Lecture Notes on

# *Denotational Semantics*

Part II of the Computer Science Tripos 2023/24

Meven Lennon-Bertrand
Department of Computer Science and Technology
University of Cambridge

# Contents

Notes 3

1 Introduction 4
- 1.1 A basic example 5
- 1.2 A semantics for loops 7
- 1.3 A taste of domain theory 8

2 Least Fixed Points 11
- 2.1 Posets and monotone functions 11
- 2.2 Least elements and pre-fixed points 12
- 2.3 Least upper bounds 13
- 2.4 Complete partial orders and domains 15
- 2.5 Continuous functions 17
- 2.6 Kleene's fixed point theorem 19
- 2.7 Exercises 20

3 Constructions on Domains 21
- 3.1 Flat domains 21
- 3.2 Products of domains 22
- 3.3 Function domains 25
- 3.4 Exercises 28

4 Scott Induction 29
- 4.1 Reasoning on fixed points 29
- 4.2 Building chain-closed subsets 30
- 4.3 Using Scott induction 31
- 4.4 Exercises 33

5 PCF 34
- 5.1 Terms and types 34
- 5.2 Operational Semantics 37
- 5.3 Contextual equivalence 40
- 5.4 Introducing denotational semantics 42
- 5.5 Exercises 43

6 Denotational Semantics for PCF 45
- 6.1 Types and contexts 45
- 6.2 Terms 45
- 6.3 Compositionality 48

6.4 Soundness . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
6.5 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

**7 Relating Denotational and Operational Semantics 52**
7.1 Formal approximation relation . . . . . . . . . . . . . . . . . . . . . . . 52
7.2 Proof of the fundamental property of formal approximation . . . . . . 54
7.3 Extensionality . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
7.4 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

**8 Full abstraction 60**
8.1 Failure of full abstraction . . . . . . . . . . . . . . . . . . . . . . . . . 60
8.2 Beyond full abstraction failure . . . . . . . . . . . . . . . . . . . . . . 62
8.3 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

# Notes

These notes are designed to accompany 10 lectures on Denotational Semantics for Part II of the Cambridge University Computer Science Tripos. They are substantially those of Andrew Pitts (who lectured the course from 1997 to 1999) with some changes and additions by Glynn Winskel (who lectured the course from 2000 to 2007), Marcelo Fiore (who lectured the course from 2008), and Meven Lennon-Bertrand. The material has been drawn from several sources, including the books mentioned below, previous versions of this course, and similar courses at some other universities. Any errors are of course all the author's own work.

## Recommended books

- Winskel, G. (1993). *The Formal Semantics of Programming Languages.* MIT Press. This is an excellent introduction to both the operational and denotational semantics of programming languages. As far as this course is concerned, the relevant chapters are 5, 8, 9, 10 (Sections 1 and 2), and 11.
- Tennent, R. D. (1991). *Semantics of Programming Languages.* Prentice-Hall. Parts I and II are relevant to this course.

## Further reading

- Gunter, C. A. (1992). *Semantics of Programming Languages. Structures and Techniques.* MIT Press.
  This is a graduate-level text containing much material not covered in this course. As far as this course is concerned, the relevant chapters are 1, 2, and 4–6.

## Feedback

Please fill out the online lecture course feedback form.

Meven Lennon-Bertrand
mgapb2@cam.ac.uk

# 1 Introduction

### What is this course about?

- Formal methods: tools for the specification, development, analysis and verification of software and hardware systems.
- Programming language theory: how to design, implement and reason about programming languages?
- Programming language semantics: what is the (mathematical) meaning of a program?

Goal: give an abstract and compositional (mathematical) model of programs.

### Why should we care?

- Insight: exposes the mathematical "essence" of programming language concepts.
- Language design: feedback from semantic concepts (monads, algebraic effects & effect handlers...).
- Rigour: semantics is necessary to specify/justify formal methods (compilers, type systems, code analysis, certification...).

### Styles of formal semantics

- **Operational**: meaning of a program in terms of the *steps of computation* it takes during execution (see Part IB Semantics).
- **Axiomatic**: indirect meaning of a program in terms of a *program logic* to reason about its properties (see Part II Hoare Logic & Model Checking).
- **Denotational**: meaning of a program defined abstractly as object of some suitable *mathematical structure* (see this course).

### Denotational semantics in a nutshell

$$\begin{array}{rcl}
\text{Syntax} & \xrightarrow{[\![-]\!]} & \text{Semantics} \\
\text{Program } P & \mapsto & \text{Denotation } [\![P]\!] \\
 & & \\
\text{Recursive program} & \mapsto & \text{Partial recursive function} \\
\text{Boolean circuit} & \mapsto & \text{Boolean function} \\
 & \ldots & \\
\text{Type} & \mapsto & \text{Domain} \\
\text{Program} & \mapsto & \text{Continuous functions between domains}
\end{array}$$

**Properties of denotational semantics**

Abstraction:

- mathematical object, implementation/machine independent;
- captures the abstract essence of programming language concepts;
- should relate to practical implementations, though...

Compositionality:

- The denotation of a phrase is defined using the *denotation* of its sub-phrases.
- $[\![P]\!]$ represents the contribution of $P$ to *any* program containing $P$.
- Much more flexible than whole-program semantics.

## 1.1 A basic example

Consider the basic programming language IMP over arithmetic and boolean expressions with control structures given by assignment, sequencing, conditionals, and loops, as follows.

**IMP syntax** ranges over *integers*

Arithmetic expressions

$$A \in \mathbf{Aexp} ::= \underline{n} \mid L \mid A + A \mid \ldots$$

Boolean expressions

$$B \in \mathbf{Bexp} ::= \texttt{true} \mid \texttt{false} \mid A = A \mid \neg B \mid \ldots$$

Commands

ranges over a set $\mathbb{L}$ of *locations*

$$C \in \mathbf{Comm} ::= \texttt{skip} \mid L := A \mid C; C \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C \mid \texttt{while}\ B\ \texttt{do}\ C$$

A *denotational semantics* for this programming language is constructed by giving a domain of interpretation to each of the syntactic categories, together with semantic functions that compositionally describe the meaning of the syntactic constructions.

Here we have three kinds of expressions, and so three semantic functions, mapping each expression to their denotation:

$$\begin{aligned}
\mathcal{A} &: \mathbf{Aexp} \to (\mathrm{State} \to \mathbb{Z}) \\
\mathcal{B} &: \mathbf{Bexp} \to (\mathrm{State} \to \mathbb{B}) \\
\mathcal{C} &: \mathbf{Comm} \to (\mathrm{State} \rightharpoonup \mathrm{State})
\end{aligned}$$

where ⇀ denotes partial functions and

$$\begin{aligned} \mathrm{State} &= (\mathbb{L} \rightharpoonup \mathbb{Z}) \\ \mathbb{Z} &= \{\ldots, -1, 0, 1, \ldots\} \\ \mathbb{B} &= \{\mathrm{true}, \mathrm{false}\}\,. \end{aligned}$$

**Semantics of arithmetic and boolean expressions**

The requirement of denotational semantics is quite a tough one. It means that the collection of mathematical objects we use to give denotations has to be sufficiently rich that it supports operations for modelling all the constructs of the programming language. For instance, the fact that our expressions contain variables means than the plain $\mathbb{Z}$ and $\mathbb{B}$ are inadequate for the semantics of **Aexp** and **Bexp**. Instead, we must have a more complex semantics, and interpret expressions as functions from the set of states.

