseteq d$, and by induction $f^n(\bot) \sqsubseteq d$. Thus, $\mathit{fix}(f) = \bigsqcup_{n\in\omega} f^n(\bot) \sqsubseteq d$.

As fixed points are certainly prefixed points, $\mathit{fix}(f)$ is the least fixed point of f. □

We say a little about the intuition behind complete partial orders and continuous functions, an intuition which will be discussed further and pinned down more precisely in later chapters. Complete partial orders correspond to types of data, data that can be used as input or output to a computation. Computable functions are modelled as continuous functions between them. The elements of a cpo are thought of as points of information and the ordering $x \sqsubseteq y$ as meaning x approximates y (or, x is less or the same information as y)—so $\bot$ is the point of least information.

We can recast, into this general framework, the method by which we gave a denotational semantics to **IMP**. We denoted a command by a partial function from states to states Σ. On the face of it this does not square with the idea that the function computed by a command should be continuous. However partial functions on states can be viewed as continuous total functions. We extend the states by a new element $\bot$ to a cpo of results $\Sigma_\bot$ ordered by

$$\bot \sqsubseteq \sigma$$

for all states σ. The cpo $\Sigma_\bot$ includes the extra element $\bot$ representing the undefined state, or more correctly null information about the state, which, as a computation progresses, can grow into the information that a particular final state is determined. It is not hard to see that the partial functions $\Sigma \rightharpoonup \Sigma$ are in 1-1 correspondence with the (total) functions $\Sigma \to \Sigma_\bot$, and that in this case any total function is continuous; the

inclusion order between partial functions corresponds to the "pointwise order"

$$f \sqsubseteq g \text{ iff } \forall \sigma \in \Sigma.\ f(\sigma) \sqsubseteq g(\sigma)$$

between functions $\Sigma \to \Sigma_\perp$. Because partial functions form a cpo so does the set of functions $[\Sigma \to \Sigma_\perp]$ ordered pointwise. Consequently, our denotational semantics can equivalently be viewed as denoting commands by elements of the cpo of continuous functions $[\Sigma \to \Sigma_\perp]$. Recall that to give the denotation of a while program we solved a recursive equation by taking the least fixed point of a continuous function on the cpo of partial functions, which now recasts to one on the cpo $[\Sigma \to \Sigma_\perp]$.

For the cpo $[\Sigma \to \Sigma_\perp]$, isomorphic to that of partial functions, more information corresponds to more input/output behaviour of a function and no information at all, $\perp$ in this cpo, corresponds to the empty partial function which contains no input/output pairs. We can think of the functions themselves as data which can be used or produced by a computation. Notice that the information about such functions comes in discrete units, the input/output pairs. Such a discreteness property is shared by a great many of the complete partial orders that arise in modelling computations. As we shall see, that computable functions should be continuous follows from the idea that the appearance of a unit of information in the output of a computable function should only depend on the presence of finitely many units of information in the input. Otherwise a computation of the function would have to make use of infinitely many units of information before yielding that unit of output. We have met this idea before; a set of rule instances determines a continuous operator when the rule instances are finitary, in that they have only finite sets of premises.

Exercise 5.12
(i) Show that the monotonic maps from Σ to $\Sigma_\perp$ are continuous and in 1-1 correspondence with the partial functions $\Sigma \rightharpoonup \Sigma$. Confirm the statement above, that a partial function is included in another iff the corresponding functions $\Sigma \to \Sigma_\perp$ are ordered pointwise.
(ii) Let D and E be cpo's. Suppose D has the property that every ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots$ is stationary, in the sense that there is an n such that $d_m = d_n$ for all $m \geq n$. Show that all monotonic functions from D to E are continuous. □

Exercise 5.13 Show that if we relax the condition that rules be finitary, and so allow rule instances with an infinite number of premises, then the operator induced by a set of rule instances need not be continuous. □

5.5 The Knaster-Tarski Theorem

In this section another abstract characterisation of least fixed points is studied. It results are only used much later, so it can be skipped at a first reading. Looking back to the last chapter, there was another characterisation of the least fixed point of an operator on sets. Recall from Exercise 4.3 of Section 4.1 that, for a set of rule instances R,

$$I_R = \bigcap \{Q \mid Q \text{ is } R\text{-closed}\}.$$

In view of Section 4.4, this can be recast as saying

$$\mathit{fix}(\widehat{R}) = \bigcap \{Q \mid \widehat{R}(Q) \subseteq Q\},$$

expressing that the least fixed point of the operator $\widehat{R}$ can be characterised as the intersection of its prefixed points. This is a special case of the *Knaster-Tarski Theorem*, a general result about the existence of least fixed points. As might be expected its statement involves a generalisation of the operation of intersection on sets to a notion dual to that least upper bound on a partial order.

