University of Cambridge
Computer Science Tripos Part IB
Lent Term 1996

Semantics of Programming Languages

Dr P. N. Benton

Contents

Chapter 1

Introduction

1.1 Formal Semantics

This course is about understanding and reasoning about programs and programming languages. Any programming language can be studied at a number of different (but related) levels, amongst which it is convenient to distinguish:

Syntax The alphabet of symbols used to write programs and some description (e.g. BNF) of the way in which those symbols may be combined to give well-formed expressions, commands, programs, etc. of the language.

Semantics The *meaning* of each expression, command, program, etc. This means some description of how programs behave when they are actually executed.

Pragmatics The way in which the language is actually implemented (e.g. compiled or interpreted, separate compilation, garbage collection) and used (e.g. typical programming techniques, suitability for different problem domains).

We shall be concerned with the second of these aspects – giving desciptions of the run-time behaviour of programs – and we shall use mathematical and logical methods to give these descriptions in a formal and rigorous way.

A formal semantics can have many uses:

- It can serve simply as a specification of how programs should behave. This is obviously of value to the compiler writer and, if the semantics is sufficiently readable, to the programmer.
- The very act of trying to give a formal semantics can help the language designer to spot mistakes and ambiguities in an informal account of how programs should execute.
- A formal semantics can be used to obtain or verify reasoning principles which may be used to prove that programs satisfy their specifications or that two programs are equivalent. This is vital if one wishes formally to verify or derive software, as is increasingly done in, for example, 'safety-critical' applications. Even if one does not wish to go to the trouble and expense of giving a completely formal proof of program correctness, if programmers are aware of the reasoning principles which they would use were they to attempt such a proof then the informal reasoning which they use whilst writing code is much more likely to be sound.

- Sophisticated program analyses and transformations, such as those used in highly optimising compilers, are not only verified with repect to a formal semantics, but are very often designed and expressed in terms of the semantics.

- A mathematical analysis of computational and programming language constructs which is independent of any particular programming language allows one to simplify and generalise. This then feeds back into computer science in the form of new programming languages and language features. For example, ML and other similar languages are based on the *lambda calculus*, which is a mathematical model of computation which predates computer programming. Similarly, some implementations of ML-like languages are based on a translation of programs into a language of *combinators*, which originally arose in mathematical logic and has since been refined in various ways to suit the needs of language designers and implementers.

- Finally, obtaining a deeper understanding of the basic nature of computation is a fascinating and worthwhile intellectual activity in its own right. Fundamental scientific research has a cultural value beyond its immediate technological applications.

Historically, semantics have been given in three main styles:

Operational Semantics specifies how programs should be executed, for example by giving a translation of programs into some simpler abstract machine language. In this course we will use a style of operational semantics called *structural operational semantics*, due to Gordon Plotkin, in which evaluation and transition relations are defined directly by induction on the syntax of the language.

Denotational Semantics gives the meaning of programs as elements of some suitable mathematical structure. This style of semantics was pioneered by Christopher Strachey and Dana Scott in the late 60s and early 70s, making use of the theory of certain special partially ordered sets.

Axiomatic Semantics defines the meaning of each programming construct by giving proof rules for it in some suitable program logic. This style of semantics was introduced by Robert Floyd and Tony Hoare. You will learn more about this in the Part II course on Specification and Verification.

Of course, these different styles of semantics each have advantages and disadvantages for particular purposes. We shall concentrate on the first two styles and the relationships between them, though there is some material in Chapter 5 on axiomatic semantics.

1.2 Outline of Course

In this course we will study the operational and denotational semantics of a simple imperative programming language which we call IMP. Since we will be making considerable use of induction, we start by recalling some basic material on inductive definitions and proofs. We then define the syntax of IMP and give it an operational semantics using transition relations. Next we give an alternative presentation of the operational semantics in the style known as 'natural semantics' and relate this to the first semantics.

We then turn to the denotational semantics of IMP. After introducing the basic mathematical concepts which we shall need, we show how IMP programs may be given meaning

as functions between certain ordered sets and relate this to the operational semantics which we gave earlier.

Having studied the operational and denotational semantics of IMP in considerable detail, we then look briefly at some slightly more advanced topics: how to treat a non-deterministic version of IMP, how to use continuations to give a denotational semantics to a version of IMP with some non-local control operators, and how to use the denotational semantics of IMP to justify Floyd-Hoare logic proof rules for the language.

Finally, an appendix contains some material on a functional view of proofs of semantic equivalence. (This is highly non-examinable and merely included because I thought it might be amusing.)

We will make continual use of the ML programming language. This is because ML makes it possible (almost!) to implement directly many of the mathematical ideas which we shall be using to understand IMP. It is hoped that this alternative, more concrete and computational, viewpoint will make understanding the mathematics easier. The use of ML should, however, only be treated as an intuitive aid to understanding the real mathematical semantics. Any more formal understanding of the relationship between IMP and the various bits of ML which we shall present would involve giving a mathematical semantics to ML and this requires rather more sophisticated ideas than we shall need in order to deal with IMP. Note that this is a slightly unusual use of a programming language. Whilst all the ML code used in these notes will be made available for students to experiment with, its real purpose is to be read, not executed. That is, it is used primarily as a language for human communication, and only incidentally as a language whereby people can control machines.

The material on using ML to implement semantic ideas is all non-examinable.

At this point we should mention that ML itself *does* in fact have a completely formally specified operational semantics (the Definition of Standard ML). I strongly recommend that you read the preface to the Definition and have at least a brief look at the rest of it, so as to get some idea of how the ideas introduced in this course scale up to real-world languages.

The prerequisites for the course are merely IA Discrete Maths and some knowledge of programming. An understanding of ML is also very desirable, but not absolutely essential since ML is only used as a metaphor for the more formal semantics.

1.3 Acknowledgements

Thanks to Andy Pitts, Gordon Plotkin and Glynn Winskel, all of whose lecture notes I have liberally plundered in writing this course, and to Larry Paulson for permission to use parsing and prettyprinting code from his book 'ML for the Working Programmer' in the programs accompanying the course. Andrew Kennedy made some very useful comments on drafts of these notes. I have used John Reynolds's diagram macros and Paul Taylor's proof tree macros.

1.4 Recommended Reading

Books

- G. Winskel *The Formal Semantics of Programming Languages.* MIT Press 1993. If you're going to buy a book on semantics, this is the one to get. Dr

Winskel used to lecture this course and the book is based in part on his lecture notes.

- R. D. Tennent *Semantics of Programming Languages.* Prentice Hall International 1991.
- M. Hennessy *The Semantics of Programming Languages.* Wiley 1990.
- R. Nielson and F. Nielson *Semantics with Applications.* Wiley 1992.
- R. Harper, R. Milner and M. Tofte *The Definition of Standard ML.* MIT Press 1990.
- R. Milner and M. Tofte *Commentary on Standard ML.* MIT Press 1991.

Papers etc.

- G. D. Plotkin *A Structural Approach to Operational Semantics.* Report DAIMI FN-19 Aarhus University 1981. Available as V105 604 in the CL library.
- G. Kahn *Natural Semantics.* In K. Fuchi and M. Nivat (eds) *Programming of Future Generation Computers.* North-Holland 1988.

Chapter 2

Inductive Definitions and Proofs

This chapter recalls some mathematical background material (from the Discrete Mathematics course) which we shall be using repeatedly in this course.

2.1 Inductive Definitions

2.1.1 Introduction

Inductively defined sets arise throughout Computer Science. For example:

1. Backus-Naur form (BNF) used in the definition of the concrete syntax of programming languages, as in the following simple definition of binary numbers (with leading zeros allowed):

$$\begin{array}{rcl} \langle bit \rangle & ::= & 0 \mid 1 \\ \langle bin \rangle & ::= & \langle bit \rangle \mid \langle bit \rangle \langle bin \rangle \end{array}$$

   which says that a *bit* is either 0 or 1 and that a *bin* is either a *bit* or a *bit* followed by a *bin*.

2. Inductive datatypes in ML, such as one corresponding to the above BNF:

```
datatype BIT = Zero | One;
datatype BIN = Single of BIT | Bitstring of BIT*BIN;
```

   or the type of binary trees with integers at the nodes:

```
datatype TREE = Empty | Node of int*TREE*TREE;
```

3. The definition of various logics as collections of inference rules, such as the following rule for introducing conjunction, which should be read as 'if from a set of assumptions Γ you can prove a formula A, and from the same set of assumptions you can also prove B, then from Γ you can prove $A \wedge B$':

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\mathcal{I}$$

What all these examples have in common is that a set S (of strings, datastructures or provable sequents) is defined by a collection of rules all which have the general form 'if a_1 up to a_n are all in the set then so is a', which we will usually write as

$$\frac{a_1 \in S \qquad a_2 \in S \quad \cdots \quad a_n \in S}{a \in S}$$

using the logical rule notation

$$\frac{\langle hypotheses \rangle}{\langle conclusion \rangle}\ \langle rulename \rangle$$

so that the BNF example (which is, of course, two definitions) could be written as

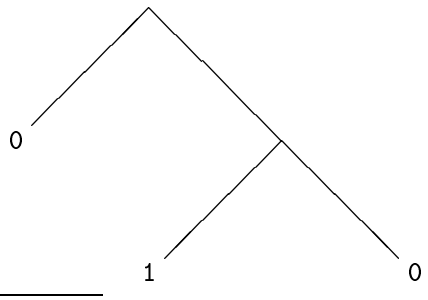
$$\frac{}{0 \in bit} \qquad \frac{}{1 \in bit}$$

to define the set *bit* and

$$\frac{b \in bit}{b \in bin} \qquad \frac{b \in bit \qquad s \in bin}{bs \in bin}$$

to define the set *bin*. Notice that we allow the case $n = 0$, i.e. no hypotheses. This simply means that the conclusion holds unconditionally. Such rules are sometimes called *axioms*. Note also that rules may contain variables (e.g. b and s in the second rule for *bin* above). Such a rule should be thought of as a *rule scheme* standing for the infinite collection of all rules arising by substituting atomic things for the variables.[1] We will also sometimes add *side conditions* to a rule scheme; these constrain the meaning of the scheme to the set of substitution instances which also satisfy the side conditions.

In this course we shall usually use the rule notation, rather than BNF, to define programming language syntax. A slight subtlety is that BNF is usually taken to define a set of *strings* (i.e. concrete syntax), whereas we shall always think of syntax as a set of *trees* (i.e. abstract syntax). This allows us to avoid any issues relating to parsing, which are completely irrelevant for this course (though they are obviously important for compiler writers). Thus the set *bin* defined above should be thought of as containing trees like this:[2]



[1] Strictly speaking, this is not quite right since we haven't said what 'atomic things' one is allowed to plug in for the variables. The correct answer is that we can only really make an inductive definition of a *subset* S of some already existing set U, and the atoms are all the elements of U. In practice, however, it doesn't usually matter exactly what U is, as long as it's large enough to contain everything of interest (in the case of binary numbers, for example, U could be the set of all finite strings of ASCII characters, or it might be all those together with all countably infinite strings or whatever). We will usually omit all mention of U, with the tacit understanding that a suitable set could easily be found were anybody to press us on the matter.

[2] So when (to save time and paper) we write syntax in a linear way, the use of parentheses is a meta-notation to indicate the intended tree structure, rather than a proper part of the abstract syntax itself.

Given an inference rule

$$\frac{a_1 \in S \qquad a_2 \in S \quad \cdots \quad a_n \in S}{a \in S}\ R$$

we say that a set S is *closed under* R if

$$((a_1 \in S) \wedge (a_2 \in S) \wedge \cdots \wedge (a_n \in S)) \Rightarrow (a \in S)$$

2.1.2 What do inductive definitions mean?

Consider the set of natural numbers $\{0, 1, 2, 3, \ldots\}$. Another way to describe this set, without using '...', is by induction. We take a constant symbol Z, which we intend to mean 0, and a unary function symbol S which is intended to represent the successor function (so $S(Z)$ represents 1, $S(S(Z))$ represents 2 and so on). The following two rules then constitute an inductive definition of the set $\mathbb{N} = \{Z, S(Z), S(S(Z)), \ldots\}$:

1. Z is a natural number. In symbols, $Z \in \mathbb{N}$, which we can can also write as an inference rule with no hypotheses:

$$\frac{}{Z \in \mathbb{N}}$$

2. If n is a natural number then $S(n)$ is a natural number. We can write this as $n \in \mathbb{N} \Rightarrow S(n) \in \mathbb{N}$ or as

$$\frac{n \in \mathbb{N}}{S(n) \in \mathbb{N}}$$

But how do these two rules *specify* the set we intend, *viz.*$\{Z, S(Z), S(S(Z)), \ldots\}$? After all, the two rules are only conditions which we want $\mathbb{N}$ to satisfy, and there are many other sets which also satisfy both conditions, such as $\mathbb{N}' = \{Z, S(Z), S(S(Z)), \ldots, \nabla, S(\nabla), S(S(\nabla)), \ldots\}$ where ∇ is just some arbitrary new symbol. The reason $\mathbb{N}'$ is not what we meant to define is that it contains a lot of extra junk which doesn't *need* to be there (such as $S(\nabla)$). Note that $\mathbb{N} \subset \mathbb{N}'$ – when we make an inductive definition such as that given above, it is understood to mean the *least* set (with respect to the subset ordering) satisfying all the clauses of the definition. Formally:

Definition 1 *Given an inductive definition comprising a set of rules $\mathcal{R}$, $\mathcal{R}$ is said to* inductively define *the set S if*

1. *S is closed under all the rules in $\mathcal{R}$*
2. *For any S' such that S' is closed under all the rules in $\mathcal{R}$, $S \subseteq S'$.*

It is not, however, immediately clear that there *is* a unique smallest set satisfying any inductive definition, i.e. that inductive definitions really do define something.

Proposition 1 (Uniqueness) *Given an inductive definition in the form of a set of rules $\mathcal{R}$, the set defined by $\mathcal{R}$, if it exists, is unique.*

Proof. Assume that S_1 and S_2 both satisfy the conditions of Definition 1 above. Because S_1 satisfies part 1 of the definition and S_2 satisfies part 2, we have $S_1 \subseteq S_2$. A symmetric argument yields $S_2 \subseteq S_1$, so that $S_1 = S_2$. □

At this point it's convenient to introduce a new notion, that of the operator associated with a set of rules. If $\mathcal{R}$ is a set of rules, indexed by a set I:

$$\mathcal{R} \;=\; \{R_i \mid i \in I\}$$

where the rule R_i has the form

$$\frac{h_{i,1} \quad h_{i,2} \quad \cdots \quad h_{i,n_i}}{c_i}\; R_i$$

then $\Phi_{\mathcal{R}}$ is an operator which takes sets to sets, defined by

$$\Phi_{\mathcal{R}}(T) = \{c_i \mid (h_{i,1} \in T) \wedge \cdots \wedge (h_{i,n_i} \in T)\}$$

The following two properties of Φ are immediate from the definition:

Lemma 2

1. *For any set of rules $\mathcal{R}$, $\Phi_{\mathcal{R}}$ is monotonic. That is*

$$\textit{if } X \subseteq Y \textit{ then } \Phi_{\mathcal{R}}(X) \subseteq \Phi_{\mathcal{R}}(Y)$$

2. *A set X is closed under all the rules in $\mathcal{R}$ if and only if $\Phi_{\mathcal{R}}(X) \subseteq X$. If this is the case, we say X is a* prefixed point *of $\Phi_{\mathcal{R}}$.*

□

Lemma 3 *If $\mathcal{R}$ is a set of rules and $\{S_i \mid i \in I\}$ is a collection of sets (indexed by the set I) such that for each $i \in I$, S_i is closed under all the rules in $\mathcal{R}$, then the set $\bigcap_{i\in I} S_i$ is also closed under all the rules in $\mathcal{R}$.*

Proof.

$$\begin{aligned}
\forall i. \bigcap_{j\in I} S_j \subseteq S_i \quad &\Rightarrow \quad \forall i. \Phi_{\mathcal{R}}(\bigcap_{j\in I} S_j) \subseteq \Phi_{\mathcal{R}}(S_i) \\
&\Rightarrow \quad \forall i. \Phi_{\mathcal{R}}(\bigcap_{j\in I} S_j) \subseteq S_i \\
&\Rightarrow \quad \Phi_{\mathcal{R}}(\bigcap_{j\in I} S_j) \subseteq \bigcap_{i\in I} S_i
\end{aligned}$$

where the first inclusion is an obvious property of intersection, the first implication follows by monotonicity of $\Phi_{\mathcal{R}}$, the second by the fact that each S_i is a prefixed point of $\Phi_{\mathcal{R}}$ and the last by another property of intersection. □

Proposition 4 (Existence) *If $\mathcal{R}$ is a set of rules, then the set*

$$S \stackrel{\text{def}}{=} \bigcap\{S' \mid S' \textit{ is closed under all the rules in } \mathcal{R}\}$$

is inductively defined by $\mathcal{R}$.

Proof. By Lemma 3, S is closed under all the rules in $\mathcal{R}$ and so satisfies part 1 of Definition 1. To see that it also satisfies part 2, let S'' be closed under all the rules in $\mathcal{R}$. It should then be obvious that since $S'' \in \{S' \mid S' \text{ closed under } \mathcal{R}\}$, we have

$$S \;=\; \bigcap\{S' \mid S' \text{ closed under } \mathcal{R}\} \;\subseteq\; S''$$

as required. □

So, taking Propositions 1 and 4 together, we see that inductive definitions really do make sense. Proposition 4 says exactly that $S = \bigcap\{S' \mid \Phi_{\mathcal{R}}(S') \subseteq S'\}$ is the *least prefixed point* of $\Phi_{\mathcal{R}}$. It's worth noting the following:

Proposition 5 *$S = \bigcap\{S' \mid \Phi_{\mathcal{R}}(S') \subseteq S'\}$ is the* least fixed point *of $\Phi_{\mathcal{R}}$. That is*

1. *$\Phi_{\mathcal{R}}(S) = S$, and*
2. *If $\Phi_{\mathcal{R}}(S'') = S''$ then $S \subseteq S''$.*

Proof.

1. We already know that $\Phi_{\mathcal{R}}(S) \subseteq S$ because S is a prefixed point. Thus we want to show $S \subseteq \Phi_{\mathcal{R}}(S)$. Well, let $Z = \Phi_{\mathcal{R}}(S)$. By monotonicity applied to the fact that $Z \subseteq S$ we get that $\Phi_{\mathcal{R}}(Z) \subseteq Z$ and hence that $Z \in \{S' \mid \Phi_{\mathcal{R}}(S') \subseteq S'\}$. Thus $\bigcap\{S' \mid \Phi_{\mathcal{R}}(S') \subseteq S'\} \subseteq Z$, i.e. $S \subseteq \Phi_{\mathcal{R}}(S)$, as required.
2. If S'' is a fixed point, it is a prefixed point and hence $S \subseteq S''$ as S is the least prefixed point.

□

2.1.3 Upwards characterisation of inductively defined sets

The way we have explained the meaning of inductive definitions is in some sense 'downwards' – we start with a collection of candidates for the meaning of the definition, which are, in general, too big; the true meaning is then extracted as the intersection of all the candidates.

There is another way of describing the set defined by an inductive definition which works from the bottom up. The intuitive idea is that one builds the set up in stages, starting with the empty set and at each stage adding in those extra things which the rules say have to be there as consequences of the previous stage. The set defined by the inductive definition is then the limit of this chain of successive approximations. In the case of the definition of natural numbers, for example, we build the chain like this:

$\mathbb{N}_0$	=	$\{\}$	start with the empty set
$\mathbb{N}_1$	=	$\{Z\}$	the rule for Z says add Z without any condition
$\mathbb{N}_2$	=	$\{Z, S(Z)\}$	now the S rule says add $S(Z)$ because $Z \in \mathbb{N}_1$
$\vdots$			
$\mathbb{N}$	=	$\bigcup_{i=0}^{\infty} \mathbb{N}_i$	the (infinite) limit is the union of all the (finite) approximations

You should recognise this as the way in which the construction of the Herbrand universe of a set of clauses is explained in Dr Paulson's 'Logic and Proof' course. (Indeed, one view of pure Prolog is that it is essentially a language for making inductive definitions.)

We can make this intuitive account more formal. If $\mathcal{R}$ is a set of rules, we define the chain of approximations (inductively!) like this:

$$\begin{array}{rcl} S_0 & = & \{\} \\ S_{n+1} & = & \Phi_{\mathcal{R}}(S_n) \end{array}$$

Note that we are justified in calling this a chain, since Lemma 2 implies that

$$S_0 \subseteq S_1 \subseteq S_2 \subseteq \cdots$$

(Why?) The limit is then simply

$$S_\omega \;=\; \bigcup_{n=0}^{\infty} S_n$$

And this does actually work:

Proposition 6 *Given a set of rules $\mathcal{R}$, the set*

$$S_\omega \stackrel{\text{def}}{=} \bigcup_{n=0}^{\infty} \Phi_{\mathcal{R}}^n(\emptyset)$$

is inductively defined by $\mathcal{R}$.

Proof. There are two parts to Definition 1 and we check each in turn. Firstly, we need to check that S_ω is closed under all the rules in $\mathcal{R}$. Take a typical rule

$$\frac{h_1 \in S \quad \cdots \quad h_k \in S}{c \in S}\; R$$

and assume that $h_i \in S_\omega$ for each $1 \leq i \leq k$. Then there must be some finite approximation S_m such that $h_i \in S_m$ for each i. Then by the definition of $\Phi_{\mathcal{R}}$, $c \in S_{m+1} \subseteq S_\omega$ and we're done.

Now we have to check the second part of the definition, i.e. that S_ω is contained within any other set, call it T, which is closed under all the rules in $\mathcal{R}$. We shall establish this by mathematical induction (which we shall discuss in the next section). Firstly note that $\emptyset = S_0 \subseteq T$. This is the base case of the induction. Now assume that $S_m \subseteq T$. It's easy to see that

$$\begin{array}{rcl} S_{m+1} & = & \Phi_{\mathcal{R}}(S_m) \\ & \subseteq & \Phi_{\mathcal{R}}(T) \\ & \subseteq & T \end{array}$$

where the last inclusion follows from the fact that T is a prefixed point for $\Phi_{\mathcal{R}}$ (Lemma 2, part 2) and the middle one from the fact that $S_m \subseteq T$ and monotonicity (Lemma 2, part 1). So by mathematical induction we have that $S_n \subseteq T$ for all $n \in \mathbb{N}$, and it's then clear that S_ω, being the union of all the S_ns, is also contained in T as required. Thus S_ω is the least prefixed point, and is equal to the S we defined in the previous section. □

As another example, the meaning of the ML datatype of binary trees which we gave earlier

```
datatype TREE = Empty | Node of int*TREE*TREE;
```

can be built as the limit of the chain of approximations which starts

$$
\begin{array}{rcl}
\mathtt{TREE}_0 & = & \emptyset \\
\mathtt{TREE}_1 & = & \{\mathtt{Empty}\} \\
\mathtt{TREE}_2 & = & \{\mathtt{Empty}, \mathtt{Node}(\mathtt{0},\mathtt{Empty},\mathtt{Empty}), \mathtt{Node}(\mathtt{1},\mathtt{Empty},\mathtt{Empty}), \ldots\} \\
\mathtt{TREE}_3 & = & \{\mathtt{Empty}, \mathtt{Node}(\mathtt{0},\mathtt{Empty},\mathtt{Empty}), \mathtt{Node}(\mathtt{1},\mathtt{Empty},\mathtt{Empty}), \ldots \\
 & & \mathtt{Node}(\mathtt{0},\mathtt{Node}(\mathtt{0},\mathtt{Empty},\mathtt{Empty})), \mathtt{Node}(\mathtt{1},\mathtt{Node}(\mathtt{0},\mathtt{Empty},\mathtt{Empty})), \ldots \\
 & & \vdots \\
 & & \} \\
\vdots & &
\end{array}
$$

2.1.4 Simultaneous inductive definitions

The ideas of the previous section can be generalised to the case where a collection of sets $S_1, S_2, \ldots, S_k$ are defined by a set of rules which each look like

$$\frac{x_1 \in S_{i_1} \quad \cdots \quad x_n \in S_{i_n}}{x \in S_i}$$

For example, we might define the syntax of integer and boolean expressions in some (slightly C-like) language by rules including the following:

$$\frac{}{\underline{n} \in \mathit{Iexp}}\ n \in \mathbb{Z} \qquad \frac{}{\mathtt{true} \in \mathit{Bexp}} \qquad \frac{}{\mathtt{false} \in \mathit{Bexp}}$$

$$\frac{e_1 \in \mathit{Iexp} \qquad e_2 \in \mathit{Iexp}}{e_1 + e_2 \in \mathit{Iexp}} \qquad \frac{b_1 \in \mathit{Bexp} \qquad b_2 \in \mathit{Bexp}}{b_1 \&\& b_2 \in \mathit{Bexp}}$$

$$\frac{b \in \mathit{Bexp} \qquad e_1 \in \mathit{Iexp} \qquad e_2 \in \mathit{Iexp}}{(b ? e_1 : e_2) \in \mathit{Iexp}} \qquad \frac{e_1 \in \mathit{Iexp} \qquad e_2 \in \mathit{Iexp}}{(e_1 = e_2) \in \mathit{Bexp}}$$

Note that the integer expressions depend on the boolean expressions and vice-versa. The formal meaning of such mutually dependent inductive definitions is a generalisation of that of a single inductive definition, and is left as an exercise for the diligent reader.

2.1.5 Derivations

If the set S is defined by an inductive definition $\mathcal{R} = \{R_i \mid i \in I\}$ then each $s \in S$ is there for a reason – this is the essence of the second part of Definition 1, each such s is there because it is forced to be by some finite number of applications of rules in $\mathcal{R}$. These can be written in a tree which we call a *derivation* of the statement $s \in S$. For example, in the case of our integer and boolean expressions, the following is a typical derivation:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{3} \in \mathit{Iexp}} \qquad \dfrac{}{\mathsf{4} \in \mathit{Iexp}}}{(\mathsf{3} = \mathsf{4}) \in \mathit{Bexp}} \qquad \dfrac{}{\mathsf{5} \in \mathit{Iexp}} \qquad \dfrac{}{\mathsf{6} \in \mathit{Iexp}}}{((\mathsf{3} = \mathsf{4})\ ?\ \mathsf{5}\ :\ \mathsf{6}) \in \mathit{Iexp}}$$

There may, in general, be more than one derivation that a particular element belongs to the set. This doesn't happen in our example above since each syntactic form is the conclusion of exactly one rule.

Given a set of rules $\mathcal{R}$ defining a set S, the set of derivations in $\mathcal{R}$ is itself an inductively defined set. It is defined by the following two rules:

1. Any rule $R \in \mathcal{R}$ with no hypotheses is a derivation.
2. If $\mathcal{D}_1, \ldots, \mathcal{D}_n$ are derivations in $\mathcal{R}$ with conclusions $h_1 \in S, \ldots, h_n \in S$ respectively, and $R \in \mathcal{R}$ is a rule with hypotheses $h_1 \in S$ through to $h_n \in S$ and conclusion $c \in S$, then the following is a derivation:

$$\frac{\begin{array}{ccc}\mathcal{D}_1 & & \mathcal{D}_n \\ h_1 \in S & \cdots & h_n \in S\end{array}}{c \in S}\ R$$

2.1.6 Inductively defined functions

Assume that S is inductively defined by $\mathcal{R} = \{R_i \mid i \in I\}$ where

$$R_i \ = \ \frac{h_{i,1} \quad h_{i,2} \quad \cdots \quad h_{i,n_i}}{c_i}\ R_i$$

and that furthermore there is a unique derivation for each $s \in S$. If T is any set, then to define a function $f : S \to T$, it suffices for each i to give $f(c_i)$ in terms of the n_i values $f(h_{i,1}), \ldots, f(h_{i,n_i})$.This is, of course, the way in which one defines functions over datatypes using pattern matching and recursion in ML.[3] For example:

```
datatype NAT = Z | S of NAT;

fun double Z = Z
 |  double (S(n)) = S(S(double(n)));
```

2.2 Inductive Proofs

We now turn from defining sets to proving things about them.

2.2.1 Mathematical induction

This means induction over the natural numbers, and is somthing with which you should already be familiar. (Indeed, we have used it once already in these notes, to prove Proposition 6.)

Proposition 7 (Mathematical Induction) *Suppose that P is some property of the natural numbers, so $P \subseteq \mathbb{N}$. If P is closed under the following rules*

$$\frac{}{0 \in P} \qquad \frac{n \in P}{n+1 \in P}$$

then P is the whole of $\mathbb{N}$.

[3] This is actually a gross simplification, but never mind.

Proof. Suppose that the result is false, so that P is closed under the rules but there is some $m \in \mathbb{N}$ such that $m \notin P$. We can furthermore take m to be the smallest such number (the 'minimal criminal'). Now, since P is closed under the first rule, we have that $0 \in P$ so that $m \neq 0$. This means that $m = m' + 1$ for some $m' \in \mathbb{N}$. But now $m' \notin P$ (or else $m \in P$ by the fact that P is closed under the second rule), and m' is strictly smaller than m, which contradicts the minimality of m. So no such m exists and $P = \mathbb{N}$. □

Here's a familiar and rather trivial example of a proof by mathematical induction:

Proposition 8

$$\forall n. \sum_{i=0}^{n} i \;=\; \frac{n(n+1)}{2}$$

Proof. Let $P = \{n \mid \sum_{i=0}^{n} i = n(n+1)/2\}$ and we have to check that P is closed under the two rules for zero and successor.

1. For zero, we calculate
$$\sum_{i=0}^{0} i \;=\; 0 \;=\; 0.(0+1)/2$$
so that $0 \in P$.

2. For the successor rule, we assume $n \in P$ and then
$$\begin{aligned} \sum_{i=0}^{n+1} i &= \sum_{i=0}^{n} i + (n+1) \\ &= \frac{n(n+1)}{2} + (n+1) \text{ by the inductive assumption} \\ &= \frac{(n+1)((n+1)+1)}{2} \end{aligned}$$
so $(n+1) \in P$.

Then applying Proposition 7, we get that $P = \mathbb{N}$ as required. □

You should be able to see that the conditions required of P for mathematical induction to be applicable are closely related to the inductive definition of the natural numbers in terms of Z and $S()$ which we gave in Section 2.1.2. This is no accident and generalises to give an induction principle for any inductively defined set.

2.2.2 Rule induction

Proposition 9 (Rule Induction) *Let the set S be inductively defined by a set of rules $\mathcal{R}$ and $P \subseteq S$. Then if P is closed under all the rules in $\mathcal{R}$, P is the whole of S.*

Proof. By the second part of Definition 1, which says what it is for S to be inductively defined by $\mathcal{R}$, we have $S \subseteq P$. Then since we assumed $P \subseteq S$ we have $P = S$. □

Mathematical induction is just the special case of rule induction which arises when S is $\mathbb{N}$. In the case that rule induction is applied to a set of syntactic objects, where there is

one rule for each syntactic construct, rule induction is also known as *structural induction* because it becomes an induction over the syntactic structure of objects in the set.