$$\begin{aligned} \mathcal{A}[\![\underline{n}]\!] &= \lambda s \in \mathrm{State}.\ n \\ \mathcal{A}[\![A_1 + A_2]\!] &= \lambda s \in \mathrm{State}.\ \mathcal{A}[\![A_1]\!]\,(s) + \mathcal{A}[\![A_2]\!]\,(s) \\ \mathcal{A}[\![L]\!] &= \lambda s \in \mathrm{State}.\ s(L) \end{aligned}$$

$$\begin{aligned} \mathcal{B}[\![\texttt{true}]\!] &= \lambda s \in \mathrm{State}.\ \mathrm{true} \\ \mathcal{B}[\![\texttt{false}]\!] &= \lambda s \in \mathrm{State}.\ \mathrm{false} \\ \mathcal{B}[\![A_1 = A_2]\!] &= \lambda s \in \mathrm{State}.\ \mathrm{eq}\,(\mathcal{A}[\![A_1]\!]\,(s), \mathcal{A}[\![A_2]\!]\,(s)) \\ &\qquad \text{where } \mathrm{eq}(a, a') = \begin{cases} \mathrm{true} & \text{if } a = a' \\ \mathrm{false} & \text{if } a \neq a' \end{cases} \end{aligned}$$

**Semantics of commands**

Some commands are straightforward to deal with. For example, conditional expressions can be given a denotational semantics in terms of a *semantic* branching function applied to the denotations of the immediate sub-expressions. Similarly, the denotational semantics of the sequential composition of commands can be given by the operation of composition of partial functions from states to states. In a sense, we are lucky: our choice of semantics already supports semantic operations corresponding to these commands.

$$\mathcal{C}[\![\texttt{skip}]\!] = \lambda s \in \text{State}.\ s$$

This is compositionality!

$$\begin{aligned}
\mathcal{C}[\![\texttt{if } B \texttt{ then } C \texttt{ else } C']\!] &= \lambda s \in \text{State}.\ \text{if}\,(\mathcal{C}[\![B]\!]\,(s), \mathcal{C}[\![C]\!]\,(s), \mathcal{C}[\![C']\!]\,(s)) \\
&\quad \text{where if}(b, x, x') = \begin{cases} x & \text{if } b = \text{true} \\ x' & \text{if } b = \text{false} \end{cases} \\
\mathcal{C}[\![L := A]\!] &= \lambda s \in \text{State}.\ s[L \mapsto \mathcal{A}[\![A]\!]\,(s)] \\
&\quad \text{where } s[L \mapsto n](L') = \begin{cases} n & \text{if } L' = L \\ s(L) & \text{otherwise} \end{cases} \\
\mathcal{C}[\![C; C']\!] &= \mathcal{C}[\![C']\!] \circ \mathcal{C}[\![C]\!] \\
&= \lambda s \in \text{State}.\ \mathcal{C}[\![C']\!]\,(\mathcal{C}[\![C]\!]\,(s))
\end{aligned}$$

From now on, we keep only $[\![-]\!]$ and drop the names of the semantic functions, which should be clear from the context.

## 1.2 A semantics for loops

We now proceed to consider the last missing piece for our denotational semantics of the basic programming language IMP: `while`-loops. However, this looping construct is not so easy to explain compositionally! The transition semantics of a `while`-loop

$$\langle \texttt{while } B \texttt{ do } C, s\rangle \rightsquigarrow \langle \texttt{if } B \texttt{ then } (C; \texttt{while } B \texttt{ do } C) \texttt{ else skip}, s\rangle$$

suggests that these two should have the same denotation. Using the denotational semantics of sequential composition, `if` and `skip`, we obtain the following:

$$\begin{aligned}
[\![\texttt{while } B \texttt{ do } C]\!] &= [\![\texttt{if } B \texttt{ then } (C; \texttt{while } B \texttt{ do } C) \texttt{ else skip}]\!] \\
&= \lambda s \in \text{State}.\ \text{if}([\![B]\!]\,, [\![\texttt{while } B \texttt{ do } C]\!] \circ [\![C]\!]\,(s), s)
\end{aligned}$$

This cannot be used directly to define $[\![\texttt{while } B \texttt{ do } C]\!]$, since the right-hand side contains the left-hand side Rather, $[\![\texttt{while } B \texttt{ do } C]\!]$ should be a solution of the following *fixed point equation*

$$[\![\texttt{while } B \texttt{ do } C]\!] = F_{[\![B]\!],[\![C]\!]}(\texttt{while } B \texttt{ do } C)$$

$$\begin{aligned}
\text{where} \quad F_{b,c} : \quad (\text{State} \rightharpoonup \text{State}) &\to (\text{State} \rightharpoonup \text{State}) \\
w &\mapsto \lambda s \in \text{State}.\ \text{if}(b(s), w \circ c(s), s).
\end{aligned}$$

The requirement is now clear, but this raises more questions:

- Why/when does $w = F_{b,c}(w)$ have a solution?
- What if it has several solutions? Which one should be our $[\![\texttt{while } B \texttt{ do } C]\!]$?

## 1.3 A taste of domain theory

Such fixed point equations arise very often in giving denotational semantics to languages with recursive features. Beginning with Dana Scott's pioneering work in the late 60s, a mathematical theory called *domain theory* has been developed to provide a setting in which not only can we always find solutions for the fixed point equations arising from denotational semantics, but also we can pick out solutions that are minimal in a suitable sense—and this turns out to ensure a good match between denotational and operational semantics. The key idea is to consider a partial order between the mathematical objects used as denotations—this partial order expresses the fact that one object is *approximated by*, or *carries more information than*, or is *more well-defined than* another one below it in the ordering. Then the minimal solution of a fixed point equation can be constructed as the limit of an increasing chain of approximations to the solution. These ideas will be made mathematically precise and general in the next section; but first we illustrate how they work out concretely in a particular case.

For definiteness, let us consider the particular `while`-loop

$$\texttt{while}\ X > 0\ \texttt{do}\ (Y := X * Y; X := X - 1) \tag{1}$$

where $X$ and $Y$ are two distinct locations and where the set of locations $\mathbb{L}$ is simply $\{X, Y\}$. In this case a state is an assignment $[X \mapsto x, Y \mapsto y]$ with $x, y \in \mathbb{Z}$, recording the current contents of the locations $X$ and $Y$ respectively.

We are trying to define the denotation of (1) as a partial function $w : \mathrm{State} \rightharpoonup \mathrm{State}$ that should be a solution to the fixed point equation

$$w = F_{[\![X>0]\!],[\![Y:=X*Y;X:=X-1]\!]}(w).$$

That is, we are looking for a fixed point of the following $F : D \to D$, where $D$ is $(\mathrm{State} \rightharpoonup \mathrm{State})$:

$$F(w)([X \mapsto x, Y \mapsto y]) = \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ w([X \mapsto x-1, Y \mapsto x \cdot y]) & \text{if } x > 0. \end{cases}$$

### The poset of partial functions

**Partial order $\sqsubseteq$ on $D$:**

$w \sqsubseteq w'$ if for all $s \in \mathrm{State}$, if $w$ is defined at $s$
then so is $w'$ and moreover $w(s) = w'(s)$.
if the graph of $w$ is included in the graph of $w'$.

The order $\sqsubseteq$ embodies the kind of "information ordering" mentioned above: if $w \sqsubseteq w'$, then $w'$ agrees with $w$ wherever the latter is defined, but it may be defined at some other arguments as well.

**Least element $\bot \in D$:**

$$\begin{array}{lcl} \bot & = & \text{totally undefined partial function} \\ & = & \text{partial function with empty graph} \end{array}$$

This is a minimal element with respect to $\sqsubseteq$, that is, the function with the least information possible: the one that is never defined.