Definition: For a partial order $(P, \sqsubseteq)$ and subset $X \subseteq P$ say p is an *lower bound* of X iff

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

Say p is a *greatest lower bound* (glb) of X iff
(i) p is a lower bound of X, and
(ii) for all lower bounds q of X, we have $q \sqsubseteq p$.
When a subset X of a partial order has a greatest lower bound we shall write it as $\bigsqcap X$. We write $\bigsqcap \{d_0, d_1\}$ as $d_0 \sqcap d_1$.

Just as sometimes lubs are called *suprema* (or *sups*), glbs are sometimes called *infima* (or *infs*).

Definition: A *complete lattice* is a partial order which has greatest lower bounds of arbitrary subsets.

Although we have chosen to define complete lattices as partial orders which have all greatest lower bounds we could alternatively have defined them as those partial orders with all least upper bounds, a consequence of the following exercise.

Exercise 5.14 Prove a complete lattice must also have least upper bounds of arbitrary subsets. Deduce that if $(L, \sqsubseteq)$ is a complete lattice then so is $(L, \sqsupseteq)$, ordered by the converse relation. □

Theorem 5.15 *(Knaster-Tarski Theorem for minimum fixed points)*
Let $(L, \sqsubseteq)$ be a complete lattice. Let $f : L \to L$ be a monotonic function, i.e. such that if $x \sqsubseteq y$ then $f(x) \sqsubseteq f(y)$ (but not necessarily continuous). Define

$$m = \bigsqcap \{x \in L \mid f(x) \sqsubseteq x\}.$$

Then m is a fixed point of f and the least prefixed point of f.

Proof: Write $X = \{x \in L \mid f(x) \sqsubseteq x\}$. As above, define $m = \bigsqcap X$. Let $x \in X$. Certainly $m \sqsubseteq x$. Hence $f(m) \sqsubseteq f(x)$ by the monotonicity of f. But $f(x) \sqsubseteq x$ because $x \in X$. So $f(m) \sqsubseteq x$ for any $x \in X$. It follows that $f(m) \sqsubseteq \bigsqcap X = m$. This makes m a prefixed point and, from its definition, it is clearly the least one. As $f(m) \sqsubseteq m$ we obtain $f(f(m)) \sqsubseteq f(m)$ from the monotonicity of f. This ensures $f(m) \in X$ which entails $m \sqsubseteq f(m)$. Thus $f(m) = m$. We conclude that m is indeed a fixed point and is the least prefixed point of f. □

As a corollary we can show that a monotonic function on a complete lattice has a *maximum* fixed point.

Theorem 5.16 *(Knaster-Tarski Theorem for maximum fixed points)*
Let $(L, \sqsubseteq)$ be a complete lattice. Let $f : L \to L$ be a monotonic function. Define

$$M = \bigsqcup \{x \in L \mid x \sqsubseteq f(x)\}.$$

Then M is a fixed point of f and the greatest postfixed point of f. (A postfixed point is an element x such that $x \sqsubseteq f(x)$.)

Proof: This follows from the theorem for the minimum-fixed-point case by noticing that a monotonic function on $(L, \sqsubseteq)$ is also a monotonic function on the complete lattice $(L, \sqsupseteq)$. □

The Knaster-Tarski Theorem is important because it applies to any monotonic function on a complete lattice. However most of the time we will be concerned with least fixed points of continuous functions which we shall construct by the techniques of the previous section, as least upper bounds of ω-chains in a cpo.

5.6 Further reading

This chapter has given an example of a denotational semantics. Later chapters will expand on the range and power of the denotational method. Further elementary material

can be found in the books by Bird [21], Loeckx and Sieber [58], Schmidt [88], and Stoy [95] (though the latter bases its treatment on complete lattices instead of complete partial orders). A harder but very thorough book is that by de Bakker [13]. The denotational semantics of **IMP** has come at a price, the more abstract use of least fixed points in place of rules. However there is also a gain. By casting its meaning within the framework of cpo's and continuous functions **IMP** becomes amenable to the techniques there. The book [69] has several examples of applications to the language of while programs.