As an example, we will consider proving some things about functions which manipulate lists in ML by structural induction – you will see more proofs like these in Dr Paulson's IB course on Foundations of Functional Programming. Lists of integers are defined by the following inductive datatype declaration[4]:

```
datatype INTLIST = Nil | Cons of int*INTLIST;
```

Given this inductive definition, we can define the `append` function inductively like this:

```
(* append : INTLIST*INTLIST -> INLIST *)
fun append(Nil,ys)        = ys
 |  append(Cons(x,xs),ys) = Cons(x,append(xs,ys));
```

Proposition 10 *The* `append` *function is associative. That is to say, for any* `xs,ys,zs`*:*

$$\texttt{append}(\texttt{xs}, \texttt{append}(\texttt{ys}, \texttt{zs})) \;=\; \texttt{append}(\texttt{append}(\texttt{xs}, \texttt{ys}), \texttt{zs})$$

Proof. We prove this by structural induction on `xs`. There are two cases:

1. If $\texttt{xs} = \texttt{Nil}$ then

$$\begin{aligned} \texttt{append}(\texttt{Nil}, \texttt{append}(\texttt{ys}, \texttt{zs})) &= \texttt{append}(\texttt{ys}, \texttt{zs}) \\ &= \texttt{append}(\texttt{append}(\texttt{Nil}, \texttt{ys}), \texttt{zs}) \end{aligned}$$

2. If $\texttt{xs} = \texttt{Cons}(\texttt{w}, \texttt{ws})$ then

$$\begin{aligned} \texttt{append}(\texttt{Cons}(\texttt{w}, \texttt{ws}), \texttt{append}(\texttt{ys}, \texttt{zs})) &= \texttt{Cons}(\texttt{w}, \texttt{append}(\texttt{ws}, \texttt{append}(\texttt{ys}, \texttt{zs}))) \\ \text{(induction)} &= \texttt{Cons}(\texttt{w}, \texttt{append}(\texttt{append}(\texttt{ws}, \texttt{ys}), \texttt{zs})) \\ &= \texttt{append}(\texttt{Cons}(\texttt{w}, \texttt{append}(\texttt{ws}, \texttt{ys})), \texttt{zs}) \\ &= \texttt{append}(\texttt{append}(\texttt{Cons}(\texttt{w}, \texttt{ws}), \texttt{ys}), \texttt{zs}) \end{aligned}$$

□

Something to watch out for when doing any kind of induction is that you will, to make the proof work, sometimes have to prove something slightly stronger than the result for which you are really aiming. Here are some more inductively defined functions to manipulate lists:

```
(* reverse : INTLIST -> INTLIST *)
fun reverse Nil = Nil
 |  reverse (Cons(x,xs)) = append(reverse xs, Cons(x,Nil));

(* revapp : INLIST*INLIST -> INTLIST *)
fun revapp (Nil,ys) = ys
 |  revapp (Cons(x,xs),ys) = revapp(xs,Cons(x,ys));

(* rev : INTLIST -> INTLIST *)
fun rev xs = revapp (xs,Nil);
```

[4]Of course, lists are already built in to the language, but we'll pretend they aren't.

and let us suppose we want to prove the following by structural induction on lists:

Proposition 11

$$\forall \texttt{xs} \in \texttt{INTLIST}.\ \texttt{rev xs} = \texttt{reverse xs}$$

One's first attempt at a proof would be to try to use structural induction on `xs` to prove the result directly. There are two syntax formation rules to consider

1. For `Nil` we observe that `reverse Nil` = `Nil` from the definition of `reverse` and that

$$\begin{aligned} \texttt{rev Nil} &= \texttt{revapp (Nil, Nil)} \\ &= \texttt{Nil} \end{aligned}$$

   so that case is OK.

2. For `Cons` we have that for any `x` and `xs`

$$\begin{aligned} \texttt{rev (Cons(x, xs))} &= \texttt{revapp (Cons(x, xs), Nil)} \\ &= \texttt{revapp (xs, Cons(x, Nil))} \end{aligned}$$

   and that

$$\begin{aligned} \texttt{reverse (Cons(x, xs))} &= \texttt{append(reverse xs, Cons(x, Nil))} \\ &= \texttt{append(rev xs, Cons(x, Nil))} \text{ by induction} \\ &= \texttt{append(revapp(xs, Nil), Cons(x, Nil))} \end{aligned}$$

   but then we're stuck. The problem is that the induction hypothesis doesn't say anything at all about `revapp` when its second argument is non-`Nil`.

So we have to prove a stronger statement which implies what we want:

Lemma 12

$$\forall \texttt{xs}.\ \forall \texttt{ys}.\ \texttt{revapp(xs, ys)} = \texttt{append(reverse xs, ys)}$$

Proof. We prove this by induction on `xs`:

1. In the case where `xs` is `Nil` we need to show

$$\forall \texttt{ys}.\ \texttt{revapp(Nil, ys)} = \texttt{append(reverse Nil, ys)}$$

   The left-hand side (LHS) is equal to `ys` by the definition of `revapp`, whilst the RHS is equal to `append(Nil,ys)` by the definition of `reverse`, and this is `ys` by the definition of `append`.

2. In the case where `xs` is `Cons(z,zs)` we reason as follows

$$\begin{aligned} &\texttt{revapp(Cons(z, zs), ys)} = \texttt{revapp(zs, Cons(z, ys))} \text{ (defn. of revapp)} \\ &\text{(induction)} = \texttt{append(reverse zs, Cons(z, ys))} \\ &\text{(defn. of append)} = \texttt{append(reverse zs, append(Cons(z, Nil), ys))} \\ &\text{(Proposition 10)} = \texttt{append(append(reverse zs, Cons(z, Nil)), ys)} \\ &\text{(defn. of reverse)} = \texttt{append(reverse(z, zs), ys)} \end{aligned}$$

□

Proposition 11 then follows immediately from Lemma 12. In a case like this it can require a certain amount of intelligence and experience (not to mention luck) to see exactly what the stronger induction hypothesis should be to make the proof go through. Indeed, finding the right hypothesis is sometimes referred to as the 'aha!' or 'eureka!' step in an inductive proof since it appears to be plucked magically out of thin air, but once you have it the rest of the proof is often fairly mechanical. A common strategy for finding induction hypotheses is to try a simple one and if the proof fails to go though, try to see *why* it fails, and use that as guidance as to how the hypothesis should be strengthened. The problem of finding induction hypotheses also shows up as the problem of finding loop invariants when proving properties of programs using Floyd-Hoare logic (see the Appendix and next year's Specification and Verification course).

2.3 Exercises

1. Given our inductive definition of $\mathbb{N}$, give an inductive definition of the usual 'less-than' relation $\leq \subseteq \mathbb{N} \times \mathbb{N}$.

2. What can you say about the set defined by an set of rules which doesn't contain any axioms?

3. Can every set be defined by an inductive definition? Given a set S, can it be that the set $\mathcal{R}$ of rules defining S is not unique? For a given set of rules $\mathcal{R}$, can two distinct sets of rule schemes denote $\mathcal{R}$?

4. Notice that we are quite happy to deal with inductive definitions which have an infinite number of rules (remember that a rule scheme is just shorthand for all its substitution instances). All our rules are, however, constrained to have a finite number of hypotheses. Think about what would happen if we were to relax this restriction. Do such definitions define anything? What happens to the downward ($\bigcap$) construction? What about the upwards ($\bigcup$) one?

5. Work out the formal details of exactly what simultaneous inductive definitions mean (Section 2.1.4). If you don't already know, find out how to make mutually recursive datatype declarations in ML and think of some practical examples.

6. Why, when defining functions from S by induction in Section 2.1.6, did we insist that every element of S had to have a unique derivation? Do elements of inductive datatypes in ML always have unique derivations?

7. The Fibonacci numbers are defined inductively by

$$F_0 = 0 \qquad F_1 = 1 \qquad F_{n+2} = F_{n+1} + F_n$$

Prove, by mathematical induction, that

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$$

where

$$\phi = \frac{(1+\sqrt{5})}{2} \qquad \text{and} \qquad \hat{\phi} = 1 - \phi.$$

8. Given a set $Prop$ of propositions, the set of *contexts* over $Prop$ is defined by

$$\frac{}{[] \in Ctxt} \qquad \frac{\Gamma \in Ctxt \qquad A \in Prop}{\Gamma, A \in Ctxt}$$

So, intuitively, a context is a finite list of propositions, separated by commas. Define a relation $\sim \subseteq Ctxt \times Ctxt$ by induction such that $\Gamma \sim \Gamma'$ just when Γ and Γ' are the same list of propositions but in a different order. (You may need to make use of some auxiliary relations.) Prove that your relation $\sim$ is an equivalence relation. (Warning: this question is fairly tricky!)

9. Assume we are given a set of basic propositions $Atom$. Let $Prop$, the set of conjunctive propositions over $Atom$, be defined by

$$\frac{A \in Atom}{A \in Prop} \qquad \frac{A \in Prop \qquad B \in Prop}{A \wedge B \in Prop}$$

Now let $Ctxt$ be the set of contexts over $Prop$ and $\sim$ be the equivalence relation on contexts as in the previous question. The entailment relation $\vdash \subseteq Ctxt \times Prop$, which we write infix, of a little logic is then defined as follows:

$$\frac{}{\Gamma, A \vdash A} \qquad \frac{\Gamma \vdash A \qquad \Gamma \sim \Gamma'}{\Gamma' \vdash A} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

Prove by induction that the following are all admissible rules (i.e. adding them does not make any difference to the set of derivable sequents):

(a)

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}$$

(b)

$$\frac{(\Gamma, A), A \vdash B}{\Gamma, A \vdash B}$$

(c)

$$\frac{\Gamma \vdash A \qquad \Gamma, A \vdash B}{\Gamma \vdash B}$$

Chapter 3

IMP and its Operational Semantics

3.1 The Syntax of IMP

Throughout this course we shall work with a toy imperative programming language which we call IMP. IMP is also sometimes called the language of *while programs*. The syntax of IMP comprises three sets (or *syntactic categories*): *Bexp* for boolean-valued expressions, *Iexp* for integer-valued expressions and *Com* for commands. These are defined inductively in terms of some auxiliary sets

$$\begin{array}{rcl} \mathbb{Z} & = & \{\ldots,-2,-1,0,1,2,3,\ldots\} \text{ the integers} \\ \mathbb{B} & = & \{\mathit{true},\mathit{false}\} \text{ the booleans} \\ \mathit{Iop} & = & \{+,-,\times,\ldots\} \text{ some finite set of integer operations} \\ \mathit{Bop} & = & \{=,>,\ldots\} \text{ some finite set of boolean operators} \\ \mathit{Pvar} & = & \{\mathtt{x},\mathtt{y},\ldots\} \text{ some infinite set of program variables} \end{array}$$

We will not worry too much about exactly what operators are built in to the language. The syntax of IMP is then defined inductively by the rules shown in Figure 3.1. We use typewriter font (like `this`) for expressions in the language and math italic (like *this*) for metavariables ranging over the various syntactic categories and auxiliary sets.

If $n \in \mathbb{Z}$ is an integer, we write $\underline{n}$ for the syntactic IMP expression corresponsing to n. So, for example, $5 \in \mathbb{Z}$ but $\underline{5} = \mathtt{5} \in \mathit{Iexp}$. Likewise, $\mathit{true} \in \mathbb{B}$ is a boolean value, whereas $\underline{\mathit{true}} = \mathtt{true} \in \mathit{Bexp}$ is the corresponding IMP phrase. A similar convention is used for the integer and boolean operations, so $\mathit{iop} \in \mathit{Iop}$ should be thought of as a proper mathematical function $\mathit{iop}\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, whereas $\underline{\mathit{iop}}$ is the syntactic *name* of that function which we use in the programming language. For example, $\times \in \mathit{Iop}$ is the multiplication function, but $\underline{\times} = *$ is the textual symbol we use to indicate multiplication in IMP. Whilst this might appear abstruse, it is not mere pedantry – these distinctions between syntax ($\mathtt{5} \in \mathit{Iexp}$) and semantics ($5 \in \mathbb{Z}$) are absolutely central to this course.[1]

[1] This is also discussed in Dr Forster's Part II Philosophy lectures as the 'use-mention distinction' which occurs in natural language. When we write a word (for simplicity, a noun) then we are normally *using* it – we expect the reader mentally to dereference the marks on the page to obtain (the idea of) the real-world object which they denote. By using quotation marks, we can *mention* the word, referring to the syntactic object. The following two sentences illustrate the idea:

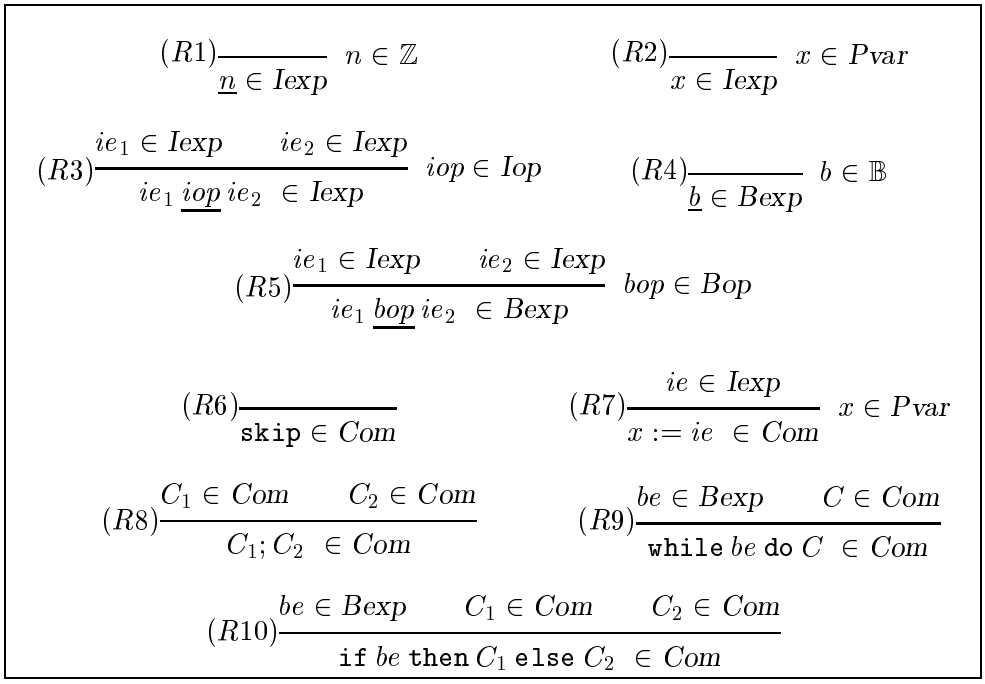


Figure 3.1: The Syntax of IMP

The syntax of IMP is simple enough that you should be able to guess (informally) how programs are supposed to behave. (We will shortly see how to formalise that behaviour.) For example, the following program computes the factorial of 5, leaving the result in the variable `r`:

```
x := 5;(r := 1; (while x>1 do (r := r*x; x := x-1)))
```

We can also express the syntax of IMP as ML datatypes (using the builtin type `string` to represent *Pvar*, `int` to represent $\mathbb{Z}$ and `bool` for $\mathbb{B}$):

```
datatype IOP = Plus | Times | Minus;
datatype IEXP = N of int | Pvar of string | Iop of IOP*IEXP*IEXP;

datatype BOP = Equal | Greater;
datatype BEXP = B of bool | Bop of BOP*IEXP*IEXP;

datatype COM = Skip | Assign of string*IEXP | Seq of COM*COM |
               If of BEXP*COM*COM | While of BEXP*COM;
```

Notice that the distinction between integers and IMP numerals which we made such a

---

Philosophers are all very rich.
"Philosophers" is a word with twelve letters.

This kind of distinction is important in many disciplines. The (often rather complex) interplay of signifier and signified, sign and referent, is at the heart of much work in philosophy, metamathematics, linguistics and even sociology. Whilst this is clearly all well outside the scope of these lectures, you should at least be aware that it is A Very Important Idea.

fuss about earlier shows up quite clearly in the ML code, with the place of the underline operation taken by the constructor `N()`. Thus `5:int`, but `N(5):IEXP`.[2]

3.2 Transition Semantics of IMP

In this section we give IMP an operational semantics using a transition relation which expresses how a command or expression successively rewrites, or evolves, to another. This is similar to the β-reduction relation for the λ-calculus (IB Foundations of Functional Programming) or the labelled transitions used to define the dynamic behaviour of CCS agents or Pi Calculus processes (Part II Concurrency Theory and the Pi Calculus). One difference is that how an IMP phrase behaves depends not just on the phrase itself, but also on the values currently held in each of the program variables. Similarly, the behaviour of a command consists not just of rewriting to a new phrase, but may also involve changes to some of the variables.

3.2.1 States

We will refer to an assignment of an integer value to each program variable as a *state*. Formally, we define the set of all states by

$$States \stackrel{\text{def}}{=} Pvar \to \mathbb{Z}$$

so a state is a function from variable names to integers. If $S \in States$, $x \in Pvar$ and $S(x) = n$ then n is the integer stored in variable x in state S.

If $S \in States$,$x \in Pvar$ and $n \in \mathbb{Z}$ then we write $S[n/x]$ for the state S with x updated to n. In symbols

$$(S[n/x])(y) \stackrel{\text{def}}{=} \begin{cases} n & \text{if } y = x \\ S(y) & \text{otherwise} \end{cases}$$

for all $y \in Pvar$.

We can code states in ML in several ways. The one which mimics the mathematical treatment most closely uses ML functions:

```
type STATES = string -> int;

(* update : STATES*string*int -> STATES *)
fun update (S,x,n) = fn y => if y=x then n else S(y);
```

but this has the slight disadvantage (for interactive experimentation) that states are then not printable values. For this reason alone, we will instead let states be (finite) *partial* functions from strings to integers. These can then be represented by association lists (which can be printed):

[2]The observant and picky reader will also notice that the analogous distinction for integer and boolean operations still exists in the ML code, but has been reversed by comparison with the mathematical treatment. In the ML, an term of type `IOP` is the *name* of an operation which will be mapped to the operation itself by a function `iopmeaning` which we will give later on. This contrasts with the mathematics, where an element of *Iop* is the actual operation, which can be mapped to its name by applying the underline function. There's no significant difference – it's just a matter of what is taken as basic and what is derived.

```
type STATES = (string*int) list;

(* lookup : string*STATES -> int *)
exception Lookup;
fun lookup(x,[]) = raise Lookup
 |  lookup(x,(y,v)::pairs) = if x=y then v else lookup(x,pairs);

(* update : STATES*string*int -> STATES *)
fun update (S,x,n) = case S of
     [] => [(x,n)]
  | ((y,v)::pairs) => if x=y then (x,n)::pairs
                             else (y,v)::(update (pairs,x,n));
```

3.2.2 Operational semantics via transition relations

We now inductively define three relations:

$$\begin{aligned} \rightarrow_I &\subseteq (Iexp \times States) \times (Iexp \times States) \\ \rightarrow_B &\subseteq (Bexp \times States) \times (Bexp \times States) \\ \rightarrow_C &\subseteq (Com \times States) \times (Com \times States) \end{aligned}$$

by the rules shown in Figure 3.2 where we write, for example,

$$\langle C, S\rangle \rightarrow_C \langle C', S'\rangle$$

instead of

$$((C, S), (C', S')) \in \rightarrow_C$$

which should be read as 'in state S the command C can make a one-step transition to the command C' and new state S'' (and similarly for integer and boolean expressions). We will sometimes simply write $\rightarrow$ for any of $\rightarrow_I, \rightarrow_B, \rightarrow_C$, since which relation is meant is usually clear from context. We will call a pair $\langle e, S\rangle$ of a phrase (an expression or a command) and a state a *configuration*.

Notes on Figure 3.2:

1. We have left out some fairly obvious side-conditions for reasons of space. For example, rule $(\rightarrow_I \cdot 1)$ has the side condition that $x \in Pvar$.

2. In rule $(\rightarrow_I \cdot 4)$, $\underline{n_1\ iop\ n_2}$ denotes $\underline{c}$ where $c \in \mathbb{Z}$ is the result of applying the actual mathematical operation $iop\colon \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ to the integers n_1 and n_2. For example, one of the instances of this rule is $\langle \mathtt{5} + \mathtt{3}, S\rangle \rightarrow_I \langle \mathtt{8}, S\rangle$ (for any S).

3. Similarly, in rule $(\rightarrow_B \cdot 3)$, $\underline{n_1\ bop\ n_2}$ stands for whichever of `true` or `false` corresponds to the value of the function $bop\colon \mathbb{Z} \times \mathbb{Z} \rightarrow \{true, false\}$ when it is applied to the two integers n_1 and n_2.

4. The rules divide into two classes. Those with no hypotheses are the ones which do real computational work, whilst the others are there to show exactly how a transition on a subphrase causes a transition on the larger phrase of which it is a part. For example, $(\rightarrow_I \cdot 1)$ and $(\rightarrow_I \cdot 4)$ make real progress, whereas the other two integer expression rules do not.

$$(\rightarrow_I \cdot 1)\frac{}{\langle x, S\rangle \rightarrow_I \langle \underline{S(x)}, S\rangle} \qquad (\rightarrow_I \cdot 2)\frac{\langle ie_1, S\rangle \rightarrow_I \langle ie_1', S'\rangle}{\langle ie_1 \ \underline{iop}\ ie_2, S\rangle \rightarrow_I \langle ie_1' \ \underline{iop}\ ie_2, S'\rangle}$$

$$(\rightarrow_I \cdot 3)\frac{\langle ie, S\rangle \rightarrow_I \langle ie', S'\rangle}{\langle \underline{n}\ \underline{iop}\ ie, S\rangle \rightarrow_I \langle \underline{n}\ \underline{iop}\ ie', S'\rangle} \qquad (\rightarrow_I \cdot 4)\frac{}{\langle \underline{n_1}\ \underline{iop}\ \underline{n_2}, S\rangle \rightarrow_I \langle \underline{n_1\ iop\ n_2}, S\rangle}$$

$$(\rightarrow_B \cdot 1)\frac{\langle ie_1, S\rangle \rightarrow_I \langle ie_1', S'\rangle}{\langle ie_1 \ \underline{bop}\ ie_2, S\rangle \rightarrow_B \langle ie_1' \ \underline{bop}\ ie_2, S'\rangle} \qquad (\rightarrow_B \cdot 2)\frac{\langle ie, S\rangle \rightarrow_I \langle ie', S'\rangle}{\langle \underline{n}\ \underline{bop}\ ie, S\rangle \rightarrow_B \langle \underline{n}\ \underline{bop}\ ie', S'\rangle}$$

$$(\rightarrow_B \cdot 3)\frac{}{\langle \underline{n_1}\ \underline{bop}\ \underline{n_2}, S\rangle \rightarrow_I \langle \underline{n_1\ bop\ n_2}, S\rangle}$$

$$(\rightarrow_C \cdot 1)\frac{\langle ie, S\rangle \rightarrow_I \langle ie', S'\rangle}{\langle x{:=}ie, S\rangle \rightarrow_C \langle x{:=}ie', S'\rangle} \qquad (\rightarrow_C \cdot 2)\frac{}{\langle x{:=}\underline{n}, S\rangle \rightarrow_C \langle \texttt{skip}, S[n/x]\rangle}$$

$$(\rightarrow_C \cdot 3)\frac{\langle C_1, S\rangle \rightarrow_C \langle C_1', S'\rangle}{\langle C_1\ ;\ C_2, S\rangle \rightarrow_C \langle C_1'\ ;\ C_2, S'\rangle} \qquad (\rightarrow_C \cdot 4)\frac{}{\langle \texttt{skip};C, S\rangle \rightarrow_C \langle C, S\rangle}$$

$$(\rightarrow_C \cdot 5)\frac{\langle be, S\rangle \rightarrow_B \langle be', S'\rangle}{\langle \texttt{if}\ be\ \texttt{then}\ C_1\ \texttt{else}\ C_2, S\rangle \rightarrow_C \langle \texttt{if}\ be'\ \texttt{then}\ C_1\ \texttt{else}\ C_2, S'\rangle}$$

$$(\rightarrow_C \cdot 6)\frac{}{\langle \texttt{if true then}\ C_1\ \texttt{else}\ C_2, S\rangle \rightarrow_C \langle C_1, S\rangle}$$

$$(\rightarrow_C \cdot 7)\frac{}{\langle \texttt{if false then}\ C_1\ \texttt{else}\ C_2, S\rangle \rightarrow_C \langle C_2, S\rangle}$$

$$(\rightarrow_C \cdot 8)\frac{}{\langle \texttt{while}\ be\ \texttt{do}\ C, S\rangle \rightarrow_C \langle \texttt{if}\ be\ \texttt{then}\ (C\ ;\ \texttt{while}\ be\ \texttt{do}\ C)\ \texttt{else skip}, S\rangle}$$

Figure 3.2: One-Step Transition Semantics of IMP

5. Following on from the last point, the rules $(\rightarrow_I \cdot 2)$ and $(\rightarrow_I \cdot 3)$ specify the evaluation order for integer expressions as being strictly left-to-right – the first operand must be a numeral before any transitions on the second operand can occur. For some applications, this might be regarded as *overspecification* (see the Exercises).

Here is an example of a simple derivation of an instance of the transition relation, where we assume that the state S is such that $S(\mathtt{y}) = 3$:

$$\dfrac{\dfrac{}{\langle \mathtt{y}, S\rangle \rightarrow_I \langle \mathtt{3}, S\rangle}\ (\rightarrow_I \cdot 1)}{\langle \mathtt{y} := \mathtt{y} + (\mathtt{3} + \mathtt{4}), S\rangle \rightarrow_C \langle \mathtt{y} := \mathtt{3} + (\mathtt{3} + \mathtt{4}), S\rangle}\ (\rightarrow_C \cdot 1)$$

You should similarly be able verify each of the following subsequent steps in the execution of this command:

$$\langle \mathtt{y} := \mathtt{3} + (\mathtt{3} + \mathtt{4}), S\rangle \ \rightarrow_C\ \langle \mathtt{y} := \mathtt{3} + \mathtt{7}, S\rangle \ \rightarrow_C\ \langle \mathtt{y} := \mathtt{10}, S\rangle \ \rightarrow_C\ \langle \mathtt{skip}, S[10/\mathtt{y}]\rangle$$

As a more involved example, assume that $S \in \mathit{States}$ satisfies $S(\mathtt{x}) = 2$ and $S(\mathtt{r}) = 60$ and let $C = (\mathtt{while}\ \mathtt{x} > \mathtt{1}\ \mathtt{do}\ C_1)$ where $C_1 = (\mathtt{r} := \mathtt{r} * \mathtt{x}\,;\, \mathtt{x} := \mathtt{x} - \mathtt{1})$. Each of the following transitions can be justified by a short proof using the rules of the transition semantics (rather tedious Exercise).

$$\begin{aligned}
\langle C, S\rangle &\rightarrow_C \langle \mathtt{if}\ \mathtt{x} > \mathtt{1}\ \mathtt{then}\ (C_1\,;\, C)\ \mathtt{else}\ \mathtt{skip}, S\rangle \\
&\rightarrow_C \langle \mathtt{if}\ \mathtt{2} > \mathtt{1}\ \mathtt{then}\ (C_1\,;\, C)\ \mathtt{else}\ \mathtt{skip}, S\rangle \\
&\rightarrow_C \langle \mathtt{if}\ \mathtt{true}\ \mathtt{then}\ (C_1\,;\, C)\ \mathtt{else}\ \mathtt{skip}, S\rangle \\
&\rightarrow_C \langle C_1\,;\, C, S\rangle \\
&\rightarrow_C \langle (\mathtt{r} := \mathtt{60} * \mathtt{x}\,;\, \mathtt{x} := \mathtt{x} - \mathtt{1})\,;\, C, S\rangle \\
&\rightarrow_C \langle (\mathtt{r} := \mathtt{60} * \mathtt{2}\,;\, \mathtt{x} := \mathtt{x} - \mathtt{1})\,;\, C, S\rangle \\
&\rightarrow_C \langle (\mathtt{r} := \mathtt{120}\,;\, \mathtt{x} := \mathtt{x} - \mathtt{1})\,;\, C, S\rangle \\
&\rightarrow_C \langle (\mathtt{skip}\,;\, \mathtt{x} := \mathtt{x} - \mathtt{1})\,;\, C, S[120/\mathtt{r}]\rangle \\
&\rightarrow_C \langle \mathtt{x} := \mathtt{x} - \mathtt{1}\,;\, C, S[120/\mathtt{r}]\rangle \\
&\rightarrow_C \langle \mathtt{x} := \mathtt{2} - \mathtt{1}\,;\, C, S[120/\mathtt{r}]\rangle \\
&\rightarrow_C \langle \mathtt{x} := \mathtt{1}\,;\, C, S[120/\mathtt{r}]\rangle \\
&\rightarrow_C \langle \mathtt{skip}\,;\, C, S[120/\mathtt{r}][1/\mathtt{x}]\rangle \\
&\rightarrow_C \langle C, S[120/\mathtt{r}][1/\mathtt{x}]\rangle \\
&\rightarrow_C \langle \mathtt{if}\ \mathtt{x} > \mathtt{1}\ \mathtt{then}\ (C_1\,;\, C)\ \mathtt{else}\ \mathtt{skip}, S[120/\mathtt{r}][1/\mathtt{x}]\rangle \\
&\rightarrow_C \langle \mathtt{if}\ \mathtt{1} > \mathtt{1}\ \mathtt{then}\ (C_1\,;\, C)\ \mathtt{else}\ \mathtt{skip}, S[120/\mathtt{r}][1/\mathtt{x}]\rangle \\
&\rightarrow_C \langle \mathtt{if}\ \mathtt{false}\ \mathtt{then}\ (C_1\,;\, C)\ \mathtt{else}\ \mathtt{skip}, S[120/\mathtt{r}][1/\mathtt{x}]\rangle \\
&\rightarrow_C \langle \mathtt{skip}, S[120/\mathtt{r}][1/\mathtt{x}]\rangle
\end{aligned}$$

3.2.3 Theorems about the transition semantics

The transition semantics is good for more than just specifying what the result of a particular program should be. We can also use it to prove statements about programs in general. Here's one simple example:

Proposition 13 (IMP expressions have no side-effects) *If either*

$$\langle ie, S\rangle \rightarrow_I \langle ie', S'\rangle \text{ or } \langle be, S\rangle \rightarrow_B \langle be', S'\rangle$$

then $S = S'$. In other words, evaluation of integer and boolean expressions has no effect on the state.