### Approximating the fixed point

Define $w_n = F^n(w)$, that is $\begin{cases} w_0 & = \bot \\ w_{n+1} & = F(w_n) \end{cases}$. By definition, we have

$$w_1[X \mapsto x, Y \mapsto y] = F(\bot)[X \mapsto x, Y \mapsto y] = \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ \text{undefined} & \text{if } x \geq 1 \end{cases}$$

$$w_2[X \mapsto x, Y \mapsto y] = F(w_1)[X \mapsto x, Y \mapsto y] = \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ [X \mapsto 0, Y \mapsto y] & \text{if } x = 1 \\ \text{undefined} & \text{if } x \geq 2 \end{cases}$$

$$w_3[X \mapsto x, Y \mapsto y] = F(w_2)[X \mapsto x, Y \mapsto y] = \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \leq 0 \\ [X \mapsto 0, Y \mapsto y] & \text{if } x = 1 \\ [X \mapsto 0, Y \mapsto 2y] & \text{if } x = 2 \\ \text{undefined} & \text{if } x \geq 3 \end{cases}$$

$$w_n[X \mapsto x, Y \mapsto y] = \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x < 0 \\ [X \mapsto 0, Y \mapsto (x!) \cdot y] & \text{if } 0 \leq x < n \\ \text{undefined} & \text{if } x \geq n \end{cases}$$

That is, we obtain an increasing sequence of partial functions

$$w_0 \sqsubseteq w_1 \sqsubseteq \ldots \sqsubseteq w_n \sqsubseteq \ldots$$

defined on larger and larger sets of states $(x, y)$ and agreeing where they are defined. The union of all these partial functions is the element $w_\infty \in D$ given by

$$w_\infty[X \mapsto x, Y \mapsto y] = \bigsqcup_{i \in \mathbb{N}} w_i = \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x < 0 \\ [X \mapsto 0, Y \mapsto (x!) \cdot y] & \text{if } x \geq 0 \end{cases}$$

Note that $w_\infty$ is a fixed point of the function $F$, since for all $x$ and $y$ we have

$$
\begin{aligned}
F(w_\infty)[X \mapsto x, Y \mapsto y] &= \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \le 0 \\ w_\infty[X \mapsto x-1, Y \mapsto x \cdot y] & \text{if } x > 0 \end{cases} && \text{(by definition of } F\text{)} \\
&= \begin{cases} [X \mapsto x, Y \mapsto y] & \text{if } x \le 0 \\ [X \mapsto 0, Y \mapsto (x-1)! \cdot x \cdot y] & \text{if } x > 0 \end{cases} && \text{(by definition of } w_\infty\text{)} \\
&= w_\infty[X \mapsto x, Y \mapsto y]
\end{aligned}
$$

In fact one can show that $w_\infty$ is the *least* fixed point of $F$, in the sense that for all $w \in D$

$$w = F(w) \quad \Rightarrow \quad w_\infty \sqsubseteq w.$$

This least fixed point $w_\infty$ is what we take as the denotation of

$$\texttt{while}\ X > 0\ \text{do}\ (Y := X * Y; X := X - 1).$$

Its construction is an instance of Kleene's Fixed Point Theorem that we prove in the next section. Note also that $w_\infty$ is indeed the function from states to states that we get from the operational semantics of the command, as given in the Part IB course on semantics.

# 2 Least Fixed Points

This section introduces a mathematical theory, *domain theory*, which amongst other things provides a general framework for constructing the least fixed points used in the denotational semantics of various programming language features. The theory was introduced by Dana Scott in the 70s.

## 2.1 Posets and monotone functions

Domain theory makes use of partially ordered sets satisfying certain completeness properties.

**Definition 1 (Partially ordered set)** A *partial order* on a set $D$ is a binary relation $\sqsubseteq$ that is
*reflexive:* $\forall d \in D.\ d \sqsubseteq d$
*transitive:* $\forall d, d', d'' \in D.\ d \sqsubseteq d' \sqsubseteq d'' \Rightarrow d \sqsubseteq d''$
*antisymmetric:* $\forall d, d' \in D.\ d \sqsubseteq d' \sqsubseteq d \Rightarrow d = d'$. *
Such a pair $(D, \sqsubseteq)$ is called a *partially ordered set*, or *poset*. $D$ is called the *underlying set* of the poset $(D, \sqsubseteq)$.

Most of the time we will refer to posets just by naming their underlying sets and use the same symbol $\sqsubseteq$ to denote the partial order in a variety of different posets.

**Example 1 (Domain of partial functions, $X \rightharpoonup Y$)** The set $(X \rightharpoonup Y)$ of all partial functions from a set $X$ to a set $Y$ can be made into a poset, as follows:
*Underlying set:* partial functions $f$ with domain of definition $\mathrm{dom}(f) \subseteq X$ and taking values in $Y$;
*Order:* $f \sqsubseteq g$ if $\mathrm{dom}(f) \subseteq \mathrm{dom}(g)$ and $\forall x \in \mathrm{dom}(f).\ f(x) = g(x)$, *i.e.* if $\mathrm{graph}(f) \subseteq \mathrm{graph}(g)$.
It was this domain for the case $X = Y = \mathrm{State}$ that we used for the denotation of commands in Section 1.1. *

**Definition 2 (Monotone function)** A function $f : D \to E$ between posets is *monotone* if

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

*

**Example 2** Given posets $D$ and $E$, for each $e \in E$ it is easy to see that the *constant function $D \to E$ with value $e$*, $\lambda d \in D\,.\,e$, is monotone. *

**Example 3** When $D$ is the domain of partial functions $(\mathrm{State} \rightharpoonup \mathrm{State})$ (Example 1), the function $F_{b,c} : D \to D$ defined in Section 1.2 in connection with the denotational semantics of `while`-loops is a monotone function. *

We leave the verification of this as an exercise.

## 2.2 Least elements and pre-fixed points

**Definition 3 (Least element)** Suppose that $D$ is a poset and that $S$ is a subset of $D$. An element $d \in S$ is the *least* element of $S$ if it satisfies

$$\forall x \in S.\ d \sqsubseteq x.$$

*

If it exists, it is unique (by antisymmetry), and is written $\bot_S$, or simply $\bot$.

Beware: a poset may not have a least element! For example, $\mathbb{Z}$ with its usual partial order does not have a least element.

**Definition 4 (Fixed point)** A *fixed point* for a function $f : D \to D$ is an element $d \in D$ satisfying $f(d) = d$. *

However, when $D$ is a poset, we can consider the weaker notion of *pre-fixed point.*

**Definition 5 ((Least) pre-fixed point)** Let $D$ be a poset and $f : D \to D$ be a function. An element $d \in D$ is a *pre-fixed point* of $f$ if it satisfies $f(d) \sqsubseteq d$. The *least pre-fixed point* of $f$, if it exists, will be written

$$\mathrm{fix}(f)$$

It is thus (uniquely) specified by the two properties:

$$f(\mathrm{fix}(f)) \sqsubseteq \mathrm{fix}(f) \qquad \text{(lfp-fix)}$$

$$\forall d \in D.\ f(d) \sqsubseteq d \Rightarrow \mathrm{fix}(f) \sqsubseteq d \qquad \text{(lfp-least)}$$

*

**Proposition 1 (Least pre-fixed points are least fixed points)** *Suppose $D$ is a poset and $f : D \to D$ is a function possessing a least pre-fixed point,* $\mathrm{fix}(f)$*. Provided $f$ is* monotone*,* $\mathrm{fix}(f)$ *is in particular a fixed point for $f$, and hence is the least element of the set of fixed points for $f$, since every fixed point is a pre-fixed point.* *

Proof By definition, $\mathrm{fix}(f)$ is a pre-fixed point. Thus, by monotony of $f$, we can apply $f$ to both sides of (lfp1) to conclude that

$$f(f(\mathrm{fix}(f))) \sqsubseteq f(\mathrm{fix}(f)).$$

Then applying property (lfp2) with $d = f(\mathrm{fix}(f))$, we get that

$$\mathrm{fix}(f) \sqsubseteq f(\mathrm{fix}(f)).$$

Combining this with (lfp1) and the anti-symmetry property of the partial order $\sqsubseteq$, we get $f(\mathrm{fix}(f)) = \mathrm{fix}(f)$, as required. □

Thus, while being a pre-fixed point is a weaker notion, being the *least* pre-fixed point is stronger than being the least fixed point.

## 2.3 Least upper bounds

**Definition 6 (Least upper bound of a chain)** A countable, increasing *chain* in a poset $D$ is a sequence $(d_i)_{i\in\mathbb{N}}$ of elements of $D$ satisfying

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

*

An *upper bound* for the chain is any $d \in D$ satisfying $\forall n \in \mathbb{N}.\ d_n \sqsubseteq d$. If it exists, the *least upper bound*, or *lub*, of the chain will be written as $\bigsqcup_{n\geq 0} d_n$. Thus, by definition:

- $\forall m \in \mathbb{N}.\ d_m \sqsubseteq \bigsqcup_{n\geq 0} d_n$.
- For any $d \in D$, if $\forall m \in \mathbb{N}.\ d_m \sqsubseteq d$, then $\bigsqcup_{n\geq 0} d_n \sqsubseteq d$.