Proof. This follows by rule induction, first on the definition of $\rightarrow_I$ and then on that of $\rightarrow_B$, and is left as an exercise. □

Commands, on the other hand, *can* change the state (rule $(\rightarrow_C \cdot 2)$). Proposition 13 means that we could have given an alternative set of definitions for the transition semantics in which the relations $\rightarrow_I$ and $\rightarrow_B$ were given as subsets of $(Iexp \times States) \times Iexp$ and $(Bexp \times States) \times Bexp$ respectively.

Theorem 14 (Transitions are deterministic) *For any phrases (expressions or commands) e,e',e'' and any states S,S',S'', if*

$$\langle e, S\rangle \rightarrow \langle e', S'\rangle \text{ and } \langle e, S\rangle \rightarrow \langle e'', S''\rangle$$

then $e' = e''$ and $S' = S''$.

Proof. This follows by structural induction on e. In any proof of $\langle e, S\rangle \rightarrow \langle e', S'\rangle$, the last rule used is uniquely determined by the structure of e. For example, suppose that $e = (e_1 \; \underline{iop} \; e_2)$ and that the result holds for e_1 and e_2. Then there are three cases to consider:

- If $e_1 = \underline{n_1}$ and $e_2 = \underline{n_2}$ are both constants then the last rule used in a proof of $\langle e, S\rangle \rightarrow_I \langle e', S'\rangle$ or of $\langle e, S\rangle \rightarrow_I \langle e'', S''\rangle$ must be $(\rightarrow_I \cdot 4)$ and hence $e' = \underline{e_1 \; iop \; e_2} = e''$ and $S' = S = S''$.

- If $e_1 = \underline{n_1}$ is a constant but e_2 is not then the last rule used in any proof of $\langle e, S\rangle \rightarrow$ must be $(\rightarrow_I \cdot 3)$ so that the two proofs must look like

$$(\rightarrow_I \cdot 3)\frac{\langle e_2, S\rangle \rightarrow_I \langle e_2', S'\rangle}{\langle \underline{n_1} \; \underline{iop} \; e_2, S\rangle \rightarrow_I \langle \underline{n_1} \; \underline{iop} \; e_2', S'\rangle} \qquad (\rightarrow_I \cdot 3)\frac{\langle e_2, S\rangle \rightarrow_I \langle e_2'', S''\rangle}{\langle \underline{n_1} \; \underline{iop} \; e_2, S\rangle \rightarrow_I \langle \underline{n_1} \; \underline{iop} \; e_2'', S''\rangle}$$

  Then by the induction hypothesis applied to e_2, we must have $e_2' = e_2''$ and $S' = S''$, and hence

$$e' \;=\; (\underline{n_1} \; \underline{iop} \; e_2') \;=\; (\underline{n_1} \; \underline{iop} \; e_2'') \;=\; e''$$

  as required.

- e_1 is not a constant. Then the last rule used must have been $(\rightarrow_I \cdot 2)$ and we reason much as in the previous case that $e' = e''$ and $S' = S''$.

Each of the other cases for the structure of e can be dealt with in a similar manner, and we leave them as Exercises. □

3.2.4 Evaluation sequences

A configuration $\langle e, S\rangle$ is said to be *terminal* if there is no $\langle e', S'\rangle$ such that $\langle e, S\rangle \rightarrow \langle e', S'\rangle$. A moment's inspection of the transition rules shows that the terminal configurations are precisely

$$\langle \underline{n}, S\rangle \qquad \langle \texttt{true}, S\rangle \qquad \langle \texttt{false}, S\rangle \qquad \langle \texttt{skip}, S\rangle$$

An *infinite evaluation sequence* for $\langle e, S\rangle$ is an infinite chain of one-step transitions:

$$\langle e, S\rangle = \langle e_0, S_0\rangle \;\rightarrow\; \langle e_1, S_1\rangle \;\rightarrow\; \langle e_2, S_2\rangle \;\rightarrow\; \cdots$$

where for all i, $\langle e_i, S_i\rangle$ is not terminal.

A *finite evaluation sequence* for $\langle e, S\rangle$ is finite chain

$$\langle e, S\rangle = \langle e_0, S_0\rangle \;\to\; \langle e_1, S_1\rangle \;\to\; \langle e_2, S_2\rangle \;\to\; \cdots \;\to\; \langle e_n, S_n\rangle$$

with $\langle e_n, S_n\rangle$ terminal. Evaluation sequences are also called *traces* (which roughly matches the way in which the word 'tracing' is used in the context of debugging to refer to examining the sequence of indermediate states during a particular run of a program).

By Theorem 14, each $\langle e, S\rangle$ has a *unique* evaluation sequence which is either infinite or else terminates with a terminal configuration $\langle e_n, S_n\rangle$ which is uniquely determined by $\langle e, S\rangle$. In fact we can be a bit more precise:

Lemma 15 (Expressions always terminate) *If $e \in Iexp \cup Bexp$ then for any $S \in States$, $\langle e, S\rangle$ has a finite evaluation sequence.*

Proof. Structural induction (Exercise). □

The previous lemma, together with Proposition 13, means that we can define *evaluation functions* for expressions

$$\begin{array}{rcl} Ieval & : & Iexp \to (States \to \mathbb{Z}) \\ Beval & : & Bexp \to (States \to \mathbb{B}) \end{array}$$

by

$$\begin{array}{rcl} Ieval(ie)(S) & = & \text{the unique } n \in \mathbb{Z} \text{ st. } \langle ie, S\rangle \to_I^* \langle \underline{n}, S\rangle. \\ Beval(be)(S) & = & \text{the unique } b \in \mathbb{B} \text{ st. } \langle be, S\rangle \to_B^* \langle \underline{b}, S\rangle. \end{array}$$

(Recall that $\to^*$ is the reflexive transitive closure of $\to$, defined by the following inductive rules:

$$\frac{}{x \to^* x} \qquad\qquad \frac{x \to^* y \qquad y \to z}{x \to^* z}$$

)

In contrast to the situation for expressions, commands can have infinite evaluation sequences. For example, if $C =$ `while true do skip` then

$$\begin{array}{rcl} \langle C, S\rangle & \to_C & \langle \texttt{if true then (skip ; } C\texttt{) else skip}, S\rangle \\ & \to_C & \langle \texttt{skip ; } C, S\rangle \\ & \to_C & \langle C, S\rangle \\ & \to_C & \cdots \text{ and so on for ever} \end{array}$$

However, if $\langle C, S\rangle$ *does* have a finite evaluation sequence, say

$$\langle C, S\rangle \;\to_C^*\; \langle \texttt{skip}, S'\rangle$$

then by Theorem 14 we know that S' is uniquely determined by C and S, so that C determines a *partial function* from states to states:

$$\begin{array}{rcl} Ceval & : & Com \to (States \rightharpoonup States) \\ Ceval(C)(S) & = & \begin{cases} \text{the unique } S' \text{ st. } \langle C, S\rangle \to_C^* \langle \texttt{skip}, S'\rangle \text{ if it exists} \\ \text{undefined, otherwise} \end{cases} \end{array}$$

For example, if C is the factorial program

$$C = \texttt{r} := 1;\ (\texttt{while x} > 1\ \texttt{do}\ (\texttt{r} := \texttt{r} * \texttt{x};\ \texttt{x} := \texttt{x} - 1))$$

then *Ceval*(C) is the (total) function *States* $\to$ *States* given by

$$S \mapsto \begin{cases} S[n!/\texttt{r}][1/\texttt{x}] & \text{if } n > 1 \\ S[1/\texttt{r}] & \text{if } n \leq 1 \end{cases}$$

where $n = S(\texttt{x})$.

We should remark at this point that although IMP is a long way from being a practical programming language, it *is* Turing powerful. This means that for any partial recursive function $f : \mathbb{Z} \rightharpoonup \mathbb{Z}$, there is an IMP command C which computes f in the sense that for all states S, *Ceval*$(C)(S)$ is defined iff $f(S(\texttt{x}))$ is defined and in that case *Ceval*$(C)(S)(\texttt{y}) = f(S(\texttt{x}))$. (See the Exercises at the end of the Chapter.)

3.2.5 Implementing the transition semantics in ML

To implement the transition semantics in ML, we rely on some of the mathematical results which we have just proved. In particular, the one-step transition relations are all actually partial functions because every non-terminal configuration $\langle e, S\rangle$ has a unique successor. We simply code these partial functions as ML functions `istep`, `bstep` and `cstep` (for integer expressions, boolean expressions and commands, respectively) in a way which directly expresses the rules in Figure 3.2:

```
(* iopmeaning : IOP -> ((int*int)->int) *)
fun iopmeaning iop (x:int,y:int) = case iop of
  Plus => x+y
| Times => x*y
| Minus => x-y;

(* bopmeaning : BOP -> ((int*int)->bool) *)
fun bopmeaning bop (x:int,y:int) = case bop of
  Equal => x=y
| Greater => x>y;

(* istep : IEXP*STATES -> IEXP *)
fun istep(ie,S:STATES) = case ie of
  Pvar(name) => N(lookup(name,S))
| Iop(iop,N(n1),N(n2)) => N(iopmeaning iop (n1,n2))
| Iop(iop,N(n1),ie2) => let val ie2' = istep(ie2,S)
                        in Iop(iop,N(n1),ie2')
                        end
| Iop(iop,ie1,ie2) => let val ie1' = istep(ie1,S)
                      in Iop(iop,ie1',ie2)
                      end;

(* bstep : BEXP*STATES -> BEXP *)
fun bstep (be,S:STATES) = case be of
  Bop(bop,N(n1),N(n2)) => B(bopmeaning bop (n1,n2))
```

```
| Bop(bop,N(n1),ie2) => let val ie2' = istep(ie2,S)
                        in Bop(bop,N(n1),ie2')
                        end
| Bop(bop,ie1,ie2) => let val ie1' = istep(ie1,S)
                      in Bop(bop,ie1',ie2)
                      end;

(* cstep : COM*STATES -> COM*STATES *)
fun cstep (com,S:STATES) = case com of
  Assign(name,N(n)) => (Skip,update(S,name,n))
| Assign(name,ie) => let val ie' = istep(ie,S)
                     in (Assign(name,ie'), S)
                     end
| Seq(Skip,C) => (C,S)
| Seq(C1,C2) => let val (C1',S') = cstep(C1,S)
                in (Seq(C1',C2),S')
                end
| If(B(true),C1,C2) => (C1,S)
| If(B(false),C1,C2) => (C2,S)
| If(be,C1,C2) => let val be' = bstep(be,S)
                  in (If(be',C1,C2), S)
                  end
| While(be,C) => (If(be,Seq(C,While(be,C)),Skip), S);
```

You should be able to see that each clause of the definition of (say) `cstep` corresponds to exactly one of the transition rules, though we have to use some intelligence in ordering the clauses.[3] Once we've got the one-step transitions, defining the ML versions of the functions *Ieval*,*Beval* and *Ceval* is straightforward, as we just keep applying the successor operation until we reach a terminal configuration:

```
(* ieval : IEXP -> (STATES -> int) *)
fun ieval (N(n)) (S:STATES) = n
|   ieval ie S = let val ie' = istep(ie,S)
                 in ieval ie' S
                 end;

(* beval : BEXP -> (STATES -> bool) *)
fun beval (B(b)) (S:STATES) = b
|   beval be S = let val be' = bstep (be,S)
                 in beval be' S
                 end;

(* ceval : COM -> (STATES -> STATES) *)
fun ceval Skip (S:STATES) = S
|   ceval C S = let val (C',S') = cstep (C,S)
                in ceval C' S'
```

[3] There are no clauses for terminal configurations, just as there are no transition rules for them in the semantics – attempting to compute the successor of such a configuration will simply raise an uncaught `match` exception.

```
                    end;
```

Note that the evaluation functions are tail-recursive, so that iterative IMP programs will execute in constant ML stack space. Here's an example of using the ML code to execute an IMP program:

```
-  (* initial state - everything is undefined *)
=  val (S:STATES) = [];
>  val S = [] : STATES

-  (* example factorial calculation *)
=  val factprog = Seq(Assign("x",N(5)),Seq(Assign("r",N(1)),
=           While(Bop(Greater,Pvar("x"),N(1)),
=             Seq(Assign("r",Iop(Times,Pvar("r"),Pvar("x"))),
=              Assign("x",Iop(Minus,Pvar("x"),N(1)))))));

>  val factprog = Seq(Assign ("x", N 5), ... ) : COM
-  ceval factprog S;
>  [("x",1),("r",120)] : STATES
```

In fact, the ML code which accompanies this course includes simple parsers and pretty-printers for IMP programs (based on code for Dr Paulson's book "ML for the Working Programmer"). This means you don't have to type programs in the extremely messy form used above, but can instead do this:[4]

```
-  val fibprog  = readcom "\
=\ last := 0; next := 1; n := 8;\
=\ while n>0 do\
=\   next := last+next;\
=\   last := next-last;\
=\   n := n-1\
=\ endwhile";
> val fibprog = Seq(Assign("last",N 0), ... : COM
- ceval fibprog S;
> [("last",21),("next",34),("n",0)] :  STATES
```

There are also functions to parse expressions (`readiexp` and `readbexp`) and to print phrases (`prcom`, `priexp` and `prbexp`).

Attempting to compute `ceval C S` in the case that $\langle C, S\rangle$ has an infinite evaluation sequence (that is, in the case that $Ceval(C)(S)$ is undefined) will cause ML to fail to terminate. Because of the undecidability of the halting problem, there is in general no way to predict when this will happen.

[4]The concrete syntax of IMP which the parser implements includes mandatory `endif` and `endwhile` keywords which are used in the obvious way. The default behaviour of sequential composition (i.e. ;) is to associate to the right, as is that of arithmetic operations (which also have the normal precedences). For both commands and arithmetic operators, parentheses may be used to override the default groupings. Whether or not you have to type the rather unpleasant \ continuation characters to break string literals over more than one line depends on what version of ML you use.

3.3 Structural Evaluation Relations for IMP

3.3.1 Evaluation relations

The transition semantics of the previous section allowed us to define the evaluation relations *Ieval*,*Beval* and *Ceval* in terms of the reflexive transitive closures of the one-step transition relations $\rightarrow_I$,$\rightarrow_B$ and $\rightarrow_C$. In this section we shall show that these relations can be described *directly* by a set of rules which follow the syntactic structure of IMP phrases. This kind of operational semantics, which is sometimes called 'natural semantics', is often more convenient to work with than the transition semantics.

We will define three *evaluation relations*

$$\begin{array}{rcl} \Rightarrow_I & \subseteq & \mathit{Iexp} \times \mathit{States} \times \mathbb{Z} \\ \Rightarrow_B & \subseteq & \mathit{Bexp} \times \mathit{States} \times \mathbb{B} \\ \Rightarrow_C & \subseteq & \mathit{Com} \times \mathit{States} \times \mathit{States} \end{array}$$

and we will write

$$\begin{array}{rcl} ie, S \Rightarrow_I n & \text{instead of} & (ie, S, n) \in \Rightarrow_I \\ be, S \Rightarrow_B b & \text{instead of} & (be, S, b) \in \Rightarrow_B \\ C, S \Rightarrow_C S' & \text{instead of} & (C, S, S') \in \Rightarrow_C \end{array}$$

The evaluation relations are defined by the inductive rules shown in Figure 3.3, where once again we have left out some obvious side-conditions.

Here is the same simple example as we gave on page 25, but done using the evaluation, rather than the transition, semantics. Assume that S is such that $S(\mathtt{y}) = 3$, then:

$$\dfrac{\dfrac{}{\mathtt{y}, S \Rightarrow_I 3}\,(\Rightarrow_I\cdot 2) \qquad \dfrac{\dfrac{}{\mathtt{3}, S \Rightarrow_I 3}\,(\Rightarrow_I\cdot 1) \qquad \dfrac{}{\mathtt{4}, S \Rightarrow_I 4}\,(\Rightarrow_I\cdot 1)}{\mathtt{3}+\mathtt{4}, S \Rightarrow_I 7}\,(\Rightarrow_I\cdot 3)}{\mathtt{y}+(\mathtt{3}+\mathtt{4}), S \Rightarrow_I 10}\,(\Rightarrow_I\cdot 3)$$
$$\dfrac{\mathtt{y}+(\mathtt{3}+\mathtt{4}), S \Rightarrow_I 10}{\mathtt{y} := \mathtt{y}+(\mathtt{3}+\mathtt{4}), S \Rightarrow_C S[10/\mathtt{y}]}\,(\Rightarrow_C\cdot 2)$$

Note that there is just one derivation for the entire evaluation of the command. This is in contrast to the situation for the transition semantics, where every individual transition is justified by its own derivation.

Exercise: Assume that $S \in \mathit{States}$ satisfies $S(\mathtt{x}) = 2$ and $S(\mathtt{r}) = 60$ and let $C = (\mathtt{while\ x} > \mathtt{1\ do}\ C_1)$ where $C_1 = (\mathtt{r} := \mathtt{r} * \mathtt{x}\,;\, \mathtt{x} := \mathtt{x} - \mathtt{1})$. Produce a derivation like that above which proves

$$C, S \Rightarrow_C S[120/\mathtt{r}][1/\mathtt{x}]$$

The evaluation semantics is much less 'fine-grained' than the transition semantics and this style is sometimes called *big step* operational semantics, by contrast with the *small step* style of the transition semantics. Certain low-level features which are made explicit in the small-step semantics are thus hidden in the big-step semantics. The most obvious is that, as we remarked on page 25, the transition semantics specifies that the evaluation of integer expressions proceeds in a strict left-to-right order. This is not the case for the evaluation semantics, since rule $(\Rightarrow_I\cdot 3)$ simply amounts to saying 'to evaluate ie_1 *iop* ie_2,

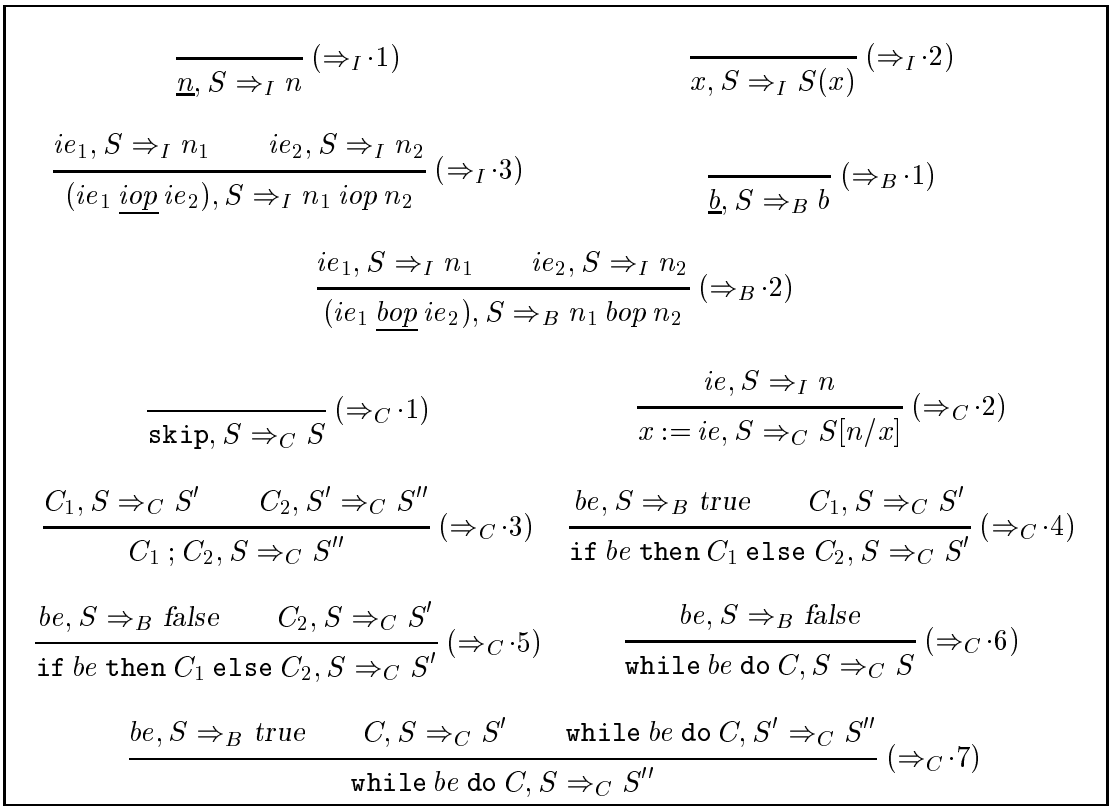


Figure 3.3: Evaluation Semantics of IMP

evaluate ie_1 and ie_2 and combine the results with iop'. In general, how much difference this makes will depend on the fine details of the language; whether we are interested in the extra low-level details provided by the transition semantics will depend on what we are using the semantics for.

3.3.2 Equivalence of transition and evaluation semantics of IMP

Now we have two different operational semantics for IMP, the obvious question to ask (particularly in view of the remarks at the end of the last section) is whether or not they agree. In this section we shall prove that they do.

Theorem 16 *For all $ie \in Iexp$, $be \in Bexp$,$C \in Com$, $S, S' \in States$, $n \in \mathbb{Z}$ and $b \in \mathbb{B}$,*

$$\begin{array}{rll} \langle ie, S\rangle \to_I^* \langle \underline{n}, S\rangle & \textit{if and only if} & ie, S \Rightarrow_I n \\ \langle be, S\rangle \to_B^* \langle \underline{b}, S\rangle & \textit{if and only if} & be, S \Rightarrow_B b \\ \langle C, S\rangle \to_C^* \langle \mathtt{skip}, S'\rangle & \textit{if and only if} & C, S \Rightarrow_C S' \end{array}$$

Proof. Firstly note that for each of the three clauses of the theorem, we have to prove both a left-to-right and a right-to-left implication. The broad structure of the proof is as follows:

1. Prove the right-to-left implications by rule induction for $\Rightarrow$.
2. Use rule induction for $\to$ to show that

$$\begin{array}{rll} \langle ie, S\rangle \to_I \langle ie', S\rangle \text{ and } ie', S \Rightarrow_I n & \text{implies} & ie, S \Rightarrow_I n \\ \langle be, S\rangle \to_B \langle be', S\rangle \text{ and } be', S \Rightarrow_B b & \text{implies} & be, S \Rightarrow_B b \\ \langle C, S\rangle \to_C \langle C', S'\rangle \text{ and } C', S' \Rightarrow_C S'' & \text{implies} & C, S \Rightarrow_C S'' \end{array}$$

3. Deduce the left-to-right implications from 2.

Proof of 1. Since $\Rightarrow_I$,$\Rightarrow_B$ and $\Rightarrow_C$ are inductively defined by the rules shown in Figure 3.3, it suffices to show that the subsets

$$\begin{array}{rcl} \{(ie, S, n) \mid \langle ie, S\rangle \to_I^* \langle \underline{n}, S\rangle\} & \subseteq & Iexp \times States \times \mathbb{Z} \\ \{(be, S, b) \mid \langle be, S\rangle \to_B^* \langle \underline{b}, S\rangle\} & \subseteq & Bexp \times States \times \mathbb{B} \\ \{(C, S, S') \mid \langle C, S\rangle \to_C^* \langle \mathtt{skip}, S'\rangle\} & \subseteq & Com \times States \times States \end{array}$$

are closed under all these rules. We will just check the case of rule $(\Rightarrow_C \cdot 7)$ (since it is the most interesting) and leave the remaining cases as Exercises.

So, suppose that the hypotheses of $(\Rightarrow_C \cdot 7)$ are in the sets. I.e. we assume the following three things:

(a) $\langle be, S\rangle \to_B^* \langle \mathtt{true}, S\rangle$

(b) $\langle C, S\rangle \to_C^* \langle \mathtt{skip}, S'\rangle$

(c) $\langle \mathtt{while}\ be\ \mathtt{do}\ C, S'\rangle \to_C^* \langle \mathtt{skip}, S''\rangle$

and we have to show that the conclusion of $(\Rightarrow_C \cdot 7)$ is in the set, i.e. that

$$\langle \texttt{while}\ be\ \texttt{do}\ C, S\rangle \ \rightarrow_C^*\ \langle \texttt{skip}, S''\rangle$$

Well, writing C_1 for $\texttt{while}\ be\ \texttt{do}\ C$ we can reason as follows:

$$\begin{array}{lll}
\langle C_1, S\rangle & \rightarrow_C & \langle \texttt{if}\ be\ \texttt{then}\ C\,;C_1\ \texttt{else skip}, S\rangle \text{ by } (\rightarrow_C \cdot 8)\\
& \rightarrow_C^* & \langle \texttt{if true then}\ C\,;C_1\ \texttt{else skip}, S\rangle \text{ by (a) and several } (\rightarrow_C \cdot 5)\text{s}\\
& \rightarrow_C & \langle C\,;C_1, S\rangle \text{ by } (\rightarrow_C \cdot 6)\\
& \rightarrow_C^* & \langle \texttt{skip}\,;C_1, S'\rangle \text{ by (b) and several } (\rightarrow_C \cdot 3)\text{s}\\
& \rightarrow_C & \langle C_1, S'\rangle \text{ by } (\rightarrow_C \cdot 4)\\
& \rightarrow_C^* & \langle \texttt{skip}, S''\rangle \text{ by (c)}
\end{array}$$

as required.

Proof of 2. This follows by rule induction on each of the relations $\rightarrow_I$,$\rightarrow_B$ and $\rightarrow_C$. Define three relations

$$\begin{array}{rcl}
\rightsquigarrow_I & \subseteq & (Iexp \times States) \times (Iexp \times States)\\
\rightsquigarrow_B & \subseteq & (Bexp \times States) \times (Bexp \times States)\\
\rightsquigarrow_C & \subseteq & (Com \times States) \times (Com \times States)
\end{array}$$

as follows:

$$\begin{array}{rll}
(ie, S) \rightsquigarrow_I (ie', S') & \text{iff} & S = S' \text{ and } \forall n \in \mathbb{Z}.(ie', S \Rightarrow_I n \text{ implies } ie, S \Rightarrow_I n)\\
(be, S) \rightsquigarrow_B (be', S') & \text{iff} & S = S' \text{ and } \forall b \in \mathbb{B}.(be', S \Rightarrow_B b \text{ implies } be, S \Rightarrow_B b)\\
(C, S) \rightsquigarrow_C (C', S') & \text{iff} & \forall S'' \in States.(C', S' \Rightarrow_C S'' \text{ implies } C, S \Rightarrow_C S'')
\end{array}$$

Then 2. is equivalent to proving $\langle ie, S\rangle \rightarrow_I \langle ie', S'\rangle$ implies $(ie, S) \rightsquigarrow_I (ie', S')$ and similarly for boolean expressions and commands. This follows by rule induction if we can show that $\rightsquigarrow_I$, $\rightsquigarrow_B$ and $\rightsquigarrow_C$ are closed under the rules defining $\rightarrow_I$,$\rightarrow_B$ and $\rightarrow_C$ respectively. We will just check the case of rule $(\rightarrow_C \cdot 8)$ and leave the other 14 cases as Exercises.

Since $(\rightarrow_C \cdot 8)$ has no hypotheses, we just have to show that

$$(\texttt{while}\ be\ \texttt{do}\ C, S) \rightsquigarrow_C (\texttt{if}\ be\ \texttt{then}\ (C\,;\texttt{while}\ be\ \texttt{do}\ C)\ \texttt{else skip}, S)$$

Writing C_1 for $\texttt{while}\ be\ \texttt{do}\ C$ this means showing that for all $S'' \in States$ if

$$\texttt{if}\ be\ \texttt{then}\ (C\,;C_1)\ \texttt{else skip}, S \Rightarrow_C S'' \tag{3.1}$$

then

$$C_1, S \Rightarrow_C S'' \tag{3.2}$$

But if (3.1) holds then it can only have been deduced by applying $(\Rightarrow_C \cdot 4)$ or $(\Rightarrow_C \cdot 5)$, and we consider each possibility in turn.

Case $(\Rightarrow_C \cdot 4)$ The derivation looks like this

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ be, S \Rightarrow_B true & C\,;C_1, S \Rightarrow_C S''\end{array}}{\texttt{if}\ be\ \texttt{then}\ (C\,;C_1)\ \texttt{else skip}, S \Rightarrow_C S''}\ (\Rightarrow_C \cdot 4)$$

but the subderivation $\mathcal{D}_2$ can only end in an instance of $(\Rightarrow_C \cdot 3)$, so we must have

$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \dfrac{\begin{array}{cc}\mathcal{D}_3 & \mathcal{D}_4\\ C, S \Rightarrow_C S' & C_1, S' \Rightarrow_C S''\end{array}}{C\,;C_1, S \Rightarrow_C S''}(\Rightarrow_C \cdot 3)\\ be, S \Rightarrow_B \mathit{true} & \end{array}}{\texttt{if}\ be\ \texttt{then}\ (C\,;C_1)\ \texttt{else skip}, S \Rightarrow_C S''}(\Rightarrow_C \cdot 4)$$

for some intermediate state S'. Given all that, it's easy to see that we can derive (3.2) like this

$$\dfrac{\begin{array}{ccc}\mathcal{D}_1 & \mathcal{D}_3 & \mathcal{D}_4\\ be, S \Rightarrow_B \mathit{true} & C, S \Rightarrow_C S' & \texttt{while}\ be\ \texttt{do}\ C, S' \Rightarrow_C S''\end{array}}{\texttt{while}\ be\ \texttt{do}\ C, S \Rightarrow_C S''}(\Rightarrow_C \cdot 7)$$

as required.

Case $(\Rightarrow_C \cdot 5)$ In this case the derivation of (3.1) looks like

$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ be, S \Rightarrow_B \mathit{false} & \texttt{skip}, S \Rightarrow_C S''\end{array}}{\texttt{if}\ be\ \texttt{then}\ (C\,;C_1)\ \texttt{else skip}, S \Rightarrow_C S''}(\Rightarrow_C \cdot 5)$$

but then $\mathcal{D}_2$ can only be an instance of $(\Rightarrow_C \cdot 1)$ so $S = S''$ and the derivation is

$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \dfrac{}{\texttt{skip}, S \Rightarrow_C S}(\Rightarrow_C \cdot 1)\\ be, S \Rightarrow_B \mathit{false} & \end{array}}{\texttt{if}\ be\ \texttt{then}\ (C\,;C_1)\ \texttt{else skip}, S \Rightarrow_C S}(\Rightarrow_C \cdot 5)$$

so that we can apply $(\Rightarrow_C \cdot 6)$ like this

$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ be, S \Rightarrow_B \mathit{false}\end{array}}{\texttt{while}\ be\ \texttt{do}\ C, S \Rightarrow_C S}(\Rightarrow_C \cdot 6)$$

to deduce (3.2) as required.