**Remark 1**

(i) We will not need to consider uncountable, or decreasing chains in a poset: so a 'chain' will always mean a countable, increasing chain.

(ii) We will also not need to consider least upper bounds of general sets rather than chains – but most of what we do here generalizes smoothly.

(iii) While the least element of $S$ is an element of $S$, the lub of a chain is not necessarily an element of the chain (and, in fact, the interesting case is when it is not).

(iv) Like the least element of a set, the lub of a chain is unique if it exists. (It does not have to exist: for example the chain $0 \leq 1 \leq 2 \leq \dots$ in $\mathbb{N}$ has no upper bound, hence no lub.)

(v) A least upper bound is sometimes called a *supremum*. Some other common notations for $\bigsqcup_{n\geq 0} d_n$ are:

$$\bigsqcup_{n=0}^{\infty} d_n \qquad \text{and} \qquad \bigsqcup\{d_n \mid n \geq 0\}\ .$$

*

The latter can be used more generally with any set: $\bigsqcup S$ is the supremum of $S$.

We can already spell out some easy properties of lubs.

**Proposition 2 (Monotonicity of lubs)** *For every pair of chains*

$$d_0 \sqsubseteq d_1 \sqsubseteq \dots \sqsubseteq d_n \sqsubseteq \dots \quad \textit{and} \quad e_0 \sqsubseteq e_1 \sqsubseteq \dots \sqsubseteq e_n \sqsubseteq \dots$$

*if* $d_n \sqsubseteq e_n$ *for all* $n \in \mathbb{N}$ *then* $\bigsqcup_n d_n \sqsubseteq \bigsqcup_n e_n$, *provided they exist.* *

**Proposition 3 (Discarding elements)** *If we discard any finite number of elements at the beginning of a chain, we do not affect its set of upper bounds and hence do not change its lub. That is, for any* $N \in \mathbb{N}$ *we have (provided any of the two exists):*

$$\bigsqcup_{n\geq 0} d_n = \bigsqcup_{n\geq 0} d_{N+n}.$$

*

**Proposition 4 (Eventually constant chain)** *The elements of a chain do not necessarily have to be distinct. In particular, we say that a chain $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ is* eventually constant *if for some $N \in \mathbb{N}$ it is the case that $\forall n \geq N.\ d_n = d_N$. For such a chain, we have $\bigsqcup_{n\geq 0} d_n = d_N$.* $\ast$

**Proposition 5 (Diagonalisation)** *Let $D$ be a poset. Suppose that the doubly-indexed family of elements $d_{m,n} \in D$ $(m, n \geq 0)$ satisfies*

$$m \leq m' \wedge n \leq n' \Rightarrow d_{m,n} \sqsubseteq d_{m',n'}. \tag{$\dagger$}$$

$\ast$

*Then, assuming they exist, the lubs form two chains*

$$\bigsqcup_{n\geq 0} d_{0,n} \sqsubseteq \bigsqcup_{n\geq 0} d_{1,n} \sqsubseteq \bigsqcup_{n\geq 0} d_{2,n} \sqsubseteq \ldots$$

*and*

$$\bigsqcup_{m\geq 0} d_{m,0} \sqsubseteq \bigsqcup_{m\geq 0} d_{m,1} \sqsubseteq \bigsqcup_{m\geq 0} d_{m,2} \sqsubseteq \ldots$$

*Moreover, again assuming they exist,*

$$\bigsqcup_{m\geq 0}\left(\bigsqcup_{n\geq 0} d_{m,n}\right) = \bigsqcup_{k\geq 0} d_{k,k} = \bigsqcup_{n\geq 0}\left(\bigsqcup_{m\geq 0} d_{m,n}\right).$$

Proof First note that if $m \leq m'$ then

$$\begin{aligned} d_{m,n} &\sqsubseteq d_{m',n} && \text{by property } (\dagger) \text{ of the } d_{m,n} \\ &\sqsubseteq \bigsqcup_{n'\geq 0} d_{m',n'} && \text{because the lub is an upper bound} \end{aligned}$$

for all $n \geq 0$, hence, by minimality of the lub, $\bigsqcup_{n\geq 0} d_{m,n} \sqsubseteq \bigsqcup_{n'\geq 0} d_{m',n'}$. Thus, we do indeed get a chain of lubs

$$\bigsqcup_{n\geq 0} d_{0,n} \sqsubseteq \bigsqcup_{n\geq 0} d_{1,n} \sqsubseteq \bigsqcup_{n\geq 0} d_{2,n} \sqsubseteq \ldots$$

Using the bound property twice we have

$$d_{k,k} \sqsubseteq \bigsqcup_{n\geq 0} d_{k,n} \sqsubseteq \bigsqcup_{m\geq 0} \bigsqcup_{n\geq 0} d_{m,n}$$

for each $k \geq 0$, and so by minimality of the lub,

$$\bigsqcup_{k\geq 0} d_{k,k} \sqsubseteq \bigsqcup_{m\geq 0} \bigsqcup_{n\geq 0} d_{m,n}. \tag{2}$$

Conversely, for each $m, n \geq 0$, note that

$$\begin{aligned} d_{m,n} &\sqsubseteq d_{\max(m,n),\max(m,n)} && \text{by property } (\dagger) \\ &\sqsubseteq \bigsqcup_{k\geq 0} d_{k,k} && \text{because the lub is an upper bound} \end{aligned}$$

and hence applying minimality of the lub twice we have

$$\bigsqcup_{m\geq 0} \bigsqcup_{n\geq 0} d_{m,n} \sqsubseteq \bigsqcup_{k\geq 0} d_{k,k}. \tag{3}$$

Combining (2) and (3) with the anti-symmetry property of $\sqsubseteq$ yields the desired equality. We obtain the additional equality by the same argument but interchanging the roles of $m$ and $n$. □

## 2.4 Complete partial orders and domains

In this course, we will be interested in certain posets, called chain complete posets and domains, which enjoy completeness properties: every chain has a least upper bound.

**Definition 7 (Cpos)** A *chain complete poset*, or *cpo*, is a poset $(D, \sqsubseteq)$ where all chains have a least upper bound. *

In a cpo, we only need to verify that a sequence of elements forms a chain to know it has a lub, so *e.g.* in Proposition 5 above we automatically know that all the lubs exist.

**Definition 8 (Domain)** A *domain* is a cpo that possesses a least element. *

It should be noted that the term 'domain' is used rather loosely in the literature on denotational semantics: there are many kinds of domains, enjoying various extra order-theoretic properties over and above the rather minimal ones of chain-completeness and possession of a least element that we need for this course. Still, most of what we will do here carries over directly to these other settings.

**Example 4 (Domain of partial functions)** The poset $(X \rightharpoonup Y)$ of partial functions from a set $X$ to a set $Y$, as defined in Example 1 can be made into a domain.

*Least element:* $\perp$ is the totally undefined function.

*Lub of a chain:* $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots$ has lub $f$ such that

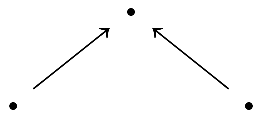
$$f(x) = \begin{cases} f_n(x) & \text{if } x \in \mathrm{dom}(f_n) \text{ for some } n \\ \text{undefined} & \text{otherwise} \end{cases}$$

*

Note that this definition of the lub is well-defined *only* if the $f_n$ form a chain. Indeed, this implies that the $f_n$ agree where they are defined, and so the definition is unambiguous. We leave it as an exercise to check that this $f$ is indeed the least upper bound of $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots$ in the poset $(X \rightharpoonup Y, \sqsubseteq)$.