Proof of 3. It is easy to see that each of the relations $\rightsquigarrow_I$, $\rightsquigarrow_B$ and $\rightsquigarrow_C$ defined in the proof of 2. is reflexive and transitive, simply because logical implication is reflexive and transitive. Furthermore, 2. says that $\rightarrow_I \subseteq \rightsquigarrow_I$, $\rightarrow_B \subseteq \rightsquigarrow_B$ and $\rightarrow_I \subseteq \rightsquigarrow_I$; thus, because the reflexive transitive closure of a relation R is the *smallest* reflexive and transitive relation containing R, we have

$$\begin{array}{rcl}\langle ie, S\rangle \rightarrow_I^* \langle ie', S'\rangle & \text{implies} & (ie, S) \rightsquigarrow_I (ie', S')\\ \langle be, S\rangle \rightarrow_B^* \langle be', S'\rangle & \text{implies} & (be, S) \rightsquigarrow_B (be', S')\\ \langle C, S\rangle \rightarrow_C^* \langle C', S'\rangle & \text{implies} & (C, S) \rightsquigarrow_C (C', S')\end{array}$$

So, if $\langle ie, S\rangle \rightarrow_I^* \langle \underline{n}, S\rangle$ then $(ie, S) \rightsquigarrow_I (\underline{n}, S)$ and hence by the definition of $\rightsquigarrow_I$

$$\forall m \in \mathbb{Z}.(\underline{n}, S \Rightarrow_I m \text{ implies } ie, S \Rightarrow_I m)$$

Taking $m = n$ and using rule $(\Rightarrow_I \cdot 1)$ gives $ie, S \Rightarrow_I n$ as required. Similarly, if $\langle be, S\rangle \rightarrow_B^* \langle \underline{b}, S\rangle$ we get that $(be, S) \rightsquigarrow_B (\underline{b}, S)$ and we can use $(\Rightarrow_B \cdot 1)$ and the definition of $\rightsquigarrow_B$ to deduce $be, S \Rightarrow_B b$. Finally, much the same reasoning applies to commands, so that if $\langle C, S\rangle \rightarrow_C^* \langle \texttt{skip}, S'\rangle$ then $(C, S) \rightsquigarrow_C (\texttt{skip}, S')$ so by the definition of $\rightsquigarrow_C$ and rule $(\Rightarrow_C \cdot 1)$ we have $C, S \Rightarrow_C S'$.

□

The full proof of Theorem 16, filling in all the cases we missed out in parts 1. and 2., is obviously fairly lengthy but it doesn't involve any more concepts – it's just a matter of checking a lot more cases. The important thing is to understand and remember the broad outline, as you should then be able to fill in the details yourself without any great difficulty.

3.3.3 Implementing the evaluation semantics in ML

Translating the big-step evaluation semantics into ML is even easier than was the case for the small-step transition semantics. Once again, we rely on the fact that the evaluation relations are actually partial functions (this follows from the equivalent fact for the transition semantics and the equivalence of the big-step and small-step semantics which we just proved). As we have previously remarked, the inference rules defining the evaluation relations do not specify an evaluation order for expressions, but we *do* have to pick one in order to translate the rules into ML code.

```
(* bigstepi : IEXP -> (STATES -> int) *)
fun bigstepi ie (S:STATES) = case ie of
  N(n) => n
| Pvar(x) => lookup(x,S)
| Iop(iop,ie1,ie2) => let val n1 = bigstepi ie1 S
                          val n2 = bigstepi ie2 S
                      in
                          iopmeaning iop (n1,n2)
                      end;

(* bigstepb : BEXP -> (STATES -> bool) *)
fun bigstepb be (S:STATES) = case be of
  B(b) => b
| Bop(bop,ie1,ie2) => let val n1 = bigstepi ie1 S
                          val n2 = bigstepi ie2 S
                      in
                          bopmeaning bop (n1,n2)
                      end;

(* bigstepc : COM -> (STATES -> STATES) *)
fun bigstepc C (S:STATES) = case C of
  Skip => S
| Assign(x,ie) => let val n = bigstepi ie S
                  in update(S,x,n)
                  end
```

```
| Seq(C1,C2) => let val S' = bigstepc C1 S
                in bigstepc C2 S'
                end
| If(be,C1,C2) => if (bigstepb be S)
                  then bigstepc C1 S
                  else bigstepc C2 S
| While(be,C1) => if (bigstepb be S)
                  then let val S' = bigstepc C1 S
                       in bigstepc C S'
                       end
                  else S;
```

This gives a very natural interpreter for IMP programs. The functions `bigstepi`,`bigstepb` and `bigstepc` have, of course, exactly the same input/output behaviour as their small-step equivalents `ieval`, `beval` and `ceval`.

3.3.4 Semantic equivalence

One of the reasons for studying semantics which we mentioned in the introduction was to have a precise notion of when one command is equivalent to another. We are now in a position to define such a notion.

If C_1 and C_2 are IMP commands, then we say C_1 and C_2 are *semantically equivalent*, and write $C_1 \approx C_2$ if for all states S and S'

$$\begin{array}{c} \mathit{Ceval}(C_1)(S) \text{ is defined and equal to } S' \\ \text{if and only if} \\ \mathit{Ceval}(C_2)(S) \text{ is defined and equal to } S' \end{array}$$

Whilst *Ceval* was defined in terms of the small-step semantics, in view of Theorem 16 we obviously have

$$C_1 \approx C_2 \text{ iff } \forall S, S' \in \mathit{States}.(C_1, S \Rightarrow_C S' \equiv C_2, S \Rightarrow_C S')$$

It's clear that the relation $\approx \subseteq \mathit{Com} \times \mathit{Com}$ is an equivalence relation, i.e. it is reflexive, symmetric and transitive. Here's an example of an interesting equivalence:

Proposition 17 *For any three commands C,C',C''*

$$(\texttt{if}\ be\ \texttt{then}\ C\ \texttt{else}\ C')\,;C'' \ \approx\ \texttt{if}\ be\ \texttt{then}\ (C\,;C'')\ \texttt{else}\ (C'\,;C'')$$

Proof. Let

$$\begin{array}{rcl} C_1 & = & (\texttt{if}\ be\ \texttt{then}\ C\ \texttt{else}\ C')\,;C'' \\ C_2 & = & \texttt{if}\ be\ \texttt{then}\ (C\,;C'')\ \texttt{else}\ (C'\,;C'') \end{array}$$

There are two things to prove, firstly that if $C_1, S \Rightarrow_C S'$ then $C_2, S \Rightarrow_C S'$ and secondly that if $C_2, S \Rightarrow_C S'$ then $C_1, S \Rightarrow_C S'$ and we prove each in turn.

If $C_1, S \Rightarrow_C S'$ then the deduction of that fact must have ended in an instance of rule $(\Rightarrow_C \cdot 3)$:

$$\frac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ \texttt{if}\ be\ \texttt{then}\ C\ \texttt{else}\ C', S \Rightarrow_C S'' & C'', S'' \Rightarrow_C S' \end{array}}{(\texttt{if}\ be\ \texttt{then}\ C\ \texttt{else}\ C')\,;C'', S\ \Rightarrow_C\ S'}\ (\Rightarrow_C \cdot 3)$$

for some state S''. There are then two possibilities for the last rule used in $\mathcal{D}_1$, *viz.* $(\Rightarrow_C \cdot 4)$ and $(\Rightarrow_C \cdot 5)$. If the last rule was $(\Rightarrow_C \cdot 4)$ then the derivation must look like

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_3\\ be, S \Rightarrow_B \mathit{true}\end{array} \qquad \begin{array}{c}\mathcal{D}_4\\ C, S \Rightarrow_C S''\end{array}}{\texttt{if}\ be\ \texttt{then}\ C\ \texttt{else}\ C', S \Rightarrow_C S''}\,(\Rightarrow_C \cdot 4) \qquad \begin{array}{c}\mathcal{D}_2\\ C'', S'' \Rightarrow_C S'\end{array}}{(\texttt{if}\ be\ \texttt{then}\ C\ \texttt{else}\ C')\,;\,C'', S \Rightarrow_C S'}\,(\Rightarrow_C \cdot 3)$$

and so we can form the following derivation

$$\dfrac{\begin{array}{c}\mathcal{D}_3\\ be, S \Rightarrow_B \mathit{true}\end{array} \qquad \dfrac{\begin{array}{c}\mathcal{D}_4\\ C, S \Rightarrow_C S''\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ C'', S'' \Rightarrow_C S'\end{array}}{C\,;\,C'', S \Rightarrow_C S'}\,(\Rightarrow_C \cdot 3)}{\texttt{if}\ be\ \texttt{then}\ (C\,;\,C'')\ \texttt{else}\ (C'\,;\,C''), S \Rightarrow_C S'}\,(\Rightarrow_C \cdot 4)$$

to show that $C_2, S \Rightarrow_C S'$ as required. The case where the last rule of $\mathcal{D}_1$ is $(\Rightarrow_C \cdot 5)$ is similar, and omitted. Thus we have proved the first part of the proposition.

Similarly, starting with $C_2, S \Rightarrow_C S'$ we can deduce $C_1, S \Rightarrow_C S'$. Hence $C_1 \approx C_2$, as required. □

Proposition 18 *For an commands C_1,C_2 and boolean expression be,*

$$\textit{if}\ \ C_1 \approx C_2\ \ \textit{then}\ \ \texttt{while}\ be\ \texttt{do}\ C_1 \approx \texttt{while}\ be\ \texttt{do}\ C_2$$

Proof. We have to show that if $\texttt{while}\ be\ \texttt{do}\ C_1, S \Rightarrow_C S'$ then $\texttt{while}\ be\ \texttt{do}\ C_2, S \Rightarrow_C S'$ for any states S and S'. Once we've done that, it's clear that the converse holds too, just by symmetry.

The proof is by induction on the derivation $\mathcal{D}$ of $\texttt{while}\ be\ \texttt{do}\ C_1, S \Rightarrow_C S'$. There are two cases for the last rule applied in $\mathcal{D}$, *viz.* $(\Rightarrow_C \cdot 6)$ and $(\Rightarrow_C \cdot 7)$. If the last rule applied was $(\Rightarrow_C \cdot 6)$, then $\mathcal{D}$ looks like this:

$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ be, S \Rightarrow_B \mathit{false}\end{array}}{\texttt{while}\ be\ \texttt{do}\ C_1, S \Rightarrow_C S}\,(\Rightarrow_C \cdot 6)$$

so that $S = S'$. In this case, we can obviously form $\mathcal{D}'$, deriving $\texttt{while}\ be\ \texttt{do}\ C_2, S \Rightarrow_C S'$ like this:

$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ be, S \Rightarrow_B \mathit{false}\end{array}}{\texttt{while}\ be\ \texttt{do}\ C_2, S \Rightarrow_C S}\,(\Rightarrow_C \cdot 6)$$

If, on the other hand, the last rule used in $\mathcal{D}$ was $(\Rightarrow_C \cdot 7)$, then $\mathcal{D}$ looks like

$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ be, S \Rightarrow_B \mathit{true}\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ C_1, S \Rightarrow_C S''\end{array} \qquad \begin{array}{c}\mathcal{D}_3\\ \texttt{while}\ be\ \texttt{do}\ C_1, S'' \Rightarrow_C S'\end{array}}{\texttt{while}\ be\ \texttt{do}\ C_1, S \Rightarrow_C S'}\,(\Rightarrow_C \cdot 7)$$

for some state S''. In this case, we can apply the assumption that $C_1 \approx C_2$ to deduce from $\mathcal{D}_2$ that there must be a $\mathcal{D}_2'$ proving $C_2, S \Rightarrow_C S''$. We can also apply the induction

hypothesis to $\mathcal{D}_3$ to obtain a derivation $\mathcal{D}_3'$ which proves $\texttt{while}\, be\, \texttt{do}\, C_1, S'' \Rightarrow_C S'$. Putting these bits together we can form $\mathcal{D}'$ to be

$$\frac{\begin{array}{ccc} \mathcal{D}_1 & \mathcal{D}_2' & \mathcal{D}_3' \\ be, S \Rightarrow_B true & C_2, S \Rightarrow_C S'' & \texttt{while}\ be\ \texttt{do}\ C_2, S'' \Rightarrow_C S' \end{array}}{\texttt{while}\ be\ \texttt{do}\ C_2, S \Rightarrow_C S'} (\Rightarrow_C \cdot 7)$$

deriving $\texttt{while}\ be\ \texttt{do}\ C_2, S \Rightarrow_C S'$ as required. □

3.3.5 Congruences

There is an obvious question to be asked here which has considerable implications for how useful this notion of semantic equivalence is in practice. The most obvious reason for having a notion of equivalence is so that one can replace some command C_1 with an equivalent (but, let us say, more efficient) command C_2 in a larger program and know that the program would still give the same results (though, we hope, more quickly). However we do not yet know that this is sound.

We can express the property we want by introducing the notion of a *command context*, which is usually written $C[]$ and defined slightly informally to be 'a command with a hole in it'. In other words, a command context is just like a command, except that it can also contain one or more holes, which are written [], as subcommands. If $C[]$ is a command context and C_1 is a command, then $C[C_1]$ is the command which results from replacing all the occurences of the hole [] in $C[]$ with C_1. Now if $\mathcal{R}$ is a binary equivalence relation on commands, we say that $\mathcal{R}$ is a *congruence* if for all $C[]$, C_1 and C_2, if $(C_1, C_2) \in \mathcal{R}$ then $(C[C_1], C[C_2]) \in \mathcal{R}$. Another way of saying this is that $\mathcal{R}$ is a congruence if it is preserved by all the constructs of the command syntax. (Exercise: Why are the two definitions equivalent?)

What we want to know is that our semantic equivalence relation $\approx$ is a congruence, as that then allows us to 'substitute equals for equals'. Luckily, it turns out that $\approx$ *is* a congruence for IMP programs. The theorem can be proved directly from the operational semantics, and Proposition 18 is actually one of the steps in the proof (this is developed further in the Exercises), but it will also follow from the results of the next chapter. The Part II Concurrency Theory course develops these ideas further – for concurrent processes there are many natural notions of equivalence, some of which are congruences and some of which are not.

3.3.6 Semantic equivalence proofs as functions (optional)

If you study the proofs of Propositions 17 and 18, and have done some of the exercises on semantic equivalence then you should be able to see that the proofs all have a similar form. There are always two implications (for the two parts of the definition of semantic equivalence), each of which has the form

$$\text{if } C_1, S \Rightarrow_C S' \text{ then } C_2, S \Rightarrow_C S'$$

These are proved by looking at the possible derivations of $C_1, S \Rightarrow_C S'$ and showing that for each one we can construct a derivation of $C_2, S \Rightarrow_C S'$. For some of the proofs concerning looping constructs, this construction requires structural induction on the derivation of $C_1, S \Rightarrow_C S'$, whereas for simpler cases (like Proposition 17) it's just a matter of splitting

the each derivation of $C_1, S \Rightarrow_C S'$ into a small number of subderivations which can be reassembled to give a derivation of $C_2, S \Rightarrow_C S'$. In any case, one can view the *proof* that (any instance of) C_1 is semantically equivalent to (the corresponding instance of) C_2 as a pair of *functions* which take derivations to derivations – given a derivation about C_1, one function returns the derivation of the same thing about C_2 (and the other function does the converse). In the case that the proof requires structural induction on derivations the functions which express the proof will themselves be defined inductively. In the case where the proof relies on an assumption that two commands are equivalent, the function corresponding to the proof will take as extra arguments the functions which witness that equivalence.

All this can be formalised in ML, but the details are rather too messy to include here. Appendix A contains further details, and the code is available electronically for those who would really like to play with it.

3.4 Exercises

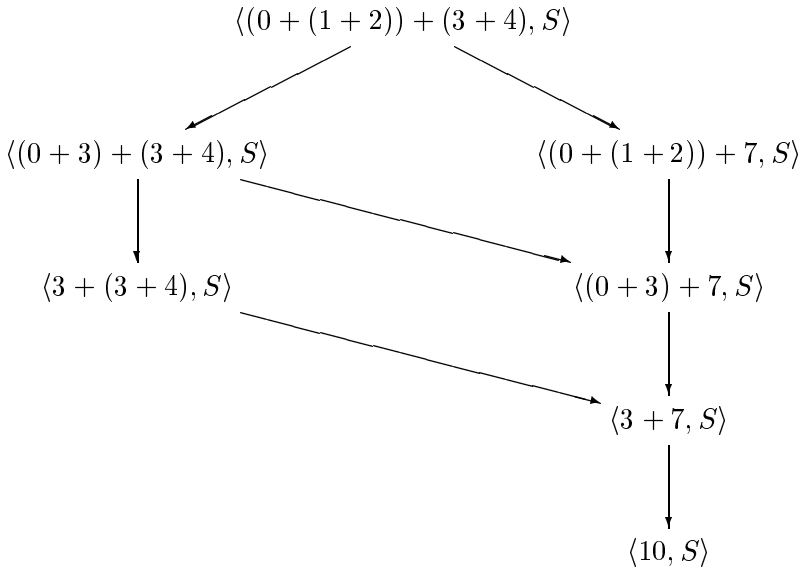
1. Check that you can give derivations in the one step transition semantics for each of the transitions in the example on page 25.

2. Do the proof of Proposition 13, that IMP expressions have no side-effects.

3. Complete the proof of Theorem 14, that transitions are deterministic.

4. Prove Lemma 15, that the evaluation of expressions always terminates.

5. How would you change the one-step transition semantics to specify a right-to-left, rather than a left-to-right evaluation order? How would you write the rules so that the evaluation order is unspecified and, for example, all the following sequences are allowed?



Prove that in this case, although the one-step transition relations $\rightarrow_I$ and $\rightarrow_B$ are no longer deterministic, the evaluation relations *Ieval*, *Beval* and *Ceval* which are defined in terms of them *are* still deterministic.

6. Produce a derivation in the big-step evaluation semantics for the example on page 31.

7. Finish the proof of Theorem 16, showing the equivalence of the small-step and big-step semantics for IMP.

8. There is a lot of choice about what constitutes a 'small step' in the transition semantics. Formulate a different version in which expressions all evaluate in just one step, but commands still generally take lots of little steps.

9. Show that for all $C \in Com$,

$$C\,;\mathtt{skip} \;\approx\; C \;\approx\; \mathtt{skip}\,;C$$

10. Show that for all commands C_1,C_2,C_3

$$(C_1\,;C_2)\,;C_3 \;\approx\; C_1\,;(C_2\,;C_3)$$

11. Give examples of $be \in Bexp$ and $C_1, C_2, C_3 \in Com$ for which

$$C1\,;(\mathtt{if}\ be\ \mathtt{then}\ C_2\ \mathtt{else}\ C_3) \;\not\approx\; \mathtt{if}\ be\ \mathtt{then}\ (C_1\,;C_2)\ \mathtt{else}\ (C_1\,;C_3)$$

12. Suppose $be, S \Rightarrow_B \mathit{true}$. Prove that there is no $S' \in States$ such that

$$\mathtt{while}\ be\ \mathtt{do}\ \mathtt{skip}, S \Rightarrow_C S'$$

13. Complete the proof that $\approx$ is a congruence using the evaluation semantics. In other words, show that if $C_1 \approx C_2$ then:

    (a) $(C_1\,;C) \;\approx\; (C_2\,;C)$ and $(C\,;C_1) \;\approx\; (C\,;C_2)$.

    (b) $(\mathtt{if}\ be\ \mathtt{then}\ C\ \mathtt{else}\ C_1) \;\approx\; (\mathtt{if}\ be\ \mathtt{then}\ C\ \mathtt{else}\ C_2)$ and $(\mathtt{if}\ be\ \mathtt{then}\ C_1\ \mathtt{else}\ C) \;\approx\; (\mathtt{if}\ be\ \mathtt{then}\ C_2\ \mathtt{else}\ C)$.

14. Prove that for any $be \in Bexp$ and $C \in Com$

$$\mathtt{while}\ be\ \mathtt{do}\ C \;\approx\; \mathtt{if}\ be\ \mathtt{then}\ (C\,;\mathtt{while}\ be\ \mathtt{do}\ C)\ \mathtt{else}\ \mathtt{skip}$$

    (You can do this in two natural ways – one is a direct proof from the evaluation semantics and the other uses the transition semantics.)

15. Augment the rules defining $Bexp$ with

$$\frac{be \in Bexp}{\mathtt{not}(be) \in Bexp}$$

    and extend the relation $\Rightarrow_B$ by the rule

$$\frac{be, S \Rightarrow_B b}{\mathtt{not}(be) \Rightarrow_B \neg b}\ (\Rightarrow_B \cdot 3)$$

    where $\neg \mathit{true} = \mathit{false}$ and $\neg \mathit{false} = \mathit{true}$.

Now augment the rules defining *Com* with

$$\frac{C \in Com \qquad be \in Bexp}{(\texttt{repeat}\ C\ \texttt{until}\ be) \in Com}$$

The intended meaning of (`repeat` C `until` be) is 'repeatedly execute C until the condition be evaluates to *true*'. Extend the evaluation relation $\Rightarrow_C$ by adding some rules which express this intention. Prove, using your new definition of $\Rightarrow_C$, that for any C and be

$$(\texttt{repeat}\ C\ \texttt{until}\ be) \ \approx\ C\,;(\texttt{while}\ \texttt{not}(be)\ \texttt{do}\ C)$$

16. Augment the commands of IMP with the following two new constructs

$$\frac{be \in Bexp}{\texttt{exitif}(be) \in Com} \qquad \frac{C_1 \in Com \qquad C_2 \in Com}{(C_1\ \texttt{orelse}\ C_2) \in Com}$$

The intended meaning of `exitif`(be) is to abort execution at the current state just in case be evaluates to *true*. The intended meaning of C_1 `orelse` C_2 is to execute C_1 until either it terminates normally (in which case C_2 is ignored completely), or until execution is aborted as above, in which case C_2 is executed. This is made precise by adding to the evaluation relations $\Rightarrow_B$,$\Rightarrow_I$ and $\Rightarrow_C$, the new relation $\Uparrow \subseteq Com \times States \times States$ (pronounced 'aborts at') with the following new rules:

$$\frac{be, S \Rightarrow_B true}{\texttt{exitif}(be), S \Uparrow S} \qquad \frac{be, S \Rightarrow_B false}{\texttt{exitif}(be), S \Rightarrow_C S}$$

$$\frac{C_1, S \Uparrow S'}{(C_1\,;C_2), S \Uparrow S'} \qquad \frac{C_1, S \Rightarrow_C S' \qquad C_2, S' \Uparrow S''}{(C_1\,;C_2), S \Uparrow S''}$$

$$\frac{be, S \Rightarrow_B true \qquad C_1, S \Uparrow S'}{\texttt{if}\ be\ \texttt{then}\ C_1\ \texttt{else}\ C_2, S \Uparrow S'} \qquad \frac{be, S \Rightarrow_B false \qquad C_2, S \Uparrow S'}{\texttt{if}\ be\ \texttt{then}\ C_1\ \texttt{else}\ C_2, S \Uparrow S'}$$

$$\frac{be, S \Rightarrow_B true \qquad C, S \Uparrow S'}{\texttt{while}\ be\ \texttt{do}\ C, S \Uparrow S'}$$

$$\frac{be, S \Rightarrow_B true \qquad C, S \Rightarrow_C S' \qquad \texttt{while}\ be\ \texttt{do}\ C, S' \Uparrow S''}{\texttt{while}\ be\ \texttt{do}\ C, S \Uparrow S''}$$

$$\frac{C_1, S \Rightarrow_C S'}{(C_1\ \texttt{orelse}\ C_2), S \Rightarrow_C S'} \qquad \frac{C_1, S \Uparrow S' \qquad C_2, S' \Rightarrow_C S''}{(C_1\ \texttt{orelse}\ C_2), S \Rightarrow_C S''}$$

$$\frac{C_1, S \Uparrow S' \qquad C_2, S' \Uparrow S''}{(C_1\ \texttt{orelse}\ C_2), S \Uparrow S''}$$

(a) For the new language, the old definition of $\approx$ still makes sense, but it is no longer a congruence. Why not? Refine the definition of semantic equivalence of commands to repair this.

(b) Call a command C *unexceptional* if $C, S \Uparrow S'$ holds for no states S, S'. For such a C, show that (`if` *be* `then` C' `else` C) is semantically equivalent to an expression built from *be*,C and C' using just `exitif`, `orelse` and `;`.

(c) Use the new language to define a macro `exit` and a new form of `while` construct with the property that `exit` will abort the smallest such enclosing new while loop.

17. If you know how to program in Prolog, experiment with implementing the two kinds of operational semantics for IMP in Prolog, rather than ML. What are the advantages and disadvantages of this approach?

18. Prove that IMP is Turing-powerful, by picking your favourite model of computation from the Computation Theory course (Turing machines, register machines or partial recursive functions) and showing how to simulate it in IMP.

Chapter 4

Denotational Semantics of IMP

The aim of this chapter is to present a different style of semantics for IMP in which the meanings of IMP phrases are given directly as (static) mathematical objects, rather than in terms of operational rules which express (dynamically) how evaluation proceeds. This approach has several payoffs. One is that we will be able to see straight away that the semantics is *compositional*. This means that the meaning of any phrase is determined solely by the meaning of its subphrases, and will show, amongst other things, that semantic equivalence is a congruence (cf. the remarks and exercises at the end of the last chapter).

Another major advantage of the denotational approach is that it gives an independent mathematical meaning to the syntactic constructs of our language. This enables one to compare the semantics of *different* languages and to identify the key concepts underlying them. For example, the way in which we will give a meaning to `while`-loops in IMP turns out to use the same techniques as are needed to give a denotational semantics to recursive functions in more sophisticated languages than IMP.

The mathematical spaces in which we will find the meanings of IMP phrases are certain kinds of partially ordered sets, called *complete partial orders* or *domains*. These structures are central to denotational semantics and can be used to treat nearly all programming language features you will meet. In particular, they can be used to give semantics to functional languages like ML and Haskell and to non-determinism and parallelism.[1]

From the point of view of this course of lectures, there is a slight pedagogical difficulty caused by the fact that our language IMP is so very trivial (no interesting datatypes, no procedures, no higher-order functions) that it is actually possible to explain its denotational semantics just in terms of partial functions between sets and without explicit mention of complete partial orders at all. However, since this naive approach does not scale up to more interesting languages, I will jump straight in to using the more general machinery of complete partial orders to give the semantics of IMP.[2]

[1] Actually there are still some things that the standard theory of domains doesn't deal with very nicely. These include dealing with sequentiality, computability and with 'fairness'. Denotational semantics is still an active research area, though the material in this course has been pretty stable and standard since the seventies.

[2] And anyway, this is the only way I can set any interesting exercises or exam questions... `:-)`

4.1 Complete Partial Orders

4.1.1 Partial orders

A binary relation $\sqsubseteq$ on a set D is a *partial order* if it is

reflexive $\forall d \in D.\ d \sqsubseteq d$

transitive $\forall d, d', d'' \in D.\ d \sqsubseteq d' \ \wedge\ d' \sqsubseteq d'' \ \Rightarrow d \sqsubseteq d''$

anti-symmetric $\forall d, d' \in D.\ d \sqsubseteq d' \ \wedge\ d' \sqsubseteq d \ \Rightarrow d = d'.$

A pair $(D, \sqsubseteq \subseteq D \times D)$ for which $\sqsubseteq$ is a partial order is called a *partially ordered set*, or *poset* for short. D is then called the *underlying set*, or *carrier*, of the poset. We will frequently abuse notation by just referring to 'the poset D' and using $\sqsubseteq$ to denote the partial order on a variety of differet posets.

The *least element*, or *bottom*, of a poset D, if it exists, is an element $\bot \in D$ such that

$$\forall d \in D.\ \bot \sqsubseteq d$$

Note that, by anti-symmetry, the bottom element of a poset, if it exists, is unique. If $\bot$ and $\bot'$ were two bottoms then we'd have $\bot \sqsubseteq \bot'$ and $\bot' \sqsubseteq \bot$ and hence $\bot = \bot'$. We will sometimes use subscripts to distinguish the bottoms of different cpos, but will also feel free to omit them.

4.1.2 Chains and least upper bounds

If $(D, \sqsubseteq)$ is a poset, then a (countable) *chain* c in D is a function $c\colon \mathbb{N} \to D$ such that $\forall n \in \mathbb{N}. c(n) \sqsubseteq c(n+1)$:

$$c(0) \ \sqsubseteq\ c(1) \ \sqsubseteq\ c(2) \ \sqsubseteq \cdots$$

If c is a chain, we will usually write c_n rather than $c(n)$.

An *upper bound* for a chain c in D is an element $d \in D$ which dominates all the elements of the chain:

$$\forall n \in \mathbb{N}.\ c_n \sqsubseteq d$$

Clearly, for a given chain, there may be no upper bound or there may be many upper bounds. The *least upper bound* $\bigsqcup_{n=0}^{\infty} c_n$ of the chain c, if it exists, is an upper bound which is $\sqsubseteq$ all other upper bounds:

$$\forall d \in D.\ \left(\bigsqcup_{n=0}^{\infty} c_n\right) \sqsubseteq d \iff (\forall n \in \mathbb{N}.\ c_n \sqsubseteq d)$$

Least upper bounds are also known as *lub*s (for obvious reasons) or *sup*s (*sup* is short for *supremum*, so *sups* is short for *suprema* (mixing English and Latin plurals)). Least upper bounds, like bottoms, are unique if they exist as a trivial consequence of the fact that $\sqsubseteq$ is antisymmetric.

A *complete partial order*, or *cpo* for short, is a poset which has least upper bounds for all (countable) chains. We will also sometimes refer to cpos as *domains*. [3]

Examples:

[3] What we are calling a cpo is often called an ω-cpo in the literature, the ω indicating that only lubs of *countable* chains are required to exist. Many authors also require cpos to have a least element, and would refer to our potentially bottomless ones as *predomains*. Even more confusingly, the term 'domain' is frequently taken to mean a cpo with some particular more complicated extra structure, which we will have no need of here.

1. If X is any set, then the powerset of X

$$\mathbb{P}(X) \stackrel{\text{def}}{=} \{S \mid S \subseteq X\}$$

ordered by $\subseteq$ is a cpo. The lub of a chain $S_0 \subseteq S_1 \subseteq \cdots$ is the *union* $\bigcup_{n=0}^{\infty} S_n$. The cpo $(\mathbb{P}(X), \subseteq)$ also has a least element: the empty set $\emptyset \subseteq X$.

2. For any sets X and Y, the set $X \rightharpoonup Y$ of partial functions from X to Y

$$X \rightharpoonup Y \stackrel{\text{def}}{=} \{f \in \mathbb{P}(X \times Y) \mid \forall x \in X. \forall y, y' \in Y.\, (x, y) \in f \wedge (x, y') \in f \Rightarrow y = y'\}$$

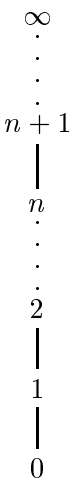
ordered by $\subseteq$ is a cpo with lubs of chains given by union and least element the empty set (i.e. the always undefined partial function), just as in 1.