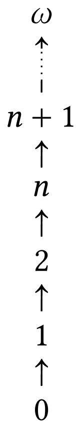
It was this domain for the case $X = Y = \text{State}$ that we used for the denotation of commands in Section 1.1.

**Example 5 (Finite cpos)** Any poset $(D, \sqsubseteq)$ whose underlying set $D$ is finite is a cpo. For in such a poset any chain is eventually constant, and we noted in Proposition 4 that such a chain always possesses a lub. Of course, a finite poset need not have a least element, and hence need not be a domain—for example, consider the poset with Hasse diagram



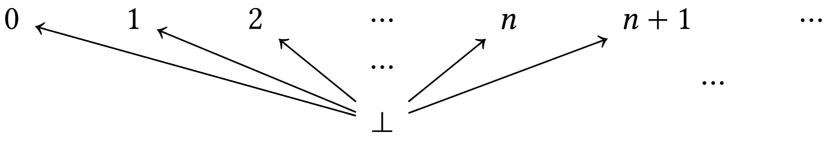
(A *Hasse diagram* for a poset $(D, \sqsubseteq)$ is a directed graph $G$ with $D$ as vertices, such that $x \sqsubseteq y$ iff there is a path in $G$ from $x$ to $y$. Equivalently, $\sqsubseteq$ is the reflexive, transitive closure of the (oriented) adjacency relation of $G$, where $x$ is adjacent to $y$ if there is an edge from $x$ to $y$.) *

**Example 6 ('Vertical' extended natural numbers)** The set $\Omega$, given by the following Hasse diagram, is a domain.



*

**Example 7 (Flat natural numbers)** The *flat natural numbers* $\mathbb{N}_\bot$ is the poset given by the following Hasse diagram:

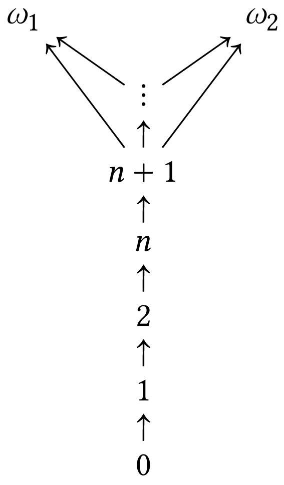


$\star$

A partial function $X \rightharpoonup \mathbb{N}$ is the same as a monotone function from the poset $(X, =)$ (equality is a trivial pre-order) to $(\mathbb{N}_\bot, \sqsubseteq)$. Thus, flat natural numbers give us a way to express partiality, which we will use further in this course.

**Example 8 (Non-example: natural numbers)** The set of natural numbers $\mathbb{N}$ equipped with the usual partial order, $\leq$, is not a cpo. For the increasing chain $0 \leq 1 \leq 2 \leq \ldots$ has no upper bound in $\mathbb{N}$. $\star$

**Example 9 (Non-example: no least upper bound)** Consider a modified version of Example 6, in which we adjoin not one but two different upper bounds to $\mathbb{N}$, corresponding to the following Hasse diagram:



Then the increasing chain $0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq \ldots$ has two upper bounds ($\omega_1$ and $\omega_2$), but no least one (since $\omega_1 \not\sqsubseteq \omega_2$ and $\omega_2 \not\sqsubseteq \omega_1$). So this poset is not a cpo. $\star$

## 2.5 Continuous functions

**Definition 9 (Continuity)** Given two cpos $D$ and $E$, a function $f\colon D \to E$ is *continuous* if

- it is monotone, and

- it preserves lubs of chains, *i.e.* for all chains $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ in $D$, we have

$$f(\bigsqcup_{n\geq 0} d_n) = \bigsqcup_{n\geq 0} f(d_n)$$

$\ast$

**Definition 10 (Strictness)** Let $D$ and $E$ be two posets with least elements $\bot_D$ and $\bot_E$. A function $f$ is *strict* if $f(\bot_D) = \bot_E$.

**Remark 2** Note that if $f\colon D \to E$ is monotone and $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ is a chain in $D$, then applying $f$ we get a chain $f(d_0) \sqsubseteq f(d_1) \sqsubseteq f(d_2) \sqsubseteq \ldots$ in $E$. Moreover, if $d$ is an upper bound of the first chain, then $f(d)$ is an upper bound of the second and hence is greater than its lub. Hence, if $f\colon D \to E$ is a monotone function between cpos, we always have

$$\bigsqcup_{n\geq 0} f(d_n) \sqsubseteq f(\bigsqcup_{n\geq 0} d_n)$$

Therefore (using the antisymmetry property of $\sqsubseteq$), *to check that a monotone function $f$ between cpos is continuous, it suffices to check for each chain $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ in $D$ that*

$$f\left(\bigsqcup_{n\geq 0} d_n\right) \sqsubseteq \bigsqcup_{n\geq 0} f(d_n)$$

*holds in $E$.*

$\ast$

**Example 10 (Constant functions)** Given cpos $D$ and $E$, for each $e \in E$ the constant function $D \to E$ with value $e$, $\lambda d \in D.\ e$, is continuous. $\ast$

**Example 11** When $D$ is the domain of partial functions (State $\rightharpoonup$ State), the function $F_{b,c}\colon D \to D$ defined in Section 1.2 connection with the denotational semantics of `while`-loops is a continuous function. We leave the verification of this as an exercise. $\ast$

**Example 12 (Non-example)** Let $\Omega$ be the domain of vertical natural numbers, as defined in Example 6. Then the function $f\colon \Omega \to \Omega$ defined by

$$\begin{cases} f(n) = 0 & (n \in \mathbb{N}) \\ f(\omega) = \omega. \end{cases}$$

is monotone and strict, but it is not continuous because

$$f(\bigsqcup_{n\geq 0} n) = f(\omega) = \omega \neq 0 = \bigsqcup_{n\geq 0} 0 = \bigsqcup_{n\geq 0} f(n).$$

$\ast$

## 2.6 Kleene's fixed point theorem

We now reach the key result about continuous functions on domains which permits us to give denotational semantics of programs involving recursive features.

Define $f^n(x)$ as follows:

$$\begin{cases} f^0(x) & \overset{\text{def}}{=} x \\ f^{n+1}(x) & \overset{\text{def}}{=} f(f^n(x)). \end{cases}$$

Since $\forall d \in D.\, \bot \sqsubseteq d$, one has $f^0(\bot) = \bot \sqsubseteq f^1(\bot)$; and by monotonicity of $f$

$$f^n(\bot) \sqsubseteq f^{n+1}(\bot) \Rightarrow f^{n+1}(\bot) = f(f^n(\bot)) \sqsubseteq f(f^{n+1}(\bot)) = f^{n+2}(\bot).$$

Therefore, by induction on $n \in \mathbb{N}$, the elements $f^n(\bot)$ form a chain in $D$:

$$f_0(\bot) \sqsubseteq f_1(\bot) \sqsubseteq \dots \sqsubseteq f_n(\bot) \sqsubseteq f_{n+1}(\bot) \sqsubseteq \dots$$

So since $D$ is a cpo, this chain has a least upper bound.

**Theorem 6 (Kleene's fixed point theorem)** *Let $f\colon D \to D$ be a continuous function on a domain $D$. Then $f$ possesses a least pre-fixed point, given by*

$$\operatorname{fix}(f) = \bigsqcup_{n \geq 0} f^n(\bot).$$

*

*By Proposition 1,* $\operatorname{fix}(f)$ *is thus also the* least fixed point *of $f$.*

This theorem is sometimes attributed (amongst others) to Tarski. Another, different, fixed point theorem more often attributed to Tarski (or Knaster-Tarski) gives the existence of fixed point of monotone functions on complete lattices (posets where every subset has an upper and lower bound).

Proof First note that

$$\begin{aligned} f(\operatorname{fix}(f)) &= f(\bigsqcup_{n \geq 0} f^n(\bot)) \\ &= \bigsqcup_{n \geq 0} f(f^n(\bot)) && \text{by continuity of } f \\ &= \bigsqcup_{n \geq 0} f^{n+1}(\bot) && \text{by definition of } f^n \\ &= \bigsqcup_{n \geq 0} f^n(\bot) && \text{by Proposition 3} \\ &= \operatorname{fix}(f). \end{aligned}$$

So $\mathrm{fix}(f)$ is a fixed point for $f$, and hence in particular a pre-fixed point. To verify that it is a *least* pre-fixed point, suppose that $d \in D$ satisfies $f(d) \sqsubseteq d$. Then since $\bot$ is least in $D$

$$f^0(\bot) = \bot \sqsubseteq d$$

and assuming $f^n(\bot) \sqsubseteq d$, we have

$$\begin{array}{rll} f^{n+1}(\bot) = f(f^n(\bot)) & \sqsubseteq f(d) & \text{monotonicity of } f \\ & \sqsubseteq d & \text{by assumption on } d. \end{array}$$

Hence by induction on $n \in \mathbb{N}$ we have $\forall n \in \mathbb{N}.\ f^n(\bot) \sqsubseteq d$. Therefore $d$ is an upper bound for the chain and hence lies above the least such, *i.e.*

$$\mathrm{fix}(f) = \bigsqcup_{n \geq 0} f^n(\bot) \sqsubseteq d.$$

Since this is the case for every pre-fixed point, $\mathrm{fix}(f)$ is indeed the least pre-fixed point, as claimed. □

**Example 13** Our running example, the function $F_{[\![B]\!],[\![C]\!]}$, is continuous (Exercise 3) on the domain State ⇀ State. So we can apply the fixed point theorem above, and define $[\![\texttt{while}\ B\ \mathrm{do}\ C]\!]$ to be $\mathrm{fix}(F_{[\![B]\!],[\![C]\!]})$. Actually, the method used to construct the partial function $w_\infty$ at the end of Section 1.2 is an instance of the method used in the proof of the fixed point theorem to construct least pre-fixed points. ⋆

## 2.7 Exercises

**Exercise 1** Verify the claims of Examples 1 and 4: that the relation $\sqsubseteq$ defined there is a partial order; that $f$ is indeed the lub of the chain $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots$; and that the totally undefined partial function is the least element.

**Exercise 2** Show the properties of least upper bounds given in Remark 1(iv), Proposition 2, Proposition 3 and Proposition 4:

- lubs are unique;
- lubs are monotone;
- discarding a finite number of elements at the beginning of a chain does not change its lub;
- eventually constant chains always have a lub, which is their ultimate value.

**Exercise 3** Let $b \in \mathrm{State} \rightharpoonup \mathbb{B}$ and $c \in \mathrm{State} \rightharpoonup \mathrm{State}$ be two monotone and continuous functions. Recall we defined $F_{b,c}$ in Section 1.2 as

$$\begin{array}{rrcl} F_{b,c}: & (\mathrm{State} \rightharpoonup \mathrm{State}) & \to & (\mathrm{State} \rightharpoonup \mathrm{State}) \\ & w & \mapsto & \lambda s \in \mathrm{State}.\ \mathrm{if}(b(s), w \circ c(s), s). \end{array}$$

Verify our claims that the function $F_{b,c}$ is monotone and continuous. When is it strict?

# 3 Constructions on Domains

Using Kleene's fixed point theorem, we now know how we can compute fixed points, given we are dealing with continuous functions in a domain. But this is only useful if we know how to construct interesting domains and continuous functions.

Thus, in this section we give various ways of building domains and continuous functions, concentrating on the ones that will be needed for a denotational semantics of the programming language PCF studied in the second half of the course. Recall that to specify a cpo one must *define* a set equipped with a binary relation and then *prove* that

(i) the relation is a partial order;

(ii) lubs exist for all chains in the partially ordered set.

Furthermore, for the cpo to be a domain, one additionally must show that

(iii) there is a least element.

Note that since lubs of chains and least elements are unique if they exist, a cpo or domain is completely determined by its underlying set and partial order. In what follows we will give various recipes for constructing cpos and domains and leave as an exercise the task of checking that they are indeed domains, *i.e.* that properties (i)–(iii) do hold.

## 3.1 Flat domains

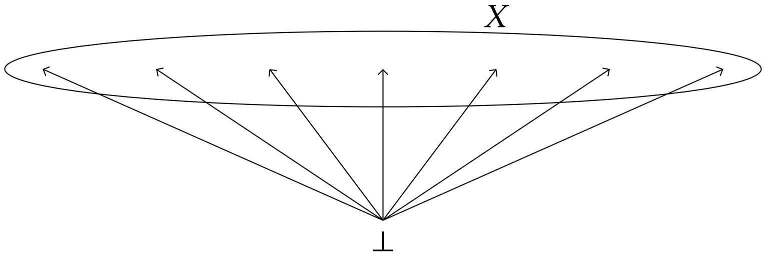
In order to model the PCF ground types `nat` and `bool`, we will use the notion of *flat domain*, that we already encountered in Example 7.

**Definition 11 (Discrete cpo)** For any set $X$, the relation of equality makes $(X, =)$ into a partial order, called the *discrete* order with underlying set $X$. This poset is in fact a cpo. *

**Definition 12 (Flat domain)** The *flat domain* on a set $X$ is defined by:

- its underlying set $X \uplus \{\bot\}$ (*i.e.* $X$ $\bot$ extended with a new element $\bot$);
- $x \sqsubseteq x'$ if either $x = \bot$ or $x = x'$. *

The Hasse diagram of a flat domain looks as follows (the only edges relate $\bot$ to the elements of $X$):



The following instances of continuous functions between flat domains will also be needed for the denotational semantics of PCF.

**Proposition 7 (Flat domain lifting)** *Let $f : X \rightharpoonup Y$ be a partial function between two sets. Then*

$$\begin{array}{rcl} f_\perp : X_\perp & \to & Y_\perp \\ d & \mapsto & \begin{cases} f(d) & \textit{if } d \in X \textit{ and } f \textit{ is defined at } d \\ \perp & \textit{if } d \in X \textit{ and } f \textit{ is not defined at } d \\ \perp & \textit{if } d = \perp \end{cases} \end{array}$$

*defines a continuous function between the corresponding flat domains.* *

## 3.2 Products of domains

**Definition 13 (Binary product of two orders)** The *product* of two posets $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$ has underlying set

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

and partial order $\sqsubseteq$ defined by

$$(d_1, d_2) \sqsubseteq (d_1', d_2') \overset{\text{def}}{\Leftrightarrow} d_1 \sqsubseteq_1 d_1' \wedge d_2 \sqsubseteq_2 d_2'$$ *

**Proposition 8 (Products preserve lubs and least element)** *lubs of chains are computed componentwise:*

$$\bigsqcup_{n \geq 0} (d_{1,n}, d_{2,n}) = (\bigsqcup_{i \geq 0} d_{1,i}, \bigsqcup_{j \geq 0} d_{2,j}).$$

*If $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$ have least elements, so does $(D_1 \times D_2, \sqsubseteq)$ with*

$$\perp_{D_1 \times D_2} = (\perp_{D_1}, \perp_{D_2})$$

*Thus, the product of two cpos (respectively domains) is a cpo (respectively domain).* *

**Proposition 9 (Projections and pairing)** *Let $D_1$ and $D_2$ be cpos. The* projections

$$\begin{array}{rcl} \pi_1 : D_1 \times D_2 & \to & D_1 \\ (d_1, d_2) & \mapsto & d_1 \end{array} \qquad \begin{array}{rcl} \pi_2 : D_1 \times D_2 & \to & D_2 \\ (d_1, d_2) & \mapsto & d_2 \end{array}$$

*are continuous functions. If $f_1 : D \to D_1$ and $f_2 : D \to D_2$ are continuous functions from a cpo D, then the* pairing *function*

$$\begin{array}{rcl} \langle f_1, f_2 \rangle : D & \to & D_1 \times D_2 \\ d & \mapsto & (f_1(d), f_2(d)) \end{array}$$

*is continuous.* *

Proof Continuity of these functions follows immediately from the characterisation of lubs of chains in $D_1 \times D_2$ given in Proposition 8. □

**Proposition 10** *Domain conditional For each domain D the conditional function*

$$\begin{array}{rcl} \mathrm{if}: \ \mathbb{B}_\perp \times (D \times D) & \to & D \\ (x, d) & \mapsto & \begin{cases} \pi_1(d) & \textit{if } x = \mathrm{true} \\ \pi_2(d) & \textit{if } x = \mathrm{false} \\ \perp_D & \textit{if } x = \perp \end{cases} \end{array}$$

*is continuous.* $\ast$

We can generalize this construction to not just a binary product, but any product.