3. For any set X, defining $\sqsubseteq$ to be the equality relation on X, i.e. $x \sqsubseteq x' \iff x = x'$, makes X into a cpo, called the *discrete* cpo on X. Note that any chain c in $(X, =)$ is constant, $c_0 = c_1 = \cdots$, and so trivially has a least upper bound c_0. $(X, =)$ has a bottom just when X has precisely one element.

4. Let $\Omega = \mathbb{N} \cup \{\infty\}$ (where ∞ is just a suggestive name for some element distinct from all those in $\mathbb{N}$), and define $\sqsubseteq$ on Ω by

$$x \sqsubseteq x' \iff (x, x' \in \mathbb{N} \wedge x \leq x') \vee (x' = \infty)$$

Then Ω is a cpo which may be pictured like this:



4.1.3 Continuous functions

If $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ are cpos, and $f\colon D \to E$ is a function between their underlying sets, then f is *monotonic* if it preserves order:

$$\forall d, d' \in D.\, d \sqsubseteq_D d' \Rightarrow f(d) \sqsubseteq_E f(d')$$

It is *continuous* if it is monotonic and also preserves least upper bounds of all chains in D:

$$f\left(\bigsqcup_{n=0}^{\infty} c_n\right) = \bigsqcup_{n=0}^{\infty} f(c_n) \tag{4.1}$$

Note that $f \circ c : \mathbb{N} \to E$ *is* a chain because f is monotonic.

Actually, one half of Equation 4.1 follows directly from monotonicity, since for any $m \in \mathbb{N}$

$$c_m \sqsubseteq_D \bigsqcup_{n=0}^{\infty} c_n$$

by the definition of upper bounds, so monotonicity gives

$$f(c_m) \sqsubseteq_E f\left(\bigsqcup_{n=0}^{\infty} c_n\right)$$

which says $f\,(\bigsqcup_{n=0}^{\infty} c_n)$ is *an* upper bound for the chain $f \circ c$. Therefore, it is $\sqsupseteq_E$ the *least* upper bound:

$$\bigsqcup_{m=0}^{\infty} f(c_m) \sqsubseteq_E f\left(\bigsqcup_{n=0}^{\infty} c_n\right)$$

which means that Equation 4.1 holds iff

$$f\left(\bigsqcup_{n=0}^{\infty} c_n\right) \sqsubseteq \bigsqcup_{n=0}^{\infty} f(c_n)$$

You should check (Exercise) that

1. For any cpo D, the identity function $\mathrm{id}_D \stackrel{\mathrm{def}}{=} \lambda d \in D.d : D \to D$ is always continuous.[4]
2. If $f : D \to E$ and $g : E \to F$ are continuous then the composition

   $$g \circ f \stackrel{\mathrm{def}}{=} \lambda d \in D.g(f(d)) \; : D \to F$$

   is continuous.
3. If X is a discrete cpo, then any function $f : X \to D$ is continuous.

4.1.4 Binary product of cpos

If D_1 and D_2 are cpos, then their binary product $D_1 \times D_2$ has as underlying set

$$D_1 \times D_2 \;=\; \{(d_1, d_2) \mid d_1 \in D_1 \;\wedge\; d_2 \in D_2\}$$

with the partial order

$$(d_1, d_2) \sqsubseteq (d_1', d_2') \iff d_1 \sqsubseteq d_1' \;\wedge\; d_2 \sqsubseteq d_2'$$

It's easy to check that this *is* a cpo, with least upper bounds calculated 'compontentwise'. If $c : \mathbb{N} \to D_1 \times D_2$ is given by $c_n = (c_n', c_n'')$ then

$$\bigsqcup_{n=0}^{\infty} c_n \;=\; \left(\bigsqcup_{n=0}^{\infty} c_n' \,,\; \bigsqcup_{n=0}^{\infty} c_n''\right)$$

If D and E both have bottoms, then so does $D \times E$, *viz.* the pair $(\bot_D, \bot_E)$.

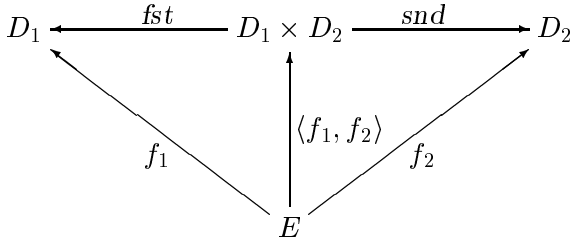
[4] Recall that $\lambda x \in A.e$, where e is some expression possibly involving the variable x, means 'the function which sends any $a \in A$ to $e[a/x]$'. It's essentially the same as `fn (x:A)=>e` in ML.

There are continuous *projection* functions $fst\colon D_1 \times D_2 \to D_1$ and $snd\colon D_1 \times D_2 \to D_2$ given by $fst(d_1, d_2) = d_1$ and $snd(d_1, d_2) = d_2$.

Given continuous functions $f_1\colon E \to D_1$ and $f_2\colon E \to D_2$, we get a continuous function $\langle f_1, f_2\rangle\colon E \to D_1 \times D_2$ defined by $\langle f_1, f_2\rangle(e) = (f_1(e), f_2(e))$. This obviously satisfies the pair of equations

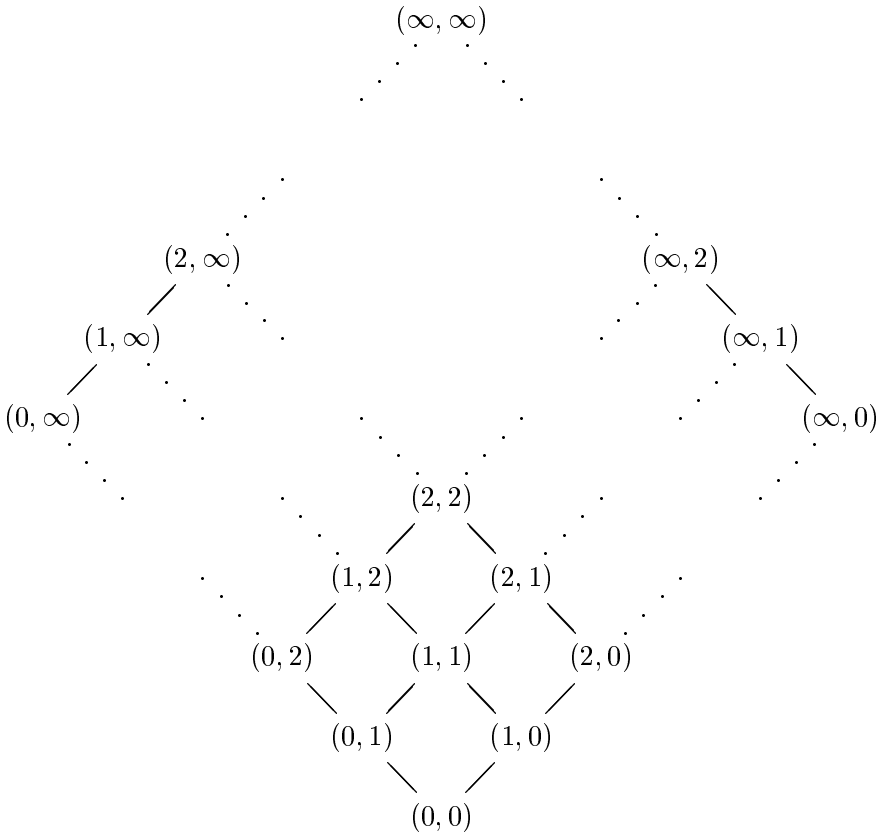
$$\begin{aligned} fst \circ \langle f_1, f_2\rangle &= f_1 \\ snd \circ \langle f_1, f_2\rangle &= f_2 \end{aligned}$$

which we can summarise in a diagram:



Given $f_1\colon E_1 \to D_1$ and $f_2\colon E_2 \to D_2$ continuous, we define the continuous function $f_1 \times f_2\colon E_1 \times E_2 \to D_1 \times D_2$ to be $\langle f_1 \circ fst, f_2 \circ snd\rangle$. In other words $(f_1 \times f_2)(e_1, e_2) = (f_1(e_1), f_2(e_2))$.

For example, if Ω is the cpo defined at the end of Section 4.1.2,then $\Omega \times \Omega$ is the cpo which we can draw like this:



4.1.5 Exponentation of cpos

If D and E are cpos, then the *exponential*, or *(continuous) function space*, cpo $[D \to E]$ has as underlying set

$$\{f\colon D \to E \mid f \text{ is continuous}\}$$

with the order

$$f \sqsubseteq f' \iff \forall d \in D.\, f(d) \sqsubseteq f'(d)$$

Note that we use much the same notation for the set of all functions between two sets and the cpo of all continuous functions between two cpos. If X and Y are sets, regarded as discrete cpos, then the exponential cpo $[X \to Y]$ is just the discrete cpo on the set of all functions from X to Y.

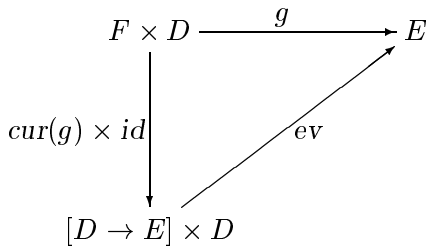
You should check that $[D \to E]$ really *is* a cpo, with lubs of chains calculated 'pointwise':

$$\bigsqcup_{n=0}^{\infty} f_n \;=\; \lambda d \in D.\, \left(\bigsqcup_{n=0}^{\infty} f_n(d) \right)$$

If E has a bottom, then $[D \to E]$ has a bottom, given by $\bot_{[D \to E]} = \lambda d \in D. \bot_E$, the constant $\bot_E$ function (which is easily seen to be continuous).

The *evaluation* function $ev\colon [D \to E] \times D \to E$ is the continuous function defined by $ev(f, d) = f(d)$.

Given a continuous $g\colon F \times D \to E$, there is a continuous function $cur(g)\colon F \to [D \to E]$, called the *Currying* of g, where for each $x \in F$, $cur(g)(x) = \lambda d \in D.\, g(x, d)$. Thus $cur(g)$ satisfies $g = ev \circ (cur(g) \times id)$, which we may draw as a diagram:



4.1.6 Lifting

Given a cpo D, the *lifted* cpo $D_\bot$ is obtained by adding a new bottom element below all those in D. Formally, the underlying set of $D_\bot$ is

$$\{[d] \mid d \in D\} \cup \{\bot\}$$

with the order

$$x \sqsubseteq x' \iff (x = \bot) \vee (\exists d, d' \in D.\, x = [d] \wedge x' = [d'] \wedge d \sqsubseteq d')$$

where $[\cdot]$ is a formal function to 'mark' all the elements of D in such a way as to make them distinct from the new $\bot$, so for any $d, d' \in D$ ($[d] = [d'] \;\Rightarrow\; d = d'$ and $\bot \neq [d]$.

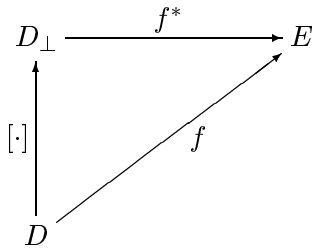
Clearly, $D_\bot$ has a bottom $\bot$. The function $[\cdot]\colon D \to D_\bot$ is continuous and order-reflecting, in the sense that

$$[d] \sqsubseteq [d'] \;\Rightarrow\; d \sqsubseteq d'$$

If $f\colon D \to E$ is continuous and E has a bottom, then we can *lift* f to get a continuous function $f^*\colon D_\perp \to E$ defined by

$$f^*(x) \stackrel{\text{def}}{=} \begin{cases} f(d) & \text{if } x = [d] \text{ for some } d \in D \\ \perp & \text{if } x = \perp \end{cases}$$

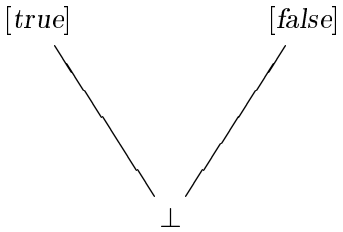
So that $f = f^* \circ [\cdot]$ which can be drawn as



The operation $f \mapsto f^*$ is itself a continuous function

$$(\cdot)^*\colon [D \to E] \to [D_\perp \to E]$$

For example, if $\mathbb{B}$ is the set $\{\mathit{true}, \mathit{false}\}$, regarded as a discrete cpo,then the cpo $\mathbb{B}_\perp$ (which is *not* discrete) looks like



A cpo like this, which is the lift of a discrete cpo, is said to be *flat*. For a flat cpo, $x \sqsubseteq x' \iff (x = \perp) \vee (x = x')$.

4.1.7 Conditionals

Regarding $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$ as a discrete cpo, for each cpo D there is a continuous function $\mathbb{B} \times D \times D \to D$ called the *conditional function* for D, whose value at $(b, d_1, d_2) \in \mathbb{B} \times D \times D$ is

$$(b \Rightarrow d_1 \mid d_2) \stackrel{\text{def}}{=} \begin{cases} d_1 & \text{if } b = \mathit{true} \\ d_2 & \text{if } b = \mathit{false} \end{cases}$$

4.1.8 Least fixed points

Suppose D is a cpo with a bottom, $\perp$, and $f\colon D \to D$ is a continuous function. Consider the sequence of elements of D

$$\perp,\ f(\perp),\ f(f(\perp)) = f^2(\perp),\ f^3(\perp), \ldots$$

We have

$$\begin{array}{rcl} \perp & \sqsubseteq & f(\perp) \text{ by definition of bottom} \\ f(\perp) & \sqsubseteq & f(f(\perp)) = f^2(\perp) \text{ by montonicity and previous line} \\ f^2(\perp) & \sqsubseteq & f^3(\perp) \text{ for the same reason} \end{array}$$

and so on. Thus

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq f^3(\bot) \sqsubseteq \cdots$$

is a chain in D, and therefore has a least upper bound:

$$\mathit{fix}(f) \stackrel{\text{def}}{=} \bigsqcup_{n=0}^{\infty} f^n(\bot)$$

where, inductively, $f^0(\bot) = \bot$ and $f^{n+1}(\bot) = f(f^n(\bot))$.

Since f is continuous, we have

$$\begin{aligned} f(\mathit{fix}(f)) &= \bigsqcup_{n=0}^{\infty} f(f^n(\bot)) \\ &= \bigsqcup_{n=0}^{\infty} f^{n+1}(\bot) \end{aligned}$$

But $\bigsqcup_{n=0}^{\infty} f^{n+1}(\bot)$ is the lub of the chain $f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \cdots$ and this is clearly the same as the lub of the chain $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \cdots$. In other words

$$f(\mathit{fix}(f)) \;=\; \mathit{fix}(f)$$

so $\mathit{fix}(f)$ is a fixed point of f.

More generally, a *prefixed point* of f is an element $d \in D$ such that $f(d) \sqsubseteq d$. Given such a d, we can reason as follows:

$$\begin{array}{rcccll} \bot & \sqsubseteq & d & & & \text{since } \bot \text{ is bottom, so} \\ f(\bot) & \sqsubseteq & f(d) & \sqsubseteq & d & \text{as } f \text{ monotone \& } d \text{ prefixed point} \\ f^2(\bot) & \sqsubseteq & f(d) & \sqsubseteq & d & \text{for the same reason} \end{array}$$

etc. Thus $\forall n \in \mathbb{N}.\ f^n(d) \sqsubseteq d$, which means that d is an upper bound for the chain $\{f^n(\bot) \mid n \in \mathbb{N}\}$. So d is $\sqsupseteq$ the least upper bound of that chain, i.e.

$$\mathit{fix}(f) \;=\; \bigsqcup_{n=0}^{\infty} f^n(\bot) \;\sqsubseteq\; d$$

which means that $\mathit{fix}(f)$ is the least element of the set of all prefixed points of f. In particular, it is also the *least fixed point* of f.[5]

In fact the operation $f \mapsto \mathit{fix}(f)$ actually determines a continuous function $\mathit{fix}\colon [D \to D] \to D$.

4.1.9 Fixpoint induction

There is a useful technique for proving properties of least fixed points, due to Scott and de Bakker, which is called *fixpoint induction* (or sometimes *Scott induction*). Assume that

[5] Of course, this all looks very familiar – it appears to be essentially the same as the arguments we used right at the start of these notes to justify inductive definitions. In fact, there are some slight differences. Previously we showed (in the 'downwards' construction) that any monotone function on a complete lattice (i.e. a set with greatest lower bounds of all subsets, which in that case were intersections) has a least fixed point. Here we have just shown that a continuous function (which is rather more than just a monotone function) over a cpo with a $\bot$ (which is rather less than a complete lattice) has a least fixed point.

D is a cpo with a bottom and that $f: D \to D$ is a continuous function. Then if $P(\cdot)$ is a particular kind of predicate over D, we can deduce $P(\mathit{fix}(f))$ by showing

$$\frac{}{P(\bot)} \qquad \text{and} \qquad \frac{P(x)}{P(f(x))}$$

But what is the special condition which $P(\cdot)$ has to satisfy to make this valid? We can find the answer just by trying to *prove* that the induction principle above is sound, and seeing what we have to assume about $P(\cdot)$ to make the proof go through. So we'll assume that we've shown the two things above and try to deduce $P(\mathit{fix}(f))$.

Recall that $\mathit{fix}(f) = \bigsqcup_{n=0}^{\infty} f^n(\bot)$. So we start by observing that as we've assumed that $P(\bot)$ holds, we have that $P(f^0(\bot))$ holds. Now assume that $P(f^n(\bot))$ holds. By the second rule above, this means that $P(f(f^n(\bot)))$ holds. But that's just $P(f^{n+1}(\bot))$. Hence we can conclude by mathematical induction that $P(f^n(\bot))$ holds for all n. But what we want to know is that $P\left(\bigsqcup_{n=0}^{\infty} f^n(\bot)\right)$ holds. To make this leap, we have to know that P itself has a special property. Clearly, a sufficient condition on P is that whenever $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ is a chain in D such that for all n, $P(x_n)$ holds, $P\left(\bigsqcup_{n=0}^{\infty} x_n\right)$ holds. In words, whenever we have a chain, all of whose elements satisfy P, the limit of the chain also satisfies P. A predicate with this property is said to be *inclusive* or *chain-closed*, and we have just shown

Theorem 19 (Fixpoint Induction) *If D is a cpo with a bottom, $f: D \to D$ is a continuous function and $P(\cdot)$ is an inclusive predicate on D, then*

$$(P(\bot) \text{ and } \forall x \in D.\, P(x) \Rightarrow P(f(x))) \qquad \text{implies} \qquad P(\mathit{fix}(f))$$

□

(Of course, we can, as usual, identify a predicate on D with the subset of elements of D which satisfy it, so we can speak of inclusive *subsets*, rather than predicates.)

As it stands, Theorem 19 doesn't *seem* to be very useful. After all, checking that a predicate is inclusive looks like quite a lot of work, in general; so all we save ourselves by appealing to the fixpoint induction theorem is a little application of ordinary mathematical induction. The reason that the method *is* useful is that we can often save ourselves the bother of explicitly checking from first principles that a particular predicate is inclusive. This is because there are a whole range of ways in which we can build inclusive predicates from other inclusive predicates. Hence we can often tell that a predicate is inclusive just by looking at it and seeing that it's made up from "inclusivity-preserving" operations (much as we can often tell that a function is continuous just by inspection because we know things like 'composition preserves continuity'). This is developed further in the Exercises.

Here's an example of a proof by fixpoint induction:

Theorem 20 *If D and E are cpos with bottom, $f: D \to E$ is a strict continuous function (i.e. $f(\bot) = \bot$) and $h: D \to D$ and $g: E \to E$ are continuous functions with $f \circ h = g \circ f$, then*

$$\mathit{fix}(g) \;=\; f(\mathit{fix}(h))$$

Proof. We will prove $\mathit{fix}(g) = f(\mathit{fix}(h)$ by proving that the left hand side is $\sqsubseteq$ the right hand side and vice versa. Firstly

$$\begin{array}{rcll} f(\mathit{fix}(h)) & = & f(h(\mathit{fix}(h))) & \text{defn. of fixpoint} \\ & = & g(f(\mathit{fix}(h))) & \text{assumption} \end{array}$$

So $f(\mathit{fix}(h))$ is a fixpoint of g, and hence $\mathit{fix}(g) \sqsubseteq f(\mathit{fix}(h))$ by minimality of least fixed points.

Now we use fixpoint induction to prove the reverse inequality. We take the predicate $P(x)$ over D to be

$$P(x) \stackrel{\text{def}}{=} (f(x) \sqsubseteq \mathit{fix}(g))$$

which is an inclusive predicate (Exercise). Then we need to check firstly that $P(\bot)$ holds. This means checking $f(\bot) \sqsubseteq \mathit{fix}(g)$ which is immediate since we assumed that f was strict, so $f(\bot) = \bot \sqsubseteq \mathit{fix}(g)$. Next we need to check that under the hypothesis that $P(x)$ holds, we can deduce $P(h(x))$ holds. So the hypothesis is that $f(x) \sqsubseteq \mathit{fix}(g)$ and we reason as follows:

$$\begin{array}{rcll} f(h(x)) & = & g(f(x)) & \text{assumption} \\ & \sqsubseteq & g(\mathit{fix}(g)) & \text{monotonicity of } g \text{ and hypothesis} \\ & = & \mathit{fix}(g) & \text{defn. of fixpoint} \end{array}$$

So $P(h(x))$ holds as required, and we can use Theorem 19 to deduce $P(\mathit{fix}(h))$, i.e. that $f(\mathit{fix}(h)) \sqsubseteq \mathit{fix}(g)$ as required. □

You can find more interesting Exercises on fixpoint induction in past examination questions (e.g. 1993 Paper 8 Question 10).

4.2 Denotational Semantics of IMP

Having dealt with the mathematical preliminaries, we can now give the denotational semantics of our language using cpos.

4.2.1 Semantics of integer and boolean expressions

We define functions

$$\begin{array}{rcl} [\![-]\!] & : & \mathit{Iexp} \to (\mathit{States} \to \mathbb{Z}) \\ [\![-]\!] & : & \mathit{Bexp} \to (\mathit{States} \to \mathbb{B}) \end{array}$$

by induction on the structure of expressions.[6] You can pronounce $[\![e]\!]$ as 'the meaning of e', and the definitions are as follows, for any $S \in \mathit{States}$:

Constants For $n \in \mathbb{Z}$, $b \in \mathbb{B}$

$$\begin{array}{rcl} [\![\underline{n}]\!](S) & \stackrel{\text{def}}{=} & n \\ [\![\underline{b}]\!](S) & \stackrel{\text{def}}{=} & b \end{array}$$

Variables For $x \in \mathit{Pvar}$

$$[\![x]\!](S) \stackrel{\text{def}}{=} S(x)$$

[6] The symbols [and] are called *semantic brackets* and part of their purpose is to emphasize that what's inside them is to be treated as a piece of syntax, rather than a mathematical expression. Since we already have some conventions (involving underlining and the use of different typefaces) for this, that aspect is not quite so important for us.

Compound expressions For $iop \in Iop$, $bop \in Bop$, $ie_1, ie_2 \in Iexp$

$$[\![ie_1 \; \underline{iop} \; ie_2]\!](S) \stackrel{\text{def}}{=} ([\![ie_1]\!](S)) \; iop \; ([\![ie_2]\!](S))$$
$$[\![ie_1 \; \underline{bop} \; ie_2]\!](S) \stackrel{\text{def}}{=} ([\![ie_1]\!](S)) \; bop \; ([\![ie_2]\!](S))$$

Proposition 21 *For all $ie \in Iexp$, $be \in Bexp$, $S \in States$, $n \in \mathbb{Z}$, $b \in \mathbb{B}$:*

$$ie, S \Rightarrow_I n \iff [\![ie]\!](S) = n$$
$$be, S \Rightarrow_B b \iff [\![be]\!](S) = b$$

In other words, the $[\![-]\!]$ functions are identical to the operationally defined functions Ieval and Beval.

Proof. This is an elementary structural induction, and is left as an Exercise. □

So the meaning of an expression is a (total) function from *States* to whichever of $\mathbb{Z}$ and $\mathbb{B}$ is appropriate. We can regard *States*,$\mathbb{Z}$ and $\mathbb{B}$ as discrete cpos, in which case the functions are trivially continuous.

4.2.2 Semantics of commands

Giving a semantics to commands is more complicated than giving a semantics to expressions. This is because we have to deal with two (closely related) extra features of commands: potential non-termination and looping constructs. Recall that for command C, we defined *Ceval*(C) to be a *partial* function $States \rightharpoonup States$. An alternative way to express this partiality is by taking total functions into the flat cpo $States_\perp$:

Proposition 22 *For any sets X and Y, there is a bijective correspondence between the set $(X \rightharpoonup Y)$ of partial functions from X to Y and the elements of the cpo $[X \to Y_\perp]$ (where we regard X and Y as discrete cpos).*

Proof. Firstly note that *any* function from X to the underlying set of $Y_\perp$ is continuous, because X is discrete.

We define a function I from $(X \rightharpoonup Y)$ to the underlying set of $[X \to Y_\perp]$ by, for $f \in (X \rightharpoonup Y)$, $x \in X$

$$I(f)(x) \stackrel{\text{def}}{=} \begin{cases} [f(x)] & \text{if } f(x) \text{ is defined} \\ \perp & \text{otherwise} \end{cases}$$

The inverse function I^{-1} sends $g\colon X \to Y_\perp$ to $I^{-1}(g)\colon X \rightharpoonup Y$ where for any $x \in X$, $I^{-1}(g)(x)$ is defined iff $g(x) = [y]$ for some $y \in Y$, and in this case $I^{-1}(g)(x) = y$.

Clearly $I^{-1}(I(f)) = f$ and $I(I^{-1}(g)) = g$. □

Note that the completely undefined partial function corresponds to the constantly $\perp$ function $(\lambda x \in X.\, \perp)\colon X \to Y_\perp$. When we define the denotation of a command as a continuous function from $States \to States_\perp$, non-termination of the command will be represented by its denotation returning $\perp$.

We can now define

$$[\![-]\!]\colon Com \to [States \to States_\perp]$$

by induction on the structure of commands. The meaning of any command will be a (trivially) continuous function from the discrete cpo *States* to the flat cpo $States_\perp$.

Skip

$$[\![\texttt{skip}]\!] \stackrel{\text{def}}{=} \lambda S \in \mathit{States}.[S]$$

Assignment If $x \in \mathit{Pvar}$ and $ie \in \mathit{Iexp}$ then the meaning of the assignment $x := ie$ in a state S is the element of $\mathit{States}_\perp$ corresponding to the state S updated to send the variable x to the integer $[\![ie]\!](S)$ (with apologies for the overloading of square brackets):

$$[\![x := ie]\!] \stackrel{\text{def}}{=} \lambda S \in \mathit{States}.[S[[\![ie]\!](S)/x]]$$

Sequencing For $C_1, C_2 \in \mathit{Com}$ we have $[\![C_1]\!]\colon \mathit{States} \to \mathit{States}_\perp$ and $[\![C_2]\!]\colon \mathit{States} \to \mathit{States}_\perp$ and we want to compose them together, for which we need to use the $(\cdot)^*$ operation (Section 4.1.6)

$$\mathit{States} \xrightarrow{[\![C_1]\!]} \mathit{States}_\perp \xrightarrow{([\![C_2]\!])^*} \mathit{States}_\perp$$

So we define

$$[\![C_1\,;C_2]\!] \stackrel{\text{def}}{=} \lambda S \in \mathit{States}.[\![C_2]\!]^*\,([\![C_1]\!](S))$$

Conditionals For $be \in \mathit{Bexp}$, $C_1, C_2 \in \mathit{Com}$ we can give the meaning of $\texttt{if}\,be\,\texttt{then}\,C_1\,\texttt{else}\,C_2$ using the continuous conditional function which we defined in Section 4.1.7:

$$[\![\texttt{if}\ be\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!] \stackrel{\text{def}}{=} \lambda S \in \mathit{States}.\,([\![be]\!](S) \Rightarrow [\![C_1]\!](S) \mid [\![C_2]\!](S))$$

While-loops This is where we need the interesting bit of the order structure of cpos. Recall that (cf. the transition semantics and the exercises at the end of the last chapter)

$$\texttt{while}\ be\ \texttt{do}\ C \;\approx\; \texttt{if}\ be\ \texttt{then}\ (C\,;\texttt{while}\ be\ \texttt{do}\ C)\ \texttt{else}\ \texttt{skip}$$

We want the denotation of the command on the left to be equal to that of the command on the right. If we write $f \in [\mathit{States} \to \mathit{States}_\perp]$ for the as yet unknown denotation of $\texttt{while}\ be\ \texttt{do}\ C$, this means that we want f to satisfy

$$f \;=\; \lambda S \in \mathit{States}.([\![be]\!](S) \Rightarrow f^*([\![C]\!](S)) \mid [S])$$

The expression on the right is a continuous function of f, which means that we can find an f satsifying the equation by taking the least fixed point of that function.

$$[\![\texttt{while}\ be\ \texttt{do}\ C]\!] \stackrel{\text{def}}{=} \mathit{fix}(\Phi)$$

where $\Phi\colon [\mathit{States} \to \mathit{States}_\perp] \to [\mathit{States} \to \mathit{States}_\perp]$ is defined by

$$\Phi \stackrel{\text{def}}{=} \lambda f \in [\mathit{States} \to \mathit{States}_\perp].\lambda S \in \mathit{States}.([\![be]\!](S) \Rightarrow f^*([\![C]\!](S)) \mid [S])$$

The fact that for any command C, $[\![C]\!]$ is indeed a well-defined continuous function is relatively easy to see. The only thing that might not be completely obvious is that the operation Φ which we used to define the meaning of `while` commands is continuous, but this can be seen from the fact that it is built up out of continuity-preserving operations. Alternatively, you can prove it from first principles (Exercise). To know that $\mathit{fix}(\Phi)$ exists

we also need to know that Φ is an operator on a domain with a least element, which it is, since $[States \to States_\perp]$ has as least element the constantly $\perp$ function.

It is worth trying to understand just how the denotation of `while` commands is constructed. Assume that there is some command Ω such that $[\![\Omega]\!] = \lambda S \in States.\perp$, then $[\![\texttt{while}\, be\, \texttt{do}\, C]\!] = fix(\Phi) \in [States \to States_\perp]$ is constructed as the least upper bound of a chain of functions $f : \mathbb{N} \to [States \to States_\perp]$, starting with the constant bottom function:

$$
\begin{array}{lcl}
f_n & : & States \to States_\perp \\
f_0 & = & \lambda S \in States.\perp \\
 & = & [\![\Omega]\!] \\
f_1 & = & \Phi(f_0) \\
 & = & \lambda S \in States.([\![be]\!](S) \Rightarrow f_0^*([\![C]\!](S)) \mid [S]) \\
 & = & \lambda S \in States.([\![be]\!](S) \Rightarrow \perp \mid [S]) \\
 & = & [\![\texttt{if}\ be\ \texttt{then}\ \Omega\ \texttt{else skip}]\!] \\
f_2 & = & \Phi(f_1) \\
 & = & \lambda S \in States.([\![be]\!](S) \Rightarrow f_1^*([\![C]\!](S)) \mid [S]) \\
 & = & [\![\texttt{if}\ be\ \texttt{then}\ (C\ ;\ \texttt{if}\ be\ \texttt{then}\ \Omega\ \texttt{else skip})\ \texttt{else skip}]\!] \\
f_3 & = & \Phi(f_2) \\
 & = & \lambda S \in States.([\![be]\!](S) \Rightarrow f_2^*([\![C]\!](S)) \mid [S]) \\
 & = & [\![\texttt{if}\ be\ \texttt{then}\ (C\ ;\ \texttt{if}\ be\ \texttt{then}\ (C\ ;\ \texttt{if}\ be\ \texttt{then}\ \Omega\ \texttt{else skip})\ \texttt{else skip})\ \texttt{else skip}]\!] \\
f_4 & = & \Phi(f_3) \\
 & = & \text{and so on...}
\end{array}
$$

So the limit of this chain, which is the denotation of the `while` command is (morally) equal to the denotation of its infinite unfolding in terms of `if` statements. Each of the f_i is the denotation of a finite approximation to the `while` command which behaves like the `while` command for up to i iterations and then fails to terminate.

4.3 Equivalence of the Denotational and Operational Semantics of IMP

Theorem 23 *For all $ie \in Iexp$,$be \in Bexp$,$C \in Com$,$n \in \mathbb{Z}$,$b \in \mathbb{B}$ and $S, S' \in States$:*

1. *$ie, S \Rightarrow_I n$ if and only if $[\![ie]\!](S) = n$.*
2. *$be, S \Rightarrow_B b$ if and only if $[\![be]\!](S) = b$.*
3. *$C, S \Rightarrow_C S'$ if and only if $[\![C]\!](S) = [S']$.*

In other words, the denotational $[\![-]\!]$ functions are equal to the operationally defined functions Ieval,Beval and Ceval (where in the third case we regard Ceval as a function $Com \to [States \to States_\perp]$ using the bijection of Proposition 22).

Proof. The first two parts are just Proposition 21. For part 3., we have two directions to prove. The left-to-right direction is proved by rule induction for $\Rightarrow_C$, whilst the right-to-left direction is shown by structural induction on C.

For the left-to-right direction, we want to show that

$$\{(C, S, S') \mid [\![C]\!](S) = [S']\} \subseteq \mathit{Com} \times \mathit{States} \times \mathit{States}$$

is closed under the rules defining $\Rightarrow_C$, using parts 1. and 2. of the theorem for those rules whose hypotheses involve $\Rightarrow_B$ or $\Rightarrow_I$. We just deal with the case of rule $(\Rightarrow_C \cdot 7)$ and leave the other rules as Exercises.

Suppose that C is `while` be `do` C' and that

(a) $[\![be]\!](S) = \mathit{true}$

(b) $[\![C']\!](S) = [S']$

(c) $[\![C]\!](S') = [S'']$

for some states S, S', S''. We have to prove that $[\![C]\!](S) = [S'']$. Now, by the definition of the denotational semantics, $[\![C]\!] = \mathit{fix}(\Phi)$ where

$$\Phi \;=\; \lambda g.\lambda S.([\![be]\!](S) \Rightarrow g^*([\![C']\!](S)) \mid [S])$$

By the discussion in Section 4.1.8,

$$[\![C]\!] \;=\; \mathit{fix}(\Phi) \;=\; \Phi(\mathit{fix}(\Phi)) \;=\; \Phi([\![C]\!])$$

so that

$$\begin{array}{rcl}
[\![C]\!](S) & = & [\![be]\!](S) \Rightarrow [\![C]\!]^*([\![C']\!](S)) \mid [S] \\
& = & \mathit{true} \Rightarrow [\![C]\!]^*([S']) \mid [S] \quad \text{by (a) and (b)} \\
& = & [\![C]\!]^*([S']) \quad \text{by definition of } \Rightarrow| \\
& = & [\![C]\!](S') \quad \text{by definition of } (\cdot)^* \\
& = & [S''] \quad \text{by (c)}
\end{array}$$

as required.

For the right-to-left direction we use induction on the structure of C. As usual, the interesting case is when C is `while` be `do` C' and we consider this case in detail and leave the others as Exercises. In this case we want to show that

$$\forall S, S'.[\![C]\!](S) = [S'] \quad \text{implies} \quad C, S \Rightarrow_C S' \tag{4.2}$$

on the inductive assumption that

$$\forall S, S'.[\![C']\!](S) = [S'] \quad \text{implies} \quad C', S \Rightarrow_C S' \tag{4.3}$$

Now, (4.2) is equivalent to

$$[\![C]\!] \sqsubseteq \mathit{Ceval}(C) \quad \text{in } [\mathit{States} \to \mathit{States}_\perp] \tag{4.4}$$

where,

$$\mathit{Ceval}(C)(S) \stackrel{\text{def}}{=} \begin{cases} [S'] & \text{if } C, S \Rightarrow_C S' \text{ for some } S' \\ \perp & \text{otherwise} \end{cases}$$

But by the definition, $[\![C]\!] = \mathit{fix}(\Phi)$ and that means, by the discussion in Section 4.1.8, that we can deduce (4.4) if we can show $\mathit{Ceval}(C)$ to be a prefixed point of Φ, as it is then $\sqsupseteq$ the least prefixed point. In other words, we want to show

$$\Phi(\mathit{Ceval}(C)) \;\sqsubseteq\; \mathit{Ceval}(C) \tag{4.5}$$

i.e. that whenever $\Phi(\mathit{Ceval}(C))(S) \neq \bot$ then $\Phi(\mathit{Ceval}(C))(S) = \mathit{Ceval}(C)(S)$. But

$$\Phi(\mathit{Ceval}(C))(S) = ([\![be]\!](S) \Rightarrow (\mathit{Ceval}(C))^*([\![C']\!](S)) \mid [S])$$

so if $\Phi(\mathit{Ceval}(C))(S) \neq \bot$ then there are two possibilities:

1. Either $[\![be]\!](S) = \mathit{true}$ and $(\mathit{Ceval}(C))^*([\![C']\!](S)) \neq \bot$, or
2. $[\![be]\!](S) = \mathit{false}$.

We consider each case in turn:

1. In this case we have

$$\bot \;\neq\; (\mathit{Ceval}(C))^*([\![C']\!](S)) \;=\; \begin{cases} \mathit{Ceval}(C)(S') & \text{if } [\![C']\!](S) = [S'] \\ \bot & \text{otherwise} \end{cases}$$

   So we must have $[\![C']\!](S) = [S']$ and $\mathit{Ceval}(C)(S') = [S'']$ for some S, S''. Then by (4.3) and the definition of *Ceval* we have

$$C', S \Rightarrow_C S' \quad \text{and} \quad C, S' \Rightarrow_C S''$$

   We are assuming that $[\![be]\!] = \mathit{true}$ so that by Part 2. of the Theorem (which was part of Proposition 21) we have

$$be, S \Rightarrow_B \mathit{true}$$

   which taken with the two instances of $\Rightarrow_C$ above, allows us to apply $(\Rightarrow_C \cdot 7)$ to obtain $C, S \Rightarrow_C S''$ and hence

$$\Phi(\mathit{Ceval}(C))(S) \quad = \quad [S''] \quad = \quad \mathit{Ceval}(C)(S)$$

   as required.

2. In the case that $[\![be]\!](S) = \mathit{false}$, then by Part 2. of the Theorem, $be, S \Rightarrow_B \mathit{false}$ and so by applying rule $(\Rightarrow_C \cdot 6)$ we get

$$C, S \Rightarrow_C S$$

   and hence

$$\Phi(\mathit{Ceval}(C))(S) \quad = \quad [S] \quad = \quad \mathit{Ceval}(C)(S)$$

   as required.

So in either case, we get $\Phi(\mathit{Ceval}(C))(S) \neq \bot$ implies $\Phi(\mathit{Ceval}(C))(S) = \mathit{Ceval}(C)(S)$ for all $S \in \mathit{States}$ and thus we have established (4.5) and hence (4.2) as explained above. □

Theorem 23 immediately implies that the operational and denotational notions of equivalence coincide:

Corollary 24 *For any* $C_1, C_2 \in \mathit{Com}$

$$C_1 \approx C_2 \quad \Longleftrightarrow \quad [\![C_1]\!] = [\![C_2]\!]$$

□

4.3.1 Adequacy and full abstraction

We have been rather fortunate here – the correspondence between the denotational and operational semantics is very accurate. For more complicated languages, it is very difficult to achieve such a precise match. However, for many purposes a precise correspondence is not strictly necessary. For example, suppose we wish to use the denotational semantics to deduce that two commands are semantically equivalent. In other words, we plan to show that the two commands have the same denotation and deduce that they will behave the same operationally. For this to be a valid procedure, we only need to know the right-to-left direction of Corollary 24, viz.

$$[\![C_1]\!] = [\![C_2]\!] \quad \Rightarrow \quad C_1 \approx C_2$$

A semantics with this property (denotational equality implies operational equality) is said to be *adequate*. A semantics which satisfies both directions of Corollary 24 (so that operational equality implies denotational equality as well as vice-versa) is said to be *fully abstract*.

Of course, if our semantics is adequate but not fully abstract, we might *fail* to prove a true fact $C_1 \approx C_2$ using the denotational semantics. But for a good semantics this won't happen very often, and in any case, we don't expect to be able to *prove* all true equivalences, simply because of the incompleteness theorem (or the insolvability of the halting problem, according to taste). Note that an adequate but non-fully abstract semantics is no good at all for proving semantic *inequivalences*, but that showing inequivalences directly from the operational semantics is usually easy (for example, for IMP, to show that two commands are inequivalent we just have to give *one* state on which they behave differently).

So why is it hard to find fully abstract denotational semantics for many interesting languages? Part of the answer is that denotational models usually contain some points which are not the denotation of any program phrase. Now this does not in itself cause a failure of full abstraction: after all, there are continuous functions from *States* to $States_\perp$ which are not the denotation of any IMP command (Exercise: why? Find one.), but our semantics for IMP *is* fully abstract. The problem arises when two semantically equivalent phrases have denotations which are different, but are only different because of the presence of the extra elements in the semantics. For example, we might have two functions which behave the same on all arguments which are the denotation of a term in the language, but which give different results on some argument which is not the denotation of any term. Examining just how this occurs can give fairly deep insights into the structure of the language in question. For example, a straightforward denotational semantics for a little functional language turns out not to be fully abstract because all the functions definable in the language are *sequential* – they can be computed without any parallelism or time-slicing. The model contains functions which cannot be so computed, such as 'parallel or', which is the continuous function $por\colon \mathbb{B}_\perp \times \mathbb{B}_\perp \to \mathbb{B}_\perp$ defined by

$$por(x,y) \;=\; \begin{cases} [true] & \text{if } x = [true] \text{ or } y = [true] \\ [false] & \text{if } x = [false] \text{ and } y = [false] \\ \perp & \text{otherwise} \end{cases}$$

You should be able to see that to implement a function with this behaviour requires some parallelism, because on a sequential machine the function has to look at one of its arguments first, and if that fails to terminate, the application as a whole will fail to

terminate, even if the other argument would have returned [*true*]. And in fact it turns out that adding a parallel or constant to the language makes the simple semantics fully abstract. The alternative, refining the definitions of domains and continuous functions so as to get full abstraction for the sequential language is an extremely challenging problem (which has recently been solved, after a fashion).

4.3.2 Compositionality and congruence

The denotational semantics we have given for IMP has a very interesting property, which is that the meaning of any phrase is given in terms of the meaning of its subphrases (go back and look!). We call this property of the semantics *compositionality*, and it is a highly desirable feature of a semantics. We can use the fact that the denotational semantics is compositional to give a slick proof of the fact that semantic equivalence is a congruence (you should have already proved this directly from the operational semantics when doing the Exercises at the end of the last chapter):

Corollary 25 (Semantic congruence) *For any $C_1, C_2 \in Com$ and $C[\cdot]$ a 'command with a hole in it',*

$$C_1 \approx C_2 \quad \textit{implies} \quad C[C_1] \approx C[C_2]$$

Proof. Because $[\![-]\!]$ is compositional, it's obvious that $[\![C_1]\!] = [\![C_2]\!]$ implies that $[\![C[C_1]]\!] = [\![C[C_2]]\!]$ and the result then follows from Corollary 24 (just using the adequacy direction). □

4.4 Information, Continuity and Computability

We have given the semantics of IMP commands in terms of continuous functions between cpos. It seems worth trying to give some slightly more intuitive explanation of *why* cpos and continuous functions were chosen, and work, for this purpose.

Cpos and continuous functions are not *a priori* obviously the place to look for the meanings of programs. One's first thought would be to just take sets and functions. This doesn't quite work for a number of reasons:

1. Non-termination. If $[\![A]\!]$ were the set associated with a type A, and $[\![B]\!]$ that associated with a type B, then taking $[\![A \to B]\!]$ to be $([\![A]\!] \to [\![B]\!])$, the set of functions between $[\![A]\!]$ and $[\![B]\!]$ would not account for the possibility of non-termination. (You should know from Computation Theory that the possibility of non-termination is a central part of all Turing-powerful computational paradigms.)

2. Recursion. Looping or recursive language constructs such as recursive function definitions in ML or `while` loops in IMP naturally lead to the denotation of certain expressions being defined in a recursive way. If the denotation of a program expression e is a function $f : [\![A]\!] \to [\![B]\!]$ then this means defining f to be a solution to an equation $f = \Phi(f)$ where Φ is some function from $([\![A]\!] \to [\![B]\!])$ to itself. We simply cannot find solutions to arbitrary such equations if we allow *all* set-theoretic functions for Φ. We feel somehow that we shouldn't have to solve arbitrary equations, since there are vastly fewer *computable* functions and these are the only ones which give rise to equations which we absolutely *have* to be able to solve. Furthermore,

some equations may have more than one solution (e.g. $f = f$ has any function at all as a solution), and we need some way to pick the one of those solutions which corresponds to what the operational semantics actually gives.

3. Recursive domain equations. As well as recursively defined elements of domains, we also have to deal with situations where the domains themselves are recursively defined. IMP is too simple to require this, but it shows up in the semantics for ML's recursive datatypes or in the case of the untyped lambda calculus, which was the original reason for Dana Scott's introduction of domain theory in the late 60s. Roughly speaking, the argument goes as follows: in the untyped lambda calculus every term is a function which can be applied to any other term and return a term. So if D is the set representing the meanings of untyped lambda terms, we would have to have $D = (D \to D)$. In fact, we'd be content with $D \cong (D \to D)$ (replacing equality with isomorphism), but even this has only a trivial solution if we take $(D \to D)$ to be the set of all functions from the set D to itself. The problem is that $(D \to D)$ is always of strictly larger cardinality (size) than D whenever D has more than one element.[7] By adding some order structure to D, and *restricting* the meaning of $(D \to D)$ to those functions which respect that order structure it is possible to find solutions to the equation and hence to find a semantics for the untyped lambda calculus.

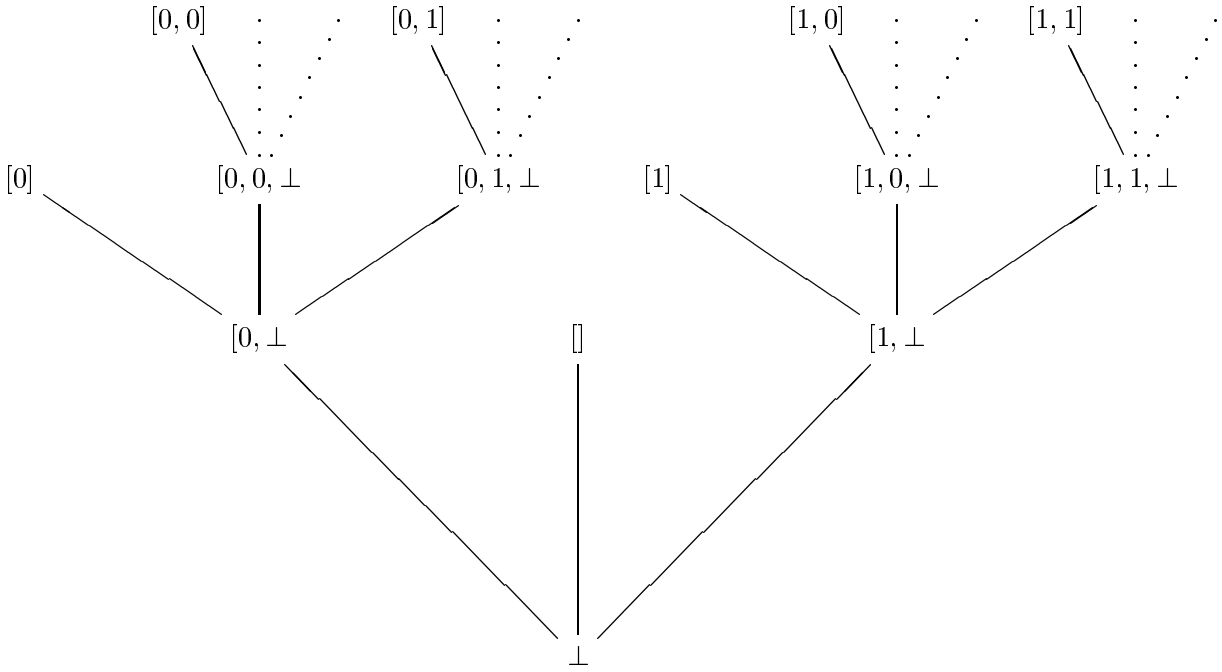
So the reasons for introducing all the technical machinery of order and continuity were initially very pragmatic – we just wanted something like a set, but sets wouldn't actually work because they had too many functions between them. So we added some structure and cut down the function space to functions that preserved the structure so that we could solve recursive domain equations (and recursive element equations). After the fact, however, we can develop an informal story which relates the order structure to an intuitive idea of information, and continuity to computability. This analogy between continuity and computability is not precise, but we can certainly argue informally that any computable function should be continuous.

Firstly, just consider non-termination. The idea of adding a new element to represent non-termination seems simple enough, but why not just take a set-theoretic union? Well, consider a function $f : \mathbb{N} \cup \{\bot\} \to \mathbb{N} \cup \{\bot\}$. If $f(\bot) = [n]$ then that means that when the argument of f fails to terminate, then f returns the natural number n. Now if f is supposed to be the meaning of a computer program, then this must mean that $f([m]) = [n]$ for all $m \in \mathbb{N}$, since f could not have decided what to do when given $\bot$ as input by looking at its input, seeing that it failed to terminate, and then returning $[n]$ – that would take for ever! So f must return $[n]$ for *any* input. So $\bot$ represents *less information* than $[m]$, and if the function is given more information as input, it must give more information as output, which we can express by putting an order on $\mathbb{N} \cup \{\bot\}$ to give $\mathbb{N}_\bot$, and restricting attention to monotone functions.

As another example, consider a computer program or system F which takes finite and infinite streams of 0s and 1s as input and returns a 0 or a 1. There are many such systems. One of them just returns a 0 without reading any input. Another returns 1 straight away. One of them reads one character from the input and returns that, whereas another negates the first character. One returns a 1 as soon as it has seen more than 42 1s in the input, one

[7] By a simple diagonalisation argument of the sort you should have met in Discrete Maths and in Computation Theory.

counts all the ones in the input and returns 0 or 1 according to whether the total is a prime number, and so on. There are some constraints on the possible behaviours, however. One of these is that if F outputs a value before it's seen all of the input, then it cannot retract that on the basis of any subsequent input. We can use $\bot$ as a 'don't know' element on the input and the output and there is then a natural information order imposed on them both. For the output side we get $\{0,1\}_\bot$, reading $\bot$ as 'F hasn't produced an answer yet' and 0 (resp. 1) as 'F has printed a 0 (resp. 1)'. The order on the input is more interesting and starts like this:



where, for example $\bot$ means we haven't seen any of the input yet, $[0,\bot$ means that we've seen a zero but we don't know what comes next and $[0]$ means that we've seen a zero and then an end-of-stream marker. Note that the information ordering is essentially a prefix ordering. You should be able to convince yourself that any implementable F has to be monotonic with respect to this ordering.

What about continuity? Take, for example, the increasing chain one obtains by aways taking the right-hand branch in the input domain. This is

$$\bot \sqsubseteq [1,\bot \sqsubseteq [1,1,\bot \sqsubseteq [1,1,1,\bot \sqsubseteq \cdots \sqsubseteq [1,1,1,\ldots]$$

where the limit is the infinite stream of 1s, which we can also write as 1^ω. Continuity imposes the extra restriction on F that it cannot return $\bot$ at all the finite stages and then suddenly return 0 or 1 at the infinite limit 1^ω. This is surely reasonable, since F can only *know* that it is being presented with an infinite stream of 1s after waiting an infinite amount of time.

This is not, of course, a truly compelling argument that computable maps are continuous, but it does give some idea of how we can rationalise the fact that domains and continuous maps suit the purposes of semantics. You should note that the other implication certainly doesn't hold – there are clearly many non-computable functions in $[\mathbb{N} \to \mathbb{N}_\bot]$, since *any* partial function $\mathbb{N} \rightharpoonup \mathbb{N}$ at all gives rise to such a continuous function. Continuity only really starts to cut things down at higher types.

4.5 Implementing the Denotational Semantics in ML

It is relatively easy to produce a rather imprecise translation of the clauses defining $[\![-]\!]$ into ML functions. We cannot, however, really reflect the subtleties of the distinctions we make between, for example, *States* and $States_\perp$ in the ML code, since all ML types already contain the possibility of non-termination, and one cannot in any case write programs which manipulate non-termination like any other value. Thus the best we can do is to write some ML code which has, morally, the *same* denotational semantics as IMP programs, rather than *being* that semantics. The ML code is closer to an alternative presentation of the semantics in which we do not make lifting explicit, and instead make the denotation of a command be a strict continuous function from $States_\perp$ to $States_\perp$ (see Exercise 17). It is virtually the same as the implementation of the big step evaluation semantics:

```
(* Denotational Semantics of IMP *)

(* denotei : IEXP -> (STATES -> int) *)
fun denotei ie (S:STATES) = case ie of
  N(n) => n
| Pvar(x) => lookup(x,S)
| Iop(iop,ie1,ie2) => let val n1 = denotei ie1 S
                          val n2 = denotei ie2 S
                      in
                          iopmeaning iop (n1,n2)
                      end;

(* denoteb : BEXP -> (STATES -> bool) *)
fun denoteb be (S:STATES) = case be of
  B(b) => b
| Bop(bop,ie1,ie2) => let val n1 = denotei ie1 S
                          val n2 = denotei ie2 S
                      in
                          bopmeaning bop (n1,n2)
                      end;

(* Fixpoint combinator *)
fun fix f x = f (fix f) x;

(* denotec : COM -> (STATES -> STATES) *)
fun denotec C (S:STATES) = case C of
  Skip => S
| Assign(x,ie) => let val n = denotei ie S
                  in update(S,x,n)
                  end
| Seq(C1,C2) => let val S' = denotec C1 S
                in denotec C2 S'
                end
| If(be,C1,C2) => if (denoteb be S)
                  then denotec C1 S
                  else denotec C2 S
```

```
| While(be,C1) =>
  let val phi = fn f => fn S'=> if (denoteb be S')
                                then f (denotec C1 S')
                                else S'
  in (fix phi) S
  end;
```

4.6 Exercises

1. Let X and Y be sets, regarded as discrete cpos. Show that a function $f: X_\perp \to Y_\perp$ is continuous if and only if one of the following holds:

   (a) f is *strict*. That is, $f(\perp) = \perp$.

   (b) f is *constant*. $\forall x \in X_\perp . f(x) = f(\perp)$.

2. Suppose that $\sqsubseteq$ is a partial order on a set X and that $f: X \to X$ is a monotone function. Show that $x_0 \in X$ is a fixed point of f if both the following two conditions hold:

   (a) $x_0 \sqsubseteq f(x_0)$

   (b) $\forall x \in X. (x \sqsubseteq f(x) \;\Rightarrow\; x \sqsubseteq x_0)$.

3. Suppose that D,E and F are cpos, and that $f: D \times E \to F$ is a function satisfying

   (a) For all $d \in D$, $\lambda y \in E. f(d, y)$ is a continuous function $E \to F$,

   (b) For all $e \in E$, $\lambda x \in D. f(x, e)$ is a continuous function $D \to F$.

   Is it the case that f is itself a continuous function from the product cpo $D \times E$ to F?

4. Show that the function *ev*: $[D \to E] \times D \to E$ of Section 4.1.5 is continuous.

5. Let D be a cpo with a least element. Show that the function *fix*: $[D \to D] \to D$ of Section 4.1.8 is continuous.

6. Let Ω be the cpo in the Examples at the end of Section 4.1.2 and 1 a one-element cpo. Show that the exponential cpo $[\Omega \to (1)_\perp]$ is in bijection with Ω itself.

7. Look again at Proposition 22. What is the the order relation on the set of partial functions $(X \rightharpoonup Y)$ which is induced from the order relation $\sqsubseteq$ on the cpo $[X \to Y_\perp]$ under the bijection I?

8. We say that two cpos D and E are *isomorphic*, and write $D \cong E$ if there are *continuous* functions $\phi: D \to E$ and $\psi: E \to D$ which are mutually inverse, so that $\phi \circ \psi$ is the identity function on E *and* $\psi \circ \phi$ is the identity on D. Prove or disprove each of the following statements:

   (a) For any cpos A and B, $A \times B \;\cong B \times A$.

   (b) For any cpos A and B, $(A \times B)_\perp \;\cong\; A_\perp \times B_\perp$.

   (c) For any cpos A, B and C, $A \times (B \times C) \;\cong\; (A \times B) \times C$.

(d) For any cpos A and B, $[A \to B]_\perp \cong [A \to B_\perp]$.

(e) For any cpos A, B and C, $[(A \times B) \to C] \cong [A \to [B \to C]]$.

(f) For any cpos A and B, $[A_\perp \to B] \cong B \times [A \to B]$.

9. Show that for any $d \in D$, if we define $P \subseteq D$ by $P = \{x \in D \mid x \sqsubseteq d\}$, then P is an inclusive subset of D.

10. Show that an arbitrary intersection of inclusive subsets of a cpo D is itself an inclusive subset of D. In other words, assume that for all $i \in I$, P_i is an inclusive subset of D and then prove that $\bigcap_{i \in I} P_i$ is an inclusive subset of D.

11. Show that the union of two inclusive subsets of D is inclusive. Deduce that any finite union of inclusive subsets is inclusive. Give an example to show that an *infinite* union of inclusive subsets need not be inclusive (hint: start by just thinking of the simplest example you can of a non-inclusive subset of a cpo).

12. Show that if $f: D \to E$ is a continuous function between two cpos, and P is an inclusive subset of E, then $f^{-1}(P)$ is an inclusive subset of D (where, of course, $f^{-1}(P) = \{d \in D \mid f(d) \in P\}$). So inclusive subsets are closed under inverse images of continuous functions. How about direct images? In other words, if P is an inclusive subset of D, is $f(P)$ always an inclusive subset of E?

13. Do the proof of Proposition 21, showing the equivalence of the denotational functions $[\![-]\!]$ with the operationally defined evaluation relations for integer and boolean expressions.

14. Complete the proof of Theorem 23 by

    (a) Showing that the set

    $$\{(C, S, S') \mid [\![C]\!](S) = [S']\} \subseteq \mathit{Com} \times \mathit{States} \times \mathit{States}$$

    is closed under the rules $(\Rightarrow_C \cdot 1)$ to $(\Rightarrow_C \cdot 6)$ of the evaluation semantics.

    (b) Completing the proof by induction on the structure of C that

    $$\forall S, S'.[\![C]\!](S) = [S'] \quad \text{implies} \quad C, S \Rightarrow_C S'$$

15. Let $\Phi: [\mathbb{N} \to \mathbb{N}_\perp] \to [\mathbb{N} \to \mathbb{N}_\perp]$ be the function which sends $g: \mathbb{N} \to \mathbb{N}_\perp$ to the function $\Phi(g): \mathbb{N} \to \mathbb{N}_\perp$ defined by

    $$\Phi(g)(n) \stackrel{\text{def}}{=} \begin{cases} [1] & \text{if } n = 0 \\ (\lambda m.[m \times n])^*(g(n-1)) & \text{otherwise} \end{cases}$$

    for all $n \in \mathbb{N}$. What is the function $\Phi(\lambda x \in \mathbb{N}.\perp)$? What is $\Phi^n(\lambda x \in \mathbb{N}.\perp)$? What is $\mathit{fix}(\Phi)$?

16. Think what would happen in the previous question if we were to replace $\mathbb{N}$ and $\mathbb{N}_\perp$ by $\mathbb{Z}$ and $\mathbb{Z}_\perp$. How many fixed points are there of the new version of Φ? Which is the least fixed point? Which one corresponds to the real behaviour of the usual definition of a very familar ML function?

17. We have made a 'modern' choice about how we should treat lifting in giving the denotational semantics of IMP. In particular, we chose to work with cpos which do not necessarily have a bottom, and to make the denotation of a command be a continuous function from *States* to $States_\perp$. This meant that we had to use the $(\cdot)^*$ operation to compose the denotation of two commands. There is an alternative presentation, which is the one used in much of the literature, in which we never use cpos without a bottom, and make the semantics of commands be *strict* continuous functions from $States_\perp$ to $States_\perp$. In this presentation the interpretation of sequencing is just ordinary functional composition. Work out the rest of the details of the semantics of IMP in this form. Compare it with the other semantics and with the ML implementation of the denotational semantics.

Chapter 5

Further Topics

We have now covered the semantics of IMP fairly thoroughly. The aim of this chapter is to give a brief, and rather informal, sketch of how we can give semantics to a couple of interesting extensions of IMP, and how Floyd-Hoare logic (that is, an axiomatic semantics) for IMP may be interpreted and proved sound using the denotational semantics we have already given. All the material in this chapter is non-examinable, in the sense that you will not be assumed to know it. Some of these topics have, however, arisen in previous years' questions, and in any case it is important to be aware that giving semantics to more sophisticated programming languages can be considerably more complex and subtle (not to mention interesting) than was the case for IMP.