**Definition 14 (General product of posets)** Given a set $I$, suppose that for each $i \in I$ we are given a cpo $(D_i, \sqsubseteq_i)$. The *product* of this whole family of cpos has

- underlying set equal to the $I$-fold cartesian product, $\prod_{i \in I} D_i$, of the sets $D_i$ – so it consists of all functions $p$ defined on $I$ and such that the value of $p$ at each $i \in I$ is some $p(i) \in D_i$;
- partial order $\sqsubseteq$ defined componentwise, that is

$$p \sqsubseteq p' \overset{\mathrm{def}}{\Leftrightarrow} \forall i \in I.\, p(i) \sqsubseteq_i p'(i).$$

$\ast$

**Remark 3** The usual binary product can be seen as a special case of the above, when taking $I$ to be a two-element set, for instance $\mathbb{B}$. Indeed, an element $p \in \prod_{i \in \mathbb{B}} D_i$ corresponds to $(p \text{ true}, p \text{ false}) \in D_{\mathrm{true}} \times D_{\mathrm{false}}$.

**Proposition 11 (General products of cpos and domains)** *As for the binary product, lubs in $(\prod_{i \in I} D_i, \sqsubseteq)$ can be computed componentwise: if $p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \dots$ is a chain in the product cpo, its lub is the function mapping each i $\in I$ to the lub in $D_i$ of the chain $p_0(i) \sqsubseteq p_1(i) \sqsubseteq p_2(i) \sqsubseteq \dots$. Said otherwise,*

$$\left( \bigsqcup_{n \geq 0} p_n \right)(i) = \bigsqcup_{n \geq 0} p_n(i) \qquad (i \in I).$$

*In particular, for each i $\in I$ the ith* projection function

$$\begin{array}{rcl} \pi_i : \ \prod_{j \in I} D_j & \to & D_i \\ p & \mapsto & p(i) \end{array}$$

*is continuous.*

*If all the $D_i$ are domains, then so is their product – the least element being the function mapping each i $\in I$ to the least element of $D_i$.* $\ast$

**Proposition 12 (Functions of two arguments)** *Let E, F and G be cpos. A function $f : (D \times E) \to F$ is monotone if and only if it is monotone in each argument separately:*

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

*Moreover, it is continuous if and only if it preserves lubs in each argument separately:*

$$f(\bigsqcup_{m\geq 0} d_m\,,\, e) = \bigsqcup_{m\geq 0} f(d_m, e)$$
$$f(d\,,\, \bigsqcup_{n\geq 0} e_n) = \bigsqcup_{n\geq 0} f(d, e_n).$$

*

Proof The 'only if' directions are straightforward. Indeed, observe that if $d \sqsubseteq d'$ then $(d,e) \sqsubseteq (d',e)$, and

$$(\bigsqcup_{m\geq 0} d_m, e) = \bigsqcup_{m\geq 0} (d_m, e)$$

as well as the companion facts for the right argument.

For the 'if' direction, suppose first that $f$ is monotone in each argument separately. Then given $(d,e) \sqsubseteq (d',e')$ in $D \times E$, by definition of the partial order on the binary product we have $d \sqsubseteq d'$ in $D$ and $e \sqsubseteq e'$ in $E$. Hence,

$$\begin{aligned} f(d,e) &\sqsubseteq f(d',e) && \text{by monotonicity in the first argument} \\ &\sqsubseteq f(d',e') && \text{by monotonicity in the second argument} \end{aligned}$$

and therefore by transitivity, $f(d,e) \sqsubseteq f(d',e')$, as required for monotonicity of $f$.

Now suppose $f$ is continuous in each argument separately. Then given a chain $(d_0, e_0) \sqsubseteq (d_1, e_1) \sqsubseteq (d_2, e_2) \sqsubseteq \ldots$ in the binary product, we have

$$\begin{aligned} f\left(\bigsqcup_{n\geq 0}(d_n, e_n)\right) &= f\left(\bigsqcup_{i\geq 0} d_i\,, \bigsqcup_{j\geq 0} e_j\right) && \text{lubs are componentwise (Prop. 8)} \\ &= \bigsqcup_{i\geq 0} f(d_i, \bigsqcup_{j\geq 0} e_j) && \text{by continuity in the first argument} \\ &= \bigsqcup_{i\geq 0}\left(\bigsqcup_{j\geq 0} f(d_i, e_j)\right) && \text{by continuity in the second argument} \\ &= \bigsqcup_{n\geq 0} f(d_n, e_n) && \text{by diagonalisation (Prop. 5)} \end{aligned}$$

as required for continuity of $f$. □

## 3.3 Function domains

The set of continuous functions between two cpos/domains can itself be made into a cpo/domain. The terminology ‘exponential’ cpo/domain is sometimes used instead of ‘function’ cpo/domain.

**Definition 15 (Cpo/domain of continuous functions)** Given two cpos $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$, the *function cpo* $(D \to E, \sqsubseteq)$ has underlying set

$$\{f : D \to E \mid \text{ is a } \textit{continuous} \text{ function}\}$$

equipped with the pointwise order:

$$f \sqsubseteq f' \overset{\text{def}}{\Leftrightarrow} \forall d \in D.\ f(d) \sqsubseteq_E f'(d).$$

As for products, lubs and least elements always exist and are computed ‘argumentwise’, using lubs in $E$:

$$\bot_{D\to E}(d) = \bot_E \qquad\qquad \left(\bigsqcup_{n\geq 0} f_n\right)(d) = \bigsqcup_{n\geq 0} f_n(d)$$

*

Proof The proof that argumentwise least elements and lubs are themselves least elements and lubs is essentially similar to that for products, see Proposition 8.

However, we should additionally show that the lub of a chain of functions, $\bigsqcup_{n\geq 0} f_n$, is continuous. The proof uses the ‘interchange law’ of Proposition 5. Given a chain in $D$,

$$\begin{aligned}
(\bigsqcup_{n\geq 0} f_n)((\bigsqcup_{m\geq 0} d_m)) &= \bigsqcup_{n\geq 0}(f_n(\bigsqcup_{m\geq 0} d_m)) && \text{definition of } \bigsqcup_{n\geq 0} f_n \\
&= \bigsqcup_{n\geq 0}(\bigsqcup_{m\geq 0} f_n(d_m)) && \text{continuity of each } f_n \\
&= \bigsqcup_{m\geq 0}(\bigsqcup_{n\geq 0} f_n(d_m)) && \text{interchange law} \\
&= \bigsqcup_{m\geq 0}((\bigsqcup_{n\geq 0} f_n)(d_m)) && \text{definition of } \bigsqcup_{n\geq 0} f_n.
\end{aligned}$$

□

Note that we actually did not use the fact that $D$ is a cpo/domain: it suffices that $E$ is. Intuitively, the structure of the function space is inherited from the structure of $E$, since it is pointwise.

All the familiar operations on functions actually lift to the cpo structure, in the sense that they are monotone and continuous, when see as functions from/into the relevant function cpos.