5.1 Non-Determinism

IMP is a completely deterministic language: in a given state, the result of evaluating a particular expression is unique and the result of executing any command (either a final state or non-termination) is also unique. We proved this for the small-step semantics in Theorem 14, and it is also implicit in the fact that, for example, the denotation of a command is given as a *function* from $States$ to $States_\perp$. Non-determinism is interesting for a variety of reasons. Firstly, adding non-deterministic constructs to a language can, perhaps surprisingly, lead to clearer programs. This is because one can avoid having to overspecify some aspects of an algorithm. For example, a program which operates on two streams of input data might be expressed as non-deterministically choosing which stream to look at next if the order in which items are picked for processing is unimportant. Secondly, non-determinism is a natural consequence of concurrency (though there is more to concurrency than non-determinism). For example, suppose that we were to extend IMP with a command form $C_1 \parallel C_2$ which is supposed to execute C_1 and C_2 in parallel. What would be the effect of $(X := 1) \parallel (X := 2)$? Assuming that assignments are atomic, the result will be that X has the value 1 *or* the value 2, depending on the order in which the processes run; the programmer must generally assume that either order is possible.

Finally, non-determinism arises even in modelling deterministic systems, if we wish to abstract away from certain low-level details. One interesting example of this phenomenon occurs when describing static analyses, such as are often performed by optimising compilers to discover program properties which can be used to compile more efficient code. The analysis is sometimes done by computing an approximate semantics of the program in which it is assumed that, for example, a value of type `int` *might* have any integer value

at all. Thus semantic techniques designed for non-deterministic languages are useful in modelling the static analysis, even when the language being analysed is deterministic.

5.1.1 Transition semantics of non-determinism

We now consider extending IMP with a non-deterministic choice construct. The rules for forming commands (Figure 3.1) are extended with

$$\frac{C_1 \in Com \qquad C_2 \in Com}{C_1 \texttt{ or } C_2 \in Com}$$

and the intended behaviour of C_1 `or` C_2 is that it non-deterministically behaves either like C_1 or like C_2. Here's how to extend the small-step transition semantics (Figure 3.2) to cope with choice:

$$\frac{}{\langle C_1 \texttt{ or } C_2, S\rangle \rightarrow_C \langle C_1, S\rangle} \qquad \frac{}{\langle C_1 \texttt{ or } C_2, S\rangle \rightarrow_C \langle C_2, S\rangle}$$

These two axioms simply say that C_1 `or` C_2 *can* make a single step to become C_1, and that it also *can* make a transition to become C_2. Note that this means that the transition relation $\rightarrow_C$ on configurations now *has* to be a general relation, because a configuration can have more than one immediate successor. Previously, $\rightarrow_C$ was actually a partial function (for example, we exploited this fact when defining `cstep` in ML).

This apparently trivial extension of IMP is actually rather more interesting than it might at first appear. Previously, a given configuration (i.e. pair of a command and a state) could behave in just two ways - it could have a finite evaluation sequence, corresponding to termination, or it could have an infinite evaluation sequence, corresponding to non-termination. But with the addition of non-deterministic choice, the possibilities are richer – not only can a command have several possible finite evaluation sequences, but it can have a mixture of some finite and some infinite evaluation sequences. This leads to some choice in how to formulate the big-step and denotational semantics.

5.1.2 An evaluation semantics for non-determinism

The obvious way to give a big-step semantics to our extended language is to add the following two rules to those in Figure 3.3 with

$$\frac{C_1, S \Rightarrow_C S'}{C_1 \texttt{ or } C_2, S \Rightarrow_C S'} \qquad \frac{C_2, S \Rightarrow_C S'}{C_1 \texttt{ or } C_2, S \Rightarrow_C S'}$$

And indeed, given these two rules, there is still a correspondence between the small-step and the big-step semantics (Exercise):

Theorem 26 *When the syntax, transition semantics and evaluation semantics of IMP are extended as above, the equivalence of Theorem 16, viz.*

$$\langle C, S\rangle \rightarrow_C^* \langle \texttt{skip}, S'\rangle \qquad \textit{if and only if} \qquad C, S \Rightarrow_C S'$$

remains valid. □

But this is not quite the whole story. Think about how non-termination is treated in the two styles of operational semantics. In the transition semantics, we can explicitly express what it is for a configuration to lead to a non-terminating computation: there is an infinite sequence of one step-transitions

$$\langle C, S\rangle \to_C \langle C', S'\rangle \to_C \langle C'', S''\rangle \to_C \cdots$$

(which we can write $\langle C, S\rangle \to_C^\omega$). In the big-step evaluation semantics, by contrast, non-termination is simply reflected by the *absence* of a derivation of a terminating evaluation. Before we added non-determinism, this was perfectly adequate:

Theorem 27 *For deterministic IMP, for any command C and state S*

$$\langle C, S\rangle \to_C^\omega \qquad \textit{if and only if} \qquad \nexists S'.\, C, S \Rightarrow_C S'$$

□

But for non-deterministic IMP, the theorem above is false. For example, for any S

$$\langle (\texttt{skip}) \texttt{ or } (\texttt{while true do skip}), S\rangle \ \to_C^\omega$$

but also

$$(\texttt{skip}) \texttt{ or } (\texttt{while true do skip}), S \ \Rightarrow_C S$$

So the big-step semantics only really captures what we might think of as "positive" information: it just tells us the set of terminating behaviours of a configuration; thus it cannot distinguish a command which *always* terminates in a given state from one which *sometimes* terminates in that state and sometimes fails to terminate. Whether we care about this distinction depends on the use we are making of the semantics.

One way to make the big-step semantics more accurate is to add a "may-diverge" predicate on configurations $\Uparrow \subseteq (Com \times States)$, but it is not immediately clear how to define $\Uparrow$. It is not too hard to write down some plausible looking rules, such as

$$\frac{be, S \Rightarrow_B true \qquad C, S \Rightarrow_C S' \qquad \texttt{while}\ be\ \texttt{do}\ C, S' \Uparrow}{\texttt{while}\ be\ \texttt{do}\ C, S \Uparrow}$$

but it is rather harder to see what those rules are supposed to mean. They certainly don't constitute an inductive definition of $\Uparrow$, because if one tries to use them to derive the divergence of a particular command, one ends up trying to construct infinite derivations. For example, if we try to prove

$$\texttt{while true do skip} \Uparrow$$

the derivation must end with

$$\frac{\texttt{true}, S \Rightarrow_B true \qquad \texttt{skip}, S \Rightarrow_C S \qquad \texttt{while true do skip}, S \Uparrow}{\texttt{while true do skip}, S \Uparrow}$$

where the final assumption is the same as the conclusion. It turns out that we *can* make sense of the rules defining the divergence predicate if we understand them as a *co-inductive definition*. The notion of co-inductive definition is essentially dual to that of inductive definition (it is defined using a greatest, rather than a least, fixed point), but it would, unfortunately, take us too far beyond the scope of this course to investigate it in more detail.[1]

[1] A theory of operational semantics defined using a mixture of inductive and co-inductive evaluation and divergence relations has been developed by the Cousots, under the name GSOS$^\infty$. See, for example, P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proceedings of the ACM Conference on Princples of Programming Languages.* 1991.

5.1.3 Non-determinism and semantic equivalence

The combination of non-determinism and non-termination which we have in the extended version of IMP leads to a range of different notions of when two commmands should be considered equivalent. Let Ω be `while true do skip` and consider the following three commands:

1. $x := 1$
2. $(x := 1)$ `or` Ω
3. Ω

Which of these should be considered equivalent? There are several reasonable positions to take:

Plotkin says none of them are equivalent, because

1. *always* terminates
2. *can* terminate and *can* fail to terminate
3. *always* fails to terminate

This point of view is sometimes referred to as *erratic* non-determinism.

Hoare says 1 and 2 are equivalent, and different from 3, because they have the same set of possible observable results. This is referred to as *angelic*, or *relational* non-determinism; it is also the semantics behind *partial correctness* assertions (see Section 5.3).

Smyth says 2 and 3 are equivalent, and different from 1, because they can both fail to terminate and so we can guarantee nothing of either of them. This is called the *demonic* ('what can go wrong, will go wrong') view of non-determinism and is the semantics for *total correctness* assertions.

So the simple big-step step semantics which we gave above corresponds to angelic non-determinism. If, as before, we write $C_1 \approx C_2$ to mean

$$\forall S, S'.\ C_1, S \Rightarrow_C S' \quad \text{iff} \quad C_2, S \Rightarrow_C S'$$

then we have

$$(x := 1) \ \approx \ ((x := 1) \texttt{ or } \Omega) \ \not\approx \ \Omega$$

These different notions of when commands should be considered equivalent correspond to different notions of what we can *observe* of programs. The idea that different notions of observation yield different equivalences on program phrases arises in many areas of semantics, and you will meet it again in the Part II course on Concurrency and the Pi Calculus. When we give a denotational semantics to a language, we generally have to decide which notion of equivalence we wish to model since we would like two commands to be equivalent just when their denotations are equal (though this ideal is sometimes rather difficult to attain).

5.1.4 Denotational semantics for angelic non-determinism

It is fairly straightforward to give a denotational semantics to non-deterministic IMP which captures the angelic view of non-determinism. Recall that $\mathbb{P}(States)$, ordered by inclusion, is a complete partial order. We will take the denotation of a command to be a function from $States$ to $\mathbb{P}(States)$. (Since $States$ is discrete, all such functions are continuous.)

It is worth noting in passing that there are several equivalent ways of presenting the domain $[States \to \mathbb{P}(States)]$ which we could have used instead:

1. $[States \to \mathbb{P}(States)] \cong \mathbb{P}(States \times States)$ which is the cpo of *relations* between states. This is why the semantics we shall given is often called the *relational* semantics.

2. If we write $\mathbb{P}^+(States)$ for the cpo of *non-empty* subsets of $States$, then $[States \to \mathbb{P}(States)] \cong [States \to (\mathbb{P}^+(States))_\perp]$, since $(\mathbb{P}^+(States))_\perp \cong \mathbb{P}(States)$. This advantage of this view is that it makes explicit the fact that we are dealing with both non-determinism and non-termination at the same time.

3. We can also combine non-determinism and non-termination in the other order. If D is a cpo, we write $\mathbb{P}_H(D)$ for the cpo of non-empty, downwards-closed and limit-closed subsets of D ordered by inclusion, where S is downwards-closed if $x \sqsubseteq y \in S \Rightarrow x \in S$, and S is limit-closed if whenever $\langle x_n \rangle$ is a chain in D such that $\forall n.\, x_n \in S$ then $\bigsqcup x_n \in S$. Given this notation, we have $[States \to \mathbb{P}(States)] \cong [States \to \mathbb{P}_H(States_\perp)]$ since $\mathbb{P}(States) \cong \mathbb{P}_H(States_\perp)$. Observe also that $\mathbb{P}_H(States) = \mathbb{P}^+(States)$ (because $States$ is discrete), so we could have used $\mathbb{P}_H$ rather than $\mathbb{P}^+$ in 2. $\mathbb{P}_H(D)$ is called the *Hoare* (or *relational*) *powerdomain* of D.

It is an instructive Exercise to check that the assertions made above are correct, i.e. that the things claimed to be cpos really are cpos, and all the claimed isomorphisms really hold.

Before we can present the angelic denotational semantics of non-deterministic IMP, we need one new piece of notation. If X and Y are sets and $f\colon X \to \mathbb{P}(Y)$ is a function, then we write $f^\flat\colon \mathbb{P}(X) \to \mathbb{P}(Y)$ for the function which sends $A \subseteq X$ to $\bigcup_{a \in A} f(a)$. (Compare this with the definition of $(\cdot)^*$ given in Section 4.1.6.) It is a simple Exercise to check that $f^\flat$ is always continuous, if we regard $\mathbb{P}(X)$ and $\mathbb{P}(Y)$ as cpos.

The relational semantics of non-deterministic IMP is shown in Figure 5.1. Note the way in which we use singleton set formation $\{\cdot\}$ in the places where we used the lifting $[\cdot]$ operation in the semantics of deterministic IMP, and that the fixpoint in the semantics of `while`-loops is well-defined because $[States \to \mathbb{P}(States)]$ *is* a cpo with a bottom element, namely the constant emptyset function $\lambda S \in States.\emptyset$. In fact the angelic denotational semantics has much the same 'shape' as the deterministic denotational semantics of Chapter 4. The difference is that we have used the powerset operation $\mathbb{P}(\cdot)$ in place of the lifting operation $(\cdot)_\perp$. The precise correspondence is as follows:

Deterministic semantics	Non-deterministic semantics
$[States \to States_\perp]$	$[States \to \mathbb{P}(States)]$
$[S]$	$\{\,S\,\}$
$f^*\colon States_\perp \to States_\perp$	$f^\flat\colon \mathbb{P}(States) \to \mathbb{P}(States)$

The new language feature is the choice operation, which we model using the union operation on $\mathbb{P}(States)$. This semantics is both adequate and fully abstract for the angelic

$$
\begin{array}{rcl}
[\![skip]\!](S) & = & \{S\} \\
[\![x := ie]\!](S) & = & \{S[[\![ie]\!](S)/x]\} \\
[\![C_1 \,;\, C_2]\!](S) & = & [\![C_2]\!]^\flat([\![C_1]\!](S)) \\
[\![\texttt{if}\ be\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!](S) & = & [\![be]\!](S) \Rightarrow [\![C_1]\!](S) \mid [\![C_2]\!](S) \\
[\![\texttt{while}\ be\ \texttt{do}\ C]\!](S) & = & \mathit{fix}(\Phi) \\
\multicolumn{3}{l}{\text{where } \Phi\colon [\mathit{States} \to \mathbb{P}(\mathit{States})] \to [\mathit{States} \to \mathbb{P}(\mathit{States})] \text{ is defined by}} \\
\multicolumn{3}{l}{\Phi(f)(S) \;=\; [\![be]\!](S) \Rightarrow f^\flat([\![C]\!](S)) \mid \{S\}} \\
[\![C_1\ \texttt{or}\ C_2]\!](S) & = & ([\![C_1]\!](S)) \cup ([\![C_2]\!](S))
\end{array}
$$

Figure 5.1: Angelic Semantics of Non-Deterministic IMP

notion of equivalence (checking this is left as an Exercise for the reader, but I believe it's correct...):

Theorem 28 *For all non-deterministic IMP commands C_1 and C_2*

$$C_1 \cong C_2 \qquad \textit{iff} \qquad [\![C_1]\!] = [\![C_2]\!]$$

□

5.1.5 Erratic non-determinism and the Egli-Milner order

It is also possible to give a denotational semantics which models the more refined notion of equivalence given by the erratic view of non-determinism. The basic idea is that we want to define the meaning of a command to be a function from *States* to $\mathbb{P}_P(\mathit{States}_\perp)$, where $\mathbb{P}_P(\mathit{States}_\perp)$ is some cpo made up out of subsets of $\mathit{States}_\perp$ but which captures the erratic notion of when two sets of possible outcomes are equivalent. We can motivate the construction by considering an interesting example program:

$$x := 0 \,;\, a := 0 \,;\, \texttt{while}\ a = 0\ \texttt{do}\ ((x := x + 1)\ \texttt{or}\ (a := 1))$$

If we write the state as the pair $(S(x), S(a))$, the possible execution sequences of this command look like this:

$$
\begin{array}{ccccccc}
(0,0) & \to & (1,0) & \to & (2,0) & \to & \cdots \\
\downarrow & & \downarrow & & \downarrow & & \\
(0,1) & & (1,1) & & (2,1) & & \cdots
\end{array}
$$

So the possible behaviours are to terminate with $S(x) = n$ and $S(a) = 1$ for any $n \in \mathbb{N}$ *or* to fail to terminate. The fact that non-termination must be a possibility can be seen as a consequence of König's Lemma – any infinite, finitely branching tree has an infinite path (IA Discrete Maths). What does this have to do with modelling erratic non-determinism? It tells us that any *infinite* set in $\mathbb{P}_P(\mathit{States}_\perp)$ should also contain $\perp$. So we take the

underlying set of $\mathbb{P}_P(States_\perp)$ to be all non-empty subsets of $States_\perp$ which are either finite or contain $\perp$.

What order relation should we put on this collection of subsets in order to get a cpo? We've got *two* order relations to combine: the $\sqsubseteq$ ordering on $States_\perp$ and the $\subseteq$ ordering on $\mathbb{P}(States_\perp)$. The natural way to combine these is to say that we move up in the order from a subset A if every element of A is increased, possibly to a *set* of larger elements. The resulting order is called the *Egli-Milner* order $\sqsubseteq_{EM}$. More formally, $A \sqsubseteq_{EM} B$ iff for all $x \in A$ there exists a non-empty subset B_x of B such that

1. $\forall y \in B_x.\, x \sqsubseteq y$
2. $B = \bigcup_{x \in A} B_x$

Equivalently:

$$A \sqsubseteq_{EM} B \quad \text{iff} \quad \begin{array}{ll} 1. & \forall x \in A. \exists y \in B. x \sqsubseteq y \\ \text{and} \quad 2. & \forall y \in B. \exists x \in A. x \sqsubseteq y \end{array}$$

This order relation turns out to be closely related to the important notion of *bisimulation* which you will meet in the Part II Concurrency course. The cpo $(\mathbb{P}_P(States_\perp), \sqsubseteq_{EM})$ which is constructed by taking non-empty subsets of $States_\perp$ which are either finite or contain $\perp$, together with the Egli-Milner order, is called the *Plotkin powerdomain* of $States_\perp$. You should check that this really is a cpo and may like to try drawing (a finite part of!) its Hasse diagram.

By taking $[\![C]\!]$ to be in $[States \to \mathbb{P}_P(States_\perp)]$, we can define a denotational semantics for non-deterministic IMP which models the equivalence on commands given by the erratic view of non-determinism. We omit the details of this semantics, but its definition looks much like that which we gave for angelic non-determinism, though using a different powerdomain.

5.2 Jumps and Continuations

Most imperative programming languages have some constructs which allow non-local changes in the flow of control. This can mean anything from a completely unrestricted `goto` command to more structured operations such as exceptions, `break` commands for exiting loop bodies, an `abort` command which terminates execution immediately, or even more sophisticated control constructs such as built-in backtracking. You should have already seen (in Exercise 16 on page 42) how, with a little bit of ingenuity, one can extend an operational semantics to describe some constructs of this kind. But how can we give a *denotational* semantics to jumps? After all, mathematical functions just capture an input/output relation, and all the different classes of jump we have just mentioned seem, at least at first sight, to involve some notion of 'program point'.

There is a general technique for giving semantics to non-local control operations, which is due, independently, to Christopher Wadsworth and Lockwood Morris, following an idea of Mazurkiewicz published in 1970. The simplest form of the idea is that the meaning of a command should be a (curried) function of *two* arguments. One argument is the usual state, but the other is a function which represents what is to be done with the final state *after* the command has terminated in order to give the result of the whole program. This function is called a *continuation*. Notice that the idea of continuations is inherently higher-order: we rely on being able to pass functions as arguments in the denotational semantics, even if the language we are modelling does not contain explicit higher-order features. The

idea of continuation is an important one, which has now spread far beyond the semantics community. Continuation passing is a widely-used functional programming technique and is also at the heart of many modern compilers, such as that for Standard ML of New Jersey. Indeed, SML/NJ even extends the SML language with explicit support for continuation-based programming, in the form of the *call-with-current-continuation* (`callcc`) primitive. Continuations are also a topic of some exciting current research, as they appear to arise naturally in the context of extracting programs from proofs in classical logic.

5.2.1 Continuation semantics of IMP

We will demonstrate the use of continuations by first considering how to give a continuation based semantics to IMP with no extensions. We fix some set A of *answers*, which will be the set from which the results of whole programs will be drawn. It doesn't much matter exactly what A is, and you can, if you like, simply take A to be *States*, so that the final result of a program is just the final state in which it terminates. Now, define the cpo *Cont* of continuations to be $[States \to A_\perp]$, so that a continuation is a function which for each state yields either a final answer or non-termination. The continuation semantics gives the meaning of each command as a function taking a continuation and a state and returning an answer (or bottom):

$$[\![C]\!] : (Cont \to (States \to A_\perp))$$

Note that this is equivalent to either of

$$[\![C]\!] : Cont \to Cont \quad \text{or} \quad [\![C]\!] : (States \to A_\perp) \to (States \to A_\perp).$$

Now, think about the denotation of the `skip` command in this form. It will take a continuation $k \in [States \to A_\perp]$ and a state $S \in States$, and it has to return an answer. Obviously, the only thing it can do to produce an answer is to apply k to S. So

$$[\![\texttt{skip}]\!] \quad = \quad \lambda k \in Cont. \lambda S \in States.\ k(S)$$

and this is indeed intuitively the right thing to do, as k is supposed to be what is done to the state resulting from executing `skip` in the state S in order to return an answer, and the state after executing `skip` in state S is just S.

The interesting case of the continuation semantics is that for sequential composition of commands. Here it helps to think first about what $([\![C]\!]\ k)$ means for C a command and k a continuation. This is a partial application, of type $States \to A_\perp$. It is a function which takes a state as argument and then returns the answer you get from *first* running C in that state and *then* applying k to the state in which C terminates. But $States \to A_\perp$ is also the type of continuations, so $([\![C]\!]\ k)$ is itself a continuation – it's a 'what to do do next' which comprises first doing whatever C does and then doing whatever k does. How does that tell us what $[\![C_1\ ;\ C_2]\!]\ k\ S$ should be? The idea here is that we want to execute C_1in state S, and then execute C_2 and finally apply the continuation to the state that results. In other words, we just want to run C_1 in the state S with the continuation $([\![C_2]\!]\ k)$! Thus

$$[\![C_1\ ;\ C_2]\!] \quad = \quad \lambda k \in Cont. \lambda S \in States.\ [\![C_1]\!]\ ([\![C_2]\!]\ k)\ S$$

This is the *pons asinorum* of continuations; once you have crossed it, the rest follows fairly easily. The full continuation-passing semantics of IMP is shown in Figure 5.2. Note that

$$
\begin{array}{rcl}
[\![\texttt{skip}]\!]\ k\ S & = & k(S) \\
[\![x := ie]\!]\ k\ S & = & k(S[[\![ie]\!](S)/x]) \\
[\![C_1\ ;\ C_2]\!]\ k\ S & = & [\![C_1]\!]\ ([\![C_2]\!]\ k)\ S \\
[\![\texttt{if}\ be\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]\ k\ S & = & [\![be]\!](S)\ \Rightarrow\ [\![C_1]\!]\ k\ S \mid [\![C_2]\!]\ k\ S \\
[\![\texttt{while}\ be\ \texttt{do}\ C]\!] & = & \mathit{fix}(\Phi)
\end{array}
$$

where $\Phi\colon (\mathit{Cont} \to \mathit{Cont}) \to (\mathit{Cont} \to \mathit{Cont})$ is given by

$$\Phi\ =\ \lambda f\colon \mathit{Cont} \to \mathit{Cont}.\lambda k\colon \mathit{Cont}.\lambda S\colon \mathit{States}.[\![be]\!](S)\ \Rightarrow\ [\![C]\!]\ (f\ k)\ S\ \mid\ k(S)$$

Figure 5.2: Continuation Semantics of IMP

to calculate the result of a whole program, we now have to supply an initial state *and* an initial continuation k_0. The initial continuation might simply be the lift of the identity function ($k_0\ =\ \lambda S \in \mathit{States}.[S]$), if we take A to be *States*, or it might extract the value of some distinguished result variable r (so $k_0\ =\ \lambda S \in \mathit{States}.[S(r)]$), if we took $A = \mathbb{N}$.

This, you may be thinking, is all very well, but we already had a perfectly satisfactory semantics for IMP. But now assume that the set A contains a distinguished error value (so perhaps $A = \mathit{States} \cup \{\mathit{Error}\}$), and we wish to add a new command `abort` which, when executed, will immediately terminate the program and return this error value. It is absolutely trivial to add this to our continuation semantics:

$$[\![\texttt{abort}]\!]\ k\ S \quad = \quad [\mathit{Error}]$$

so the `abort` command simply throws away the continuation which it is given (i.e. it discards 'the rest of the program'), and returns (the lift of) the error value. It is sometimes said that the most powerful thing a command can do with a continuation is to ignore it.

5.2.2 Continuation semantics of IMP-with-exits

Now let's consider a more interesting example, which is a slight variant of the IMP-with-exits language which was introduced in Exercise 16 on page 42. We add two new command forms: `exit` and (C_1 `orelse` C_2) to the syntax of IMP. The intended behaviour of (C_1 `orelse` C_2) is that it executes exactly like C_1 unless C_1 hits an `exit` command, in which case further execution of C_1 is abandoned and C_2 is executed starting in the state at which C_1 encountered the `exit`. If C_1 does not encounter an `exit` then C_2 is ignored. An `exit` command without an enclosing `orelse` behaves like `abort`.

We can give a continuation semantics to this language by making the denotation of each command be a function which takes *two* continuations as well as a state, and returns an answer. The intuitive idea is that the first continuation is the ordinary default continuation which is to be applied if the command terminates normally ('success'), and the second continuation is the continuation to be applied if the command encounters an `exit` ('failure'). Thus

$$[\![C]\!] : (\mathit{Cont} \to (\mathit{Cont} \to (\mathit{States} \to A_\perp)))$$

This semantics is shown in Figure 5.3.

The interesting point here is the symmetry between the pair `skip` and '`;`' and the pair `exit` and `orelse`. The `skip` command simply succeeds, so it applies the success continuation to the current state. The composite $C_1\,;C_2$ behaves as C_1 with success continuation

$$
\begin{array}{rcl}
[\![\texttt{skip}]\!]\ k_1\ k_2\ S & = & k_1(S) \\
[\![C_1\ ;\ C_2]\!]\ k_1\ k_2\ S & = & [\![C_1]\!]\ ([\![C_2]\!]\ k_1\ k_2)\ k_2\ S \\
[\![\texttt{exit}]\!]\ k_1\ k_2\ S & = & k_2(S) \\
[\![C_1\ \texttt{orelse}\ C_2]\!]\ k_1\ k_2\ S & = & [\![C_1]\!]\ k_1\ ([\![C_2]\!]\ k_1\ k_2)\ S \\
[\![\texttt{abort}]\!]\ k_1\ k_2\ S & = & [\mathit{Err}] \\
[\![\texttt{if}\ be\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]\ k_1\ k_2\ S & = & [\![be]\!](S)\ \Rightarrow\ [\![C_1]\!]\ k_1\ k_2\ S \mid [\![C_2]\!]\ k_1\ k_2\ S \\
[\![\texttt{while}\ be\ \texttt{do}\ C]\!] & = & \mathit{fix}(\Phi)
\end{array}
$$

where Φ: $(\mathit{Cont} \to (\mathit{Cont} \to \mathit{Cont})) \to (\mathit{Cont} \to (\mathit{Cont} \to \mathit{Cont}))$ is given by

$\Phi\ =\ \lambda f.\lambda k_1.\lambda k_2.\lambda S.[\![be]\!](S)\ \Rightarrow\ [\![C]\!]\ (f\ k_1\ k_2)\ k_2\ S\ \mid\ k_1(S)$

Figure 5.3: Continuation Semantics of IMP-with-exits

($[\![C_2]\!]\ k_1\ k_2$) and failure continuation k_2. Thus, if C_1 succeeds it will subsequently do C_2 and then k_1 or k_2 according to whether or not C_2 succeeds. If C_1 fails, however, C_2 is ignored and the failure continuation k_2 is invoked.

The `exit` command simply fails, so it applies the failure continuation to the current state. The command C_1 `orelse` C_2 behaves as C_1 with success continuation k_1 and failure continuation ($[\![C_2]\!]\ k_1\ k_2$). Thus, if C_1 succeeds, C_2 will be ignored and the success continuation k_1 will be invoked. If C_1 fails, then C_2 will be executed, followed by k_1 or k_2 depending on whether or not C_2 succeeds.

To ensure that an `exit` without an enclosing `orelse` should behave like `abort`, we simply have to make the initial failure continuation which is supplied to an entire program be $k_f = \lambda S \in \mathit{States}.[\mathit{Err}]$.

5.2.3 An ML implementation of IMP-with-exits

Finally, here's a fairly direct implementation of the semantics of IMP-with-exits as ML code. This constitutes a complete working interpreter for the language. (You may like to extend the IMP parser functions to deal with the extended language.) Compare this code with the mathematical semantics in Figure 5.3 and see how well they correspond.

```
datatype IOP = Plus | Times | Minus;
datatype IEXP = N of int | Pvar of string | Iop of IOP*IEXP*IEXP;

datatype BOP = Equal | Greater;
datatype BEXP = B of bool | Bop of BOP*IEXP*IEXP;

datatype COM = Skip | Assign of string*IEXP | Seq of COM*COM |
               If of BEXP*COM*COM | While of BEXP*COM |
               Abort | Exit | Orelse of COM*COM;

(* ----------------------
   States
   ----------------------
*)
 type STATES = (string*int) list;
```

```
 (* lookup : string*STATES -> int *)
 exception Lookup;
 fun lookup(x,[]) = raise Lookup
  |  lookup(x,(y,v)::pairs) = if x=y then v else lookup(x,pairs);

 (* update : STATES*string*int -> STATES *)
 fun update (S,x,n) = case S of
      [] => [(x,n)]
   | ((y,v)::pairs) => if x=y then (x,n)::pairs
                              else (y,v)::(update (pairs,x,n));

(* iopmeaning : IOP -> ((int*int)->int) *)
fun iopmeaning iop (x:int,y:int) = case iop of
  Plus => x+y
| Times => x*y
| Minus => x-y;

(* bopmeaning : BOP -> ((int*int)->bool) *)
fun bopmeaning bop (x:int,y:int) = case bop of
  Equal => x=y
| Greater => x>y;


(* types of answers and continuations *)
datatype A = OK of int | Error;
type CONT =  STATES -> A;

(* initial state - everything is undefined *)
val (S:STATES) = [];

(* initial continuation returns the value of the variable r *)
val (k:CONT) = fn S => OK(lookup("r",S));

(* error continuation *)
val (ek:CONT) = fn S => Error;

(* denotei : IEXP -> (STATES -> int) *)
fun denotei ie (S:STATES) = case ie of
  N(n) => n
| Pvar(x) => lookup(x,S)
| Iop(iop,ie1,ie2) => let val n1 = denotei ie1 S
                          val n2 = denotei ie2 S
                      in
                          iopmeaning iop (n1,n2)
                      end;

(* denoteb : BEXP -> (STATES -> bool) *)
fun denoteb be (S:STATES) = case be of
```

```
  B(b) => b
| Bop(bop,ie1,ie2) => let val n1 = denotei ie1 S
                          val n2 = denotei ie2 S
                      in
                          bopmeaning bop (n1,n2)
                      end;

(* denotec : COM -> CONT -> CONT -> STATES -> A *)
fun denotec C (k1:CONT) (k2:CONT) (S:STATES) = case C of
  Skip => k1 S
| Assign(x,ie) => let val n = denotei ie S
                  in k1 (update(S,x,n))
                  end
| Seq(C1,C2) => denotec C1 (denotec C2 k1 k2) k2 S
| If(be,C1,C2) => if (denoteb be S)
                  then denotec C1 k1 k2 S
                  else denotec C2 k1 k2 S
| While(be,C1) => if (denoteb be S)
                  then denotec C1 (denotec C k1 k2) k2 S
                  else k1 S
| Abort => Error
| Exit => k2 S
| Orelse(C1,C2) => denotec C1 k1 (denotec C2 k1 k2) S;

(* runc runs a command with initial state and continuations *)
fun runc c = denotec c k ek S;
```

Here are a couple of simple examples of this implementation of IMP-with-exits in use:

```
- (* define c1 to be r:=0; (skip orelse r:=1)  *)
= val c1 = Seq(Assign("r",N 0), Orelse(Skip,Assign("r",N 1)));
> val v1 = ... : COM
- runc c1;
> OK 0 : A

- (* define c2 to be r:=0; (exit orelse r:=1)  *)
= val c2 = Seq(Assign("r",N 0), Orelse(Exit,Assign("r",N 1)));
> val c2 = ... : COM
- runc c2;
> OK 1 : A

- (* define c3 to be
=    r:=0;
=    while true do
=     (r:=r+1;
=      if r>7 then exit else skip))
=    orelse skip
=   to demonstrate breaking out of an otherwise infinite loop
= *)
```

```
= val c3 = Seq(Assign("r",N 0), Orelse(While(B true,
= Seq(Assign("r",Iop(Plus,Pvar "r",N 1)),
= If(Bop(Greater,Pvar "r",N 7),Exit,Skip))),Skip));
> val c3 = ... : COM
- runc c3;
> OK 8 : A
```

5.3 Axiomatic Semantics of IMP

This section contains a brief account of how the Floyd-Hoare rules for proving properties of IMP programs may be justified in terms of the denotational semantics. It is not completely detailed and rigorous, but it should give a good idea of how denotational semantics can be used to justify reasoning principles for program verification. You will learn a lot more about Floyd-Hoare logic in the Part II course Specification and Verification.

5.3.1 Partial Correctness Assertions

The general form of a partial correctness statement is

$$\{P\}\, C\, \{Q\}$$

which means 'if one executes the command C starting in a state which satisfies P, then if the command terminates it will do so in a state satisfying Q'. A typical example of a valid partial correctness statement about an IMP program would be the following, asserting that a program to calculate greatest common divisors is correct:

$$\begin{array}{l}\{X = x \wedge Y = y \wedge 1 \leq x \wedge 1 \leq y\} \\ \texttt{while}\ X \neq Y\ \texttt{do}\ (\texttt{if}\ X \leq Y\ \texttt{then}\ Y := Y - X\ \texttt{else}\ X := X - Y) \\ \{X = gcd(x, y)\}\end{array}$$

We could also develop a theory of *total* correctness statements of the form $[P]\ C\ [Q]$, meaning 'if the command C is started in a state satisfying P then it will terminate in a state satisfying Q', but we will not do so here.

We first have to introduce a language in which to formulate assertions about states. These will be defined in terms of an auxiliary set of integer variables *Ivar*. We will now use lower-case letters for elements of *Ivar* and upper-case for program variables (elements of *Pvar*). The set *Aexp* of arithmetic expressions is defined inductively by the following rules:

$$\frac{n \in \mathbb{Z}}{n \in Aexp} \qquad \frac{X \in Pvar}{X \in Aexp} \qquad \frac{i \in Ivar}{i \in Aexp}$$

$$\frac{a_1 \in Aexp \quad a_2 \in Aexp}{a_1 + a_2 \in Aexp} \qquad \frac{a_1 \in Aexp \quad a_2 \in Aexp}{a_1 - a_2 \in Aexp} \qquad \frac{a_1 \in Aexp \quad a_2 \in Aexp}{a_1 \times a_2 \in Aexp}$$

It is important that the set of integer expressions *Iexp* of IMP is contained within the set *Aexp*, but this is easily seen to be true.

We can now define the set of assertions *Assn* in terms of these arithmetic expressions. These assertions are made up of logical combinations of atomic assertions about arithmetic

expressions:

$$\frac{}{true \in Assn} \qquad \frac{}{false \in Assn} \qquad \frac{a_1 \in Aexp \qquad a_2 \in Aexp}{a_1 \leq a_2 \in Assn}$$

$$\frac{A_1 \in Assn \qquad A_2 \in Assn}{A_1 \wedge A_2 \in Assn} \qquad \frac{A \in Assn}{\neg A \in Assn} \qquad \frac{i \in Ivar \qquad A \in Assn}{\forall i.A \in Assn}$$

We will feel free to use other logical connectives in assertions, regarding them as syntactic sugar for combinations of the basic ones given above. For example, if $a_1, a_2 \in Aexp$ then $(a_1 = a_2) \in Assn \stackrel{\text{def}}{=} (a_1 \leq a_2) \wedge (a_2 \leq a_1)$. Similarly, if $A \in Assn$ and $i \in Ivar$ then $(\exists i.A) \in Assn \stackrel{\text{def}}{=} \neg(\forall i.\neg A)$. Note that $Bexp$, the set of boolean expressions in our programming language, is a subset of $Assn$ (modulo some syntactic sugar).

This small language of assertions is rich enough to code up a very wide range of predicates. As an Exercise, you might like to try expressing $X = gcd(x, y)$ in $Assn$.

If $S \in States$ and $A \in Assn$, we now want to define a notion of S *satisfying* the assertion A. This depends on knowing the meaning of arithmetic expressions, which in turn will depend on some assignment of integer values for all the integer variables *and* all the program variables. So, let an *interpretation* $I \in Interp$ be a function from $Ivar$ to $\mathbb{Z}$, and we can then define

$$[\![-]\!] : Aexp \to (Interp \to (States \to \mathbb{Z}))$$

as follows

$$\begin{aligned}
&[\![n]\!](I)(S) = n \\
&[\![X]\!](I)(S) = S(X) \\
&[\![i]\!](I)(S) = I(i) \\
&[\![a_1 + a_2]\!](I)(S) = [\![a_1]\!](I)(S) + [\![a_2]\!](I)(S) \\
&[\![a_1 - a_2]\!](I)(S) = [\![a_1]\!](I)(S) - [\![a_2]\!](I)(S) \\
&[\![a_1 \times a_2]\!](I)(S) = [\![a_1]\!](I)(S) \times [\![a_2]\!](I)(S)
\end{aligned}$$

Using this, we can then define when state $S \in States$ satisfies an assertion A under an interpretation I, which we write $S \models^I A$, by induction on the structure of A as follows:

$$\frac{}{S \models^I true} \qquad \frac{[\![a_0]\!](I)(S) \;\leq\; [\![a_1]\!](I)(S)}{S \models^I (a_0 \leq a_1)}$$

$$\frac{S \models^I A_1 \qquad S \models^I A_2}{S \models^I A_1 \wedge A_2} \qquad \frac{S \not\models^I A}{S \models^I \neg A}$$

$$\frac{\forall n \in \mathbb{Z}.\, S \models^{I[n/i]} A}{S \models^I \forall i.A}$$

And we can extend the notion of satisfaction to elements of $States_\perp$ by letting the undefined state satisfy every assertion. We will feel free to overload the use of the $\models$ notation to refer to elements of $States$ or of $States_\perp$ without comment.

$$\frac{S \models^I A}{[S] \models^I A} \qquad \frac{}{\perp \models^I A}$$

Now, if $A, B \in \mathit{Assn}$ and $C \in \mathit{Com}$ we can define the notion of when the partial correctness assertion

$$\{A\}\, C\, \{B\}$$

is *valid* by defining

$$\models \{A\}\, C\, \{B\}$$

to mean

$$\forall I \in \mathit{Interp}.\, \forall S \in \mathit{States}.\, (S \models^I A \;\Rightarrow\; [\![C]\!](S) \models^I B)$$

In other words, $\{A\}\, C\, \{B\}$ is a valid partial correctness assertion if for any interpretation I and for any state S which satisfies the assertion A, the denotation of C applied to state S is a state (possibly $\perp$) satisfying B. Note that we now have a completely formal notion of when a partial correctness assertion is valid, defined in terms of the denotational semantics. We will be able to use this to *prove* that a logic for deriving partial correctness assertions is sound (deduces only valid assertions), rather than just deciding that all the rules look intuitively plausible.

We will also use the notion of validity for assertions. If $A \in \mathit{Assn}$ then write $\models A$ to mean that for all $S \in \mathit{States}$ and for all $I \in \mathit{Interp}$, $S \models^I A$.

5.3.2 Hoare Logic

We now give some proof rules for deriving partial correctness statements about IMP programs. There is one rule for each command construct and one logical rule. The rule for assignment uses the notion of substituting an integer expression into an assertion which is defined in a fairly obvious way.

$$\frac{}{\{A\}\texttt{skip}\{A\}} \qquad \frac{}{\{A[ie/X]\}X := ie\{A\}}$$

$$\frac{\{A\}C_1\{A'\} \qquad \{A'\}C_2\{A''\}}{\{A\}C_1\,;\,C_2\{A''\}} \qquad \frac{\{A \wedge be\}C_1\{A'\} \qquad \{A \wedge \neg be\}C_2\{A'\}}{\{A\}\texttt{if}\ be\ \texttt{then}\ C_1\ \texttt{else}\ C_2\{A'\}}$$

$$\frac{\{A \wedge be\}C\{A\}}{\{A\}\texttt{while}\ be\ \texttt{do}\ C\{A \wedge \neg be\}} \qquad \frac{\models (A \Rightarrow A') \qquad \{A'\}C\{B'\} \qquad \models (B' \Rightarrow B)}{\{A\}C\{B\}}$$

We write $\vdash \{A\}C\{B\}$ when $\{A\}C\{B\}$ is derivable using the above rules.

5.3.3 Soundness of Hoare Logic

We aim to prove formally that the logic given in the previous section is *sound*, that is, all the theorems $\vdash \{A\}C\{B\}$ which can be derived in the logic are valid. The full proof of this fact relies on a couple of simple lemmas concerning substitution, both of which are proved by structural induction.

Lemma 29 *If $X \in \mathit{Pvar}$ and $a, a' \in \mathit{Aexp}$ then for all $I \in \mathit{Interp}$ and $S \in \mathit{States}$*

$$[\![a[a'/X]]\!](I)(S) \;=\; [\![a]\!](I)(S[[\![a']\!](I)(S)/X])$$

□

Lemma 30 *For any $I \in Interp, A \in Assn, X \in Pvar$, $ie \in Iexp$ and $S \in States$*

$$S \models^I A[ie/X] \iff S[[\![ie]\!](S)/X] \models^I B$$

□

We will also need to know that the meanings of assertions are inclusive predicates, which is an obvious consequence of the fact that $States_\perp$ is a flat cpo:

Lemma 31 *If $c : \mathbb{N} \to States_\perp$ is a chain in $States_\perp$, $I \in Interp$ and $A \in Assn$ then*

$$\left(\forall n \in \mathbb{N}.\ c_n \models^I A\right) \Rightarrow \bigsqcup_{n=0}^{\infty} c_n \models^I A$$

□

We can now formulate our main result

Theorem 32 (Soundness) *For any $A, B \in Assn$ and $C \in Com$, if $\vdash \{A\}C\{B\}$ then $\models \{A\}C\{B\}$.*

Proof. This follows by rule induction on the rules of Hoare logic. We have to show for each rule that if the hypotheses are valid then so is the conclusion. We will only consider the case of the rule for `while` commands here, and leave the other rules as Exercises. (We will not actually need Lemmas 29 and 30 for the case we consider here, but you will need them for some of the other cases.)

So assume that the hypothesis of the `while` rule is valid, i.e. that $\models \{A \wedge be\}C\{A\}$. We wish to prove that the conclusion of the rule is also valid, i.e. that $\models \{A\}\texttt{while}\, be\, \texttt{do}\, C\{A \wedge \neg be\}$. Recall that $[\![\texttt{while}\ be\ \texttt{do}\ C]\!] = fix\Phi$ where

$$\Phi \stackrel{\text{def}}{=} \lambda f \in [States \to States_\perp].\lambda S \in States.([\![be]\!](S) \Rightarrow f^*([\![C]\!](S)) \mid [S])$$

So we have to show that for any state S and interpretation I, if $S \models^I A$ then

$$\left(\bigsqcup_{n=0}^{\infty} \Phi^n(\lambda S' \in States.\perp)\right)(S) \models^I A \wedge \neg be$$

which, by the definition of lubs in function spaces is equivalent to

$$\left(\bigsqcup_{n=0}^{\infty} \Phi^n(\lambda S' \in States.\perp)(S)\right) \models^I A \wedge \neg be$$

We will show by mathematical induction that for all $n \in \mathbb{N}$ and for all $S \in States$, if $S \models^I A$ then

$$\Phi^n(\lambda S' \in States.\perp)(S) \models^I A \wedge \neg be$$

from which the result follows by Lemma 31. (This could have been presented as an example of fixpoint induction.)

For the base case of the induction, we just have

$$\Phi^0(\lambda S' \in States.\perp)(S) = \perp \models^I A \wedge \neg be$$

Now for the induction step, writing f_n for $\Phi^n(\lambda S' \in States.\bot)$ we need to show

$$\forall S \in States.\ S \models^I A \ \Rightarrow\ f_{n+1}(S) \models^I A \wedge \neg be$$

which means showing that if $S \models^I A$ then

$$(\llbracket be \rrbracket(S) \Rightarrow f_n^*(\llbracket C \rrbracket(S)) \mid [S]) \ \models^I \ A \wedge \neg be$$

Now there are two possibilities:

1. If $\llbracket be \rrbracket(S) = \mathit{false}$ then $f_{n+1}(S) = [S]$ and we have $[S] \models^I A$ and $[S] \models^I \neg be$ and hence $f_{n+1}(S) \models^I A \wedge \neg be$ as required.

2. If $\llbracket be \rrbracket(S) = \mathit{true}$ then $f_{n+1}(S) = f_n^*(\llbracket C \rrbracket(S))$. By our assumption about the hypothesis of the `while` rule being valid, we know that $\llbracket C \rrbracket(S) \models^I A$ because $S \models^I be$ and $S \models^I A$. Hence by induction, $f_n^*(\llbracket C \rrbracket(S)) \models A \wedge \neg be$ as required.

□

An obvious question to ask is whether *completeness*, which is the converse to soundness, holds – do the Hoare logic rules prove *all* valid partial correctness assertions? It turns out that they do, but with an important caveat. Notice that the logical rule

$$\frac{\models (A \Rightarrow A') \qquad \{A'\}C\{B'\} \qquad \models (B' \Rightarrow B)}{\{A\}C\{B\}}$$

is phrased in terms of the *validity* of the assertions $A \Rightarrow A'$ and $B' \Rightarrow B$, rather than their *provability*. If we actually want to use Hoare logic to prove things about programs then we have to give a proof system for assertions as well, and such a system can never be complete by Gödel's Incompleteness Theorem. Thus if we had access to an oracle which could magically decide the truth of statements of the form $\models A$, then we could prove all valid partial correctness assertions using Hoare logic. Since we do not have such an oracle, we have to make do with a logic for deriving statements $\vdash A$, and this prevents us from being able to prove all the valid partial correctness assertions. We can therefore say that Hoare logic is *relatively complete*.

For more realistic IMP-like programming languages, such as Algol, it has been shown that there is not even a relatively complete Hoare logic, so the usefulness of the concept of relative completeness is rather limited.

Appendix A

Semantic Equivalence Proofs as ML Functions

This appendix contains some very optional material which concerns the way in which constructive proofs of semantic equivalences can be seen in terms of functions mapping derivations in the evaluation semantics to other derivations.

We will give a brief outline of how this idea can be made concrete by defining ML functions which map derivations to derivations. The details are rather unpleasant, but it's worth at least noting that it *can* be done.

Firstly, we have to decide how to represent derivations in the evaluation semantics in ML. As we have mentioned in Chapter 2, the set of derivations is itself an inductively defined set and we shall code this set as an inductive ML datatype with one constructor for each rule in the semantics. (Actually, we'll need three datatypes, corresponding to the three evaluation relations.)

What should we store at each node in the tree? If we consider a rule like $(\Rightarrow_C \cdot 3)$, it's clear that we'll need to store the two subderivations which derive $C_1, S \Rightarrow_C S'$ and $C_2, S' \Rightarrow_C S''$, but what else? Actually, nothing else. This is because the conclusion of the rule, *viz.* $C_1; C_2\ , S \Rightarrow_C S''$ is completely determined by the conclusions of the two subderivations and the fact that we know we are applying rule $(\Rightarrow_C \cdot 3)$. For a rule like $(\Rightarrow_C \cdot 4)$ we will need to know the two subderivations *and* what the else-branch, C_2, of the if-statement is, since that does not appear in either of the subderivations. Applying similar reasoning to each rule, we get the following datatypes for evaluation derivations (you will need to refer to Figure 3.3 to have any chance of understanding this!):

```
datatype IDER = Ir1 of int*STATES | Ir2 of string*STATES |
                Ir3 of IOP*IDER*IDER;

datatype BDER = Br1 of bool*STATES | Br2 of BOP*IDER*IDER;

datatype CDER = Cr1 of STATES | Cr2 of string*IDER | Cr3 of CDER*CDER |
                  Cr4 of BDER*CDER*COM | Cr5 of BDER*CDER*COM |
                  Cr6 of BDER*COM | Cr7 of BDER*CDER*CDER;
```

For example, this derivation

$$\dfrac{\dfrac{\dfrac{}{\texttt{true},[] \Rightarrow_B \mathit{true}}(\Rightarrow_B \cdot 1) \qquad \dfrac{\dfrac{}{1,[] \Rightarrow_I 1}(\Rightarrow_I \cdot 1)}{\texttt{x} := 1,[] \Rightarrow_C [x=1]}(\Rightarrow_C \cdot 2)}{\texttt{if true then x} := 1 \texttt{ else x} := 2,[] \Rightarrow_C [x=1]}(\Rightarrow_C \cdot 4) \qquad \dfrac{\dfrac{}{0,[x=1] \Rightarrow_I 0}(\Rightarrow_I \cdot 1)}{\texttt{y} := 0,[x=1] \Rightarrow_C [x=1\ y=0]}(\Rightarrow_C \cdot 2)}{(\texttt{if true then x} := 1 \texttt{ else x} := 2)\texttt{ ; y} := 0,[] \Rightarrow_C [x=1\ y=0]}(\Rightarrow_C \cdot 3)$$

is coded as this element of the ML datatype `CDER`:

```
Cr3 (Cr4 (Br1 (true,[]),Cr2 ("x",Ir1 (1,[])),Assign ("x",N 2)),Cr2 ("y",
 Ir1 (0,[("x",1)])))
```

There is still a slight problem with our representation of derivations – there are many elements of the datatypes which do not correspond to valid derivations in the semantics. This is because there is no way to enforce the restrictions caused by the fact that subderivations of a given derivation usually have to agree with each other in some way. For example, in the case of $(\Rightarrow_C \cdot 3)$, the state at which C_1 ends up as the conclusion of the first subderivation has to be the same as the state in which C_2 is started in the conclusion of the second subderivation. We cannot express this in the ML datatype, so we will have to check that it is true using a function which traverses a putative derivation tree and checks that it is well-formed.[1] The following functions will extract the conclusions of a derivation and check that it is well-formed, raising the exception `BadDer` if it is not. Note that we have to use an auxiliary function to test whether two states, represented as lists, are equal since the order of the bindings in the two lists might be different.

```
fun forall [] p = true
  | forall (x::xs) p = (p x) andalso (forall xs p);

fun eqstate (S1,S2) = (forall S1 (fn x => x mem S2))
                      andalso
                      (forall S2 (fn x => x mem S1));
exception BadDer;

(* iconc : IDER -> IEXP*STATES*int *)
fun iconc d = case d of
  Ir1(n,S:STATES) => (N n,S,n)
| Ir2(x,S) => (Pvar x,S,lookup(x,S) handle Lookup => raise BadDer)
| Ir3(iop,d1,d2) => let val (ie1,S1,n1) = iconc d1
                        val (ie2,S2,n2) = iconc d2
                    in if eqstate(S1,S2)
                       then (Iop(iop,ie1,ie2),S1,iopmeaning iop (n1,n2))
```

[1] The problem is that ML's type system is not powerful enough to express these restrictions. There *are* much more powerful type theories which can cope with this sort of thing, and it is these more powerful systems which form the basis of many automated theorem provers. In such systems one really does give formal proofs by defining functions in a way which is not entirely unrelated to the rather rough-and-ready ML code we give here. The key idea is the 'propositions-as-types' analogy which is discussed (albeit not at a sufficiently advanced level to deal with the kind of proofs we're concerned with here) in Dr Pitts's Part II course on Types.

```
                    else raise BadDer
                 end;

(* bconc : BDER -> BEXP*STATES*bool *)
fun bconc d = case d of
  Br1(b,S:STATES) => (B b,S,b)
| Br2(bop,d1,d2) => let val (ie1,S1,n1) = iconc d1
                        val (ie2,S2,n2) = iconc d2
                    in if eqstate(S1,S2)
                       then (Bop(bop,ie1,ie2),S1,bopmeaning bop (n1,n2))
                       else raise BadDer
                    end;

(* cconc : CDER -> COM*STATES*STATES *)
fun cconc d = case d of
  Cr1(S:STATES) => (Skip,S,S)
| Cr2(x,d1) => let val (ie,S,n) = iconc d1
               in (Assign(x,ie),S,update(S,x,n))
               end
| Cr3(d1,d2) => let val (C1,S,S1) = cconc d1
                    val (C2,S2,S3) = cconc d2
                in if eqstate(S1,S2)
                   then (Seq(C1,C2),S,S3)
                   else raise BadDer
                end
| Cr4(bd,cd,C2) => let val (be,S1,b) = bconc bd
                       val (C1,S2,S3) = cconc cd
                   in if eqstate(S1,S2) andalso (b=true)
                      then (If(be,C1,C2),S1,S3)
                      else raise BadDer
                   end
| Cr5(bd,cd,C1) => let val (be,S1,b) = bconc bd
                       val (C2,S2,S3) = cconc cd
                   in if eqstate(S1,S2) andalso (b=false)
                      then (If(be,C1,C2),S1,S3)
                      else raise BadDer
                   end
| Cr6(bd,C) => let val (be,S,b) = bconc bd
               in if (b=false)
                  then (While(be,C),S,S)
                  else raise BadDer
               end
| Cr7(bd,d1,d2) => let val (be,S1,b) = bconc bd
                       val (C,S2,S3) = cconc d1
                       val (C',S4,S5) = cconc d2
                   in if (b=true) andalso
                         eqstate(S1,S2) andalso
                         eqstate(S3,S4) andalso
```

```
          (C' = While(be,C))
        then
          (C',S1,S5)
        else raise BadDer
    end;
```

We can use this to extract the conclusion of the derivation we gave earlier and to check that it is well-formed:

```
- cconc (Cr3 (Cr4 (Br1 (true,[]),Cr2 ("x",Ir1 (1,[])),Assign ("x",N 2)),
= Cr2 ("y",Ir1 (0,[("x",1)]))));
> (Seq (If (B(true),Assign("x",N 1),Assign ("x",N 2)),Assign ("y",N 0)),
   [],
   [("x",1),("y",0)]) : COM*STATES*STATES
```

So now what about coding proofs as ML functions? We will start by considering half of the proof of Proposition 17, the implication that says if

$$(\texttt{if}\ be\ \texttt{then}\ C\ \texttt{else}\ C')\,;C'', S \Rightarrow_C S' \tag{A.1}$$

then

$$\texttt{if}\ be\ \texttt{then}\ (C\,;C'')\ \texttt{else}\ (C\,;C''), S \Rightarrow_C S' \tag{A.2}$$

Our function to code the proof of this implication will take as input a derivation of A.1 and return a derivation of A.2. Any derivation of A.1 must end with an application of $(\Rightarrow_C \cdot 3)$ and we then consider cases according to the last rule used in the derivation of the first hypothesis of that application of $(\Rightarrow_C \cdot 3)$ to decide how to build a derivation of A.2. This is expressed by the following ML code:

```
(* ifseqproof : CDER -> CDER *)
fun ifseqproof (Cr3(d1,d2)) =
  let val (C3,_,_) = cconc d2 in
      case d1 of
         Cr4(bd,cd1,C2) => Cr4(bd,Cr3(cd1,d2),Seq(C2,C3))
       | Cr5(bd,cd2,C1) => Cr5(bd,Cr3(cd2,d2),Seq(C1,C3))
  end;
```

Note that `ifseqproof` doesn't make any attempt to check that its input is a valid derivation of an instance of A.1, though it will raise an uncaught exception (either `match` or `BadDer`) in most such cases. Now let's see `ifseqproof` in action by applying it to the ML term coding the derivation we gave earlier:

```
- ifseqproof (Cr3 (Cr4 (Br1 (true,[]),Cr2 ("x",Ir1 (1,[])),
= Assign ("x",N 2)),Cr2 ("y",Ir1 (0,[("x",1)]))));
> Cr4 (Br1  (true,[]),Cr3 (Cr2 ("x",Ir1 (1,[])),Cr2 ("y",Ir1
  (0,[("x",1)]))),Seq (Assign ("x",N 2),Assign ("y",N 0))) : CDER
```

and this is indeed the term of type `CDER` which codes the derivation

$$\dfrac{\dfrac{}{\texttt{true}, [] \Rightarrow_B \mathit{true}}\,(\Rightarrow_B \cdot 1) \qquad \dfrac{\dfrac{\dfrac{}{1, [] \Rightarrow_I 1}\,(\Rightarrow_I \cdot 1)}{\texttt{x} := 1, [] \Rightarrow_C [x = 1]}\,(\Rightarrow_C \cdot 2) \qquad \dfrac{\dfrac{}{0, [x = 1] \Rightarrow_I 0}\,(\Rightarrow_I \cdot 1)}{\texttt{y} := 0, [x = 1] \Rightarrow_C [x = 1\ y = 0]}\,(\Rightarrow_C \cdot 2)}{\texttt{x} := 1\,;\texttt{y} := 0, [] \Rightarrow_C [x = 1\ y = 0]}\,(\Rightarrow_C \cdot 3)}{\texttt{if true then (x} := 1\,;\texttt{y} := 0)\ \texttt{else (x} := 2\,;\texttt{y} := 0), [] \Rightarrow_C [x = 1\ y = 0]}\,(\Rightarrow_C \cdot 4)$$

as we would expect.

It is important to realise that writing such an ML function *does not* constitute a mathematical proof, since we have given no formal justification that the ML code actually does what we intuitively think it does. The function `ifseqproof` merely expresses the *idea* of the formal proof of Proposition 17. However, thinking in terms of functions can be a useful way to understand and to come up with this kind of proof.

What about the proof of Proposition 18? We will just prove half of the equivalence:

$$\forall S, S' \in \mathit{States}.\ \text{if} \quad \texttt{while}\ be\ \texttt{do}\ C_1, S \Rightarrow_C S' \quad \text{then} \quad \texttt{while}\ be\ \texttt{do}\ C_2, S \Rightarrow_C S' \qquad \text{(A.3)}$$

under the assumption that $C_1 \approx C_2$. In fact, to prove A.3,it suffices to assume just half of the equivalence of C_1 and C_2 *viz.*

$$\forall S, S' \in \mathit{States}.\ \text{if} \quad C_1, S \Rightarrow_C S' \quad \text{then} \quad C_2, S \Rightarrow_C S' \qquad \text{(A.4)}$$

The function coding the proof of A.3 will map derivations of the 'if' part to derivations of the 'then' part, but it will also need to take an extra argument which codes the assumption A.4. That argument will itself be a function `f` which takes derivations of $C_1, S \Rightarrow_C S'$ to derivations of $C_2, S \Rightarrow_C S'$ (the jargon word is that `f` is a *realizer* for the implication A.4). So, to sum up, the function `whilecongproof` which we want to define will, for any be,C_1,C_2,S and S', take as input

1. A derivation `d` of `while` be `do` $C_1, S \Rightarrow_C S'$.
2. A function `f` from derivations to derivations which maps a derivation of $C_1, S'' \Rightarrow_C S'''$ to a derivation of $C_2, S'' \Rightarrow_C S'''$, for any S'' and S'''.

and it will return a derivation of `while` be `do` $C_2, S \Rightarrow_C S'$. In fact, for a trivial technical reason, `whilecongproof` also has to take the command C_2 as input[2], thus the code ends up looking like this:

```
(* whilecongproof : CDER * COM * (CDER -> CDER) -> CDER *)
fun whilecongproof (d:CDER,C2,f:CDER->CDER) =
 case d of
   Cr6(bd,C1) => Cr6(bd,C2)
 | Cr7(bd,d1,d2) => let val d2' = whilecongproof (d2,C2,f)
                        val d1' = f d1
                    in
                        Cr7(bd,d1',d2')
                    end;
```

Notice how `whilecongproof` uses recursion in the case that the derivation `d` ends with an application of $(\Rightarrow_C \cdot 7)$, and that this corresponds exactly to the use of induction in the real proof of Proposition 18. Note also how `f` is used to transform the derivation of the second hypothesis of $(\Rightarrow_C \cdot 7)$ – this is the part in the proof where we appeal to the assumption that $C_1 \approx C_2$.

For example, let's take C_1 and C_2 to be instances of the equivalence of Proposition 17

[2]This is just because in the case that the derivation 1. above uses $(\Rightarrow_C \cdot 6)$, we need to return an application of $(\Rightarrow_C \cdot 6)$ which contains C_2, but we don't actually have it in our hand, and we can't use `f` to get it because we don't have any derivation about C_1 to supply as input to `f`.

```
(* these two are equivalent by ifseqproof *)
val c1 = readcom "if x=1 then y:=y+2 else y:=y+1 endif;x:=x-1";
val c2 = readcom "if x=1 then y:=y+2;x:=x-1 else y:=y+1;x:=x-1 endif";

(* embed them in a while command *)
val c1' = While(Bop(Greater,Pvar "x",N 0),c1);
val c2' = While(Bop(Greater,Pvar "x",N 0),c2);

(* We want to show c1' equivalent to c2',
   so start with derivation of c1' doing something
    NB. makedc : COM -> (STATES -> CDER) just
    executes a command in a given state and
    returns the entire derivation associated
    with that run.
*)
val d1 = makedc c1' [("y",0),("x",2)];

(* Now we can apply the proof to get a derivation of
   c2' doing the same thing. Note the use of ifseqproof
   as a witness/realizer that c1 is equivalent to c2.
*)
val d2 = whilecongproof(d1,c2,ifseqproof);
```

We end up with `d1` being a derivation that $\texttt{c1}', [y = 0\ x = 2] \Rightarrow_C [y = 3\ x = 0]$ and `d2` the derivation that $\texttt{c2}', [y = 0\ x = 2] \Rightarrow_C [y = 3\ x = 0]$ which is what we wanted. Unfortunately, the actual derivations are rather too large to be worth including here (around 3 feet wide).