**Proposition 13 (Evaluation)** *Given cpos $D$ and $E$, the* evaluation *function*

$$\begin{array}{rccl} \mathrm{eval}: & (D \to E) \times D & \to & E \\ & (f, d) & \mapsto & f(d) \end{array}$$

*is continuous.* *

**Proposition 14 (Currying[1])** *Given any continuous function $f : D' \times D \to E$ (with $D$, $D'$ and $E$ cpos), for each $d' \in D'$ the function $\lambda d \in D.\ f(d', d)$ is continuous (Proposition 12) and hence determines an element of the function cpo $D \to E$ that we denote by $\mathrm{cur}(f)(d')$. Then*

$$\begin{array}{rccl} \mathrm{cur}(f): & D' & \to & (D \to E) \\ & d' & \mapsto & \lambda d \in D.\ f(d', d) \end{array}$$

*is well-defined (i.e. $\lambda d \in D.f(d', d)$ is a continuous function) and continuous.* *

**Proposition 15 (Continuity of composition)** *For cpos $D, E, F$, the composition function circ defined by*

$$\begin{array}{rccl} \circ: & \big((E \to F) \times (D \to E)\big) & \longrightarrow & (D \to F) \\ & (f, g) & \mapsto & \lambda d \in D.\ g(f(d)) \end{array}$$

*is a well-defined continuous function.* *

Proof For continuity of eval note that

$$\begin{array}{rll} \mathrm{eval}(\bigsqcup\limits_{n\geq 0}(f_n, d_n)) & = \mathrm{eval}(\bigsqcup\limits_{i\geq 0} f_i, \bigsqcup\limits_{j\geq 0} d_j) & \text{lubs in products are componentwise} \\ & = (\bigsqcup\limits_{i\geq 0} f_i)(\bigsqcup\limits_{j\geq 0} d_j) & \text{by definition of eval} \\ & = \bigsqcup\limits_{i\geq 0} f_i(\bigsqcup\limits_{j\geq 0} d_j) & \text{lubs in function cpos are argumentwise} \\ & = \bigsqcup\limits_{i\geq 0}\bigsqcup\limits_{j\geq 0} f_i(d_j) & \text{by continuity of each } f_i \\ & = \bigsqcup\limits_{n\geq 0} f_n(d_n) & \text{by diagonalisation} \\ & = \bigsqcup\limits_{n\geq 0} \mathrm{eval}(f_n, d_n) & \text{by definition of eval.} \end{array}$$

[1] The name 'currying' is given to this operation in honour of the logician H. B. Curry, a pioneer of combinatory logic and lambda calculus.

The continuity of each $\mathrm{cur}(f)(d')$ and then of $\mathrm{cur}(f)$ follows immediately from the fact that lubs of chains in $D_1 \times D_2$ can be calculated componentwise.

The continuity of $g \circ f$ is a direct consequence of that of $g$ and $f$. Continuity of $\circ$ again follows directly from diagonalisation. □

More interestingly, if $D$ is a domain then by Kleene's fixed point theorem (Theorem 6) we know that each continuous function $f \in (D \to D)$ possesses a least fixed point, $\mathrm{fix}(f) \in D$.

**Proposition 16 (Continuity of the fixed point operator)** *The function*

$$\mathrm{fix}\colon\ (D \to D) \ \to\ D \tag{*}$$

*is continuous.*

Proof We must first prove that $\mathrm{fix}\colon (D \to D) \to D$ is a monotone function. Suppose $f_1 \sqsubseteq f_2$ in the function domain $D \to D$. We have to prove $\mathrm{fix}(f_1) \sqsubseteq \mathrm{fix}(f_2)$. But:

$$\begin{aligned} f_1(\mathrm{fix}(f_2)) &\sqsubseteq f_2(\mathrm{fix}(f_2)) && \text{since } f_1 \sqsubseteq f_2 \\ &\sqsubseteq \mathrm{fix}(f_2) && \text{because } \mathrm{fix}(f_2) \text{ is a pre-fixed point.} \end{aligned}$$

So $\mathrm{fix}(f_2)$ is a pre-fixed point for $f_1$ and hence by minimality of $\mathrm{fix}(f_1)$ amongst pre-fixed points, we have $\mathrm{fix}(f_1) \sqsubseteq \mathrm{fix}(f_2)$, as required.

Turning now to the preservation of lubs of chains, suppose $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots$ in $D \to D$. Recalling Remark 2, we just have to prove that

$$\mathrm{fix}(\bigsqcup_{n\geq 0} f_n) \sqsubseteq \bigsqcup_{n\geq 0} \mathrm{fix}(f_n)$$

and by the minimality of the least pre-fixed point, for this it suffices to show that $\bigsqcup_{n\geq 0} \mathrm{fix}(f_n)$ is a pre-fixed point for the function $\bigsqcup_{n\geq 0} f_n$. This is the case because:

$$\begin{aligned} (\bigsqcup_{m\geq 0} f_m)(\bigsqcup_{n\geq 0} \mathrm{fix}(f_n)) &= \bigsqcup_{m\geq 0} f_m(\bigsqcup_{n\geq 0} \mathrm{fix}(f_n)) && \text{function lubs are argumentwise} \\ &= \bigsqcup_{m\geq 0} \bigsqcup_{n\geq 0} f_m(\mathrm{fix}(f_n)) && \text{by continuity of each } f_m \\ &= \bigsqcup_{k\geq 0} f_k(\mathrm{fix}(f_k)) && \text{by diagonalisation} \end{aligned}$$

Moreover, each $\mathrm{fix}(f_k)$ is a pre-fixed point, *i.e.* $f_k(\mathrm{fix}(f_k)) \sqsubseteq \mathrm{fix}(f_k)$, and so by monotony of lubs (Proposition 2),

$$(\bigsqcup_{m\geq 0} f_m)(\bigsqcup_{n\geq 0} \mathrm{fix}(f_n)) = \bigsqcup_{k\geq 0} f_k(\mathrm{fix}(f_k)) \sqsubseteq \bigsqcup_{k\geq 0} \mathrm{fix}(f_k)$$

as required. □

## 3.4 Exercises

**Exercise 4** Verify that the constructions given in Definition 12 (flat domains), Definition 13 (binary products), and Definition 14 (general product) indeed form domains (for the latter two, this is respectively Proposition 8 and Proposition 11).

Verify that the flat domain lifting of functions (Proposition 7) and the $\mathtt{if}$ function (Proposition 10) are continuous.

**Exercise 5** Let $X$ and $Y$ be sets and $X_\bot$ and $Y_\bot$ the corresponding flat domains (Definition 12). Show that a function $f\colon X_\bot \to Y_\bot$ is continuous if and only if one of the following alternatives holds:

(a) $f$ is strict, *i.e.* $f(\bot) = \bot$;

(b) $f$ is constant, *i.e.* $\forall x, x' \in X.\ f(x) = f(x')$.

**Exercise 6** Let $\{\top\}$ be a one-element set and $\{\top\}_\bot$ the corresponding flat domain. Let $\Omega$ be the domain of 'vertical natural numbers', defined in Example 6. Show that the function domain $(\Omega \to \{\top\}_\bot)$ is in bijection with $\Omega$.

**Exercise 7** Prove Propositions 14 and 15, *i.e.* that currying and composition are continuous.

# 4 Scott Induction

## 4.1 Reasoning on fixed points

We now know how to construct fixed points using Kleene's fixed point theorem (Theorem 6), provided we are considering a continuous function between domains. Moreover, in Section 3, we have given a handful of way to create new interesting domains (flat domains (Definition 12), product domains (Definitions 13 and 14), and function domains (Definition 15)), and continuous functions between those.

We are missing an ingredient, however: how to *reason* on fixed points, *i.e.* prove properties of the fixed points we know how to construct. Since Kleene's fixed point theorem gives an explicit construction of $\mathrm{fix}(f)$ as $\bigsqcup_n f^n(\bot)$, we can reason using this construction. To show $\Phi(\mathrm{fix}(f))$ for some property $\Phi$, this would typically go as follows. First, show that $\Phi(\bot)$ holds, and that if $\Phi(f^n(\bot))$ holds, then $\Phi(f^{n+1}(\bot))$ holds. By induction on $\mathbb{N}$, we get that for all $n \in \mathbb{N}$, $\Phi(f^n(\bot))$ holds. If moreover for any chain $d_0 \sqsubseteq d_1 \sqsubseteq \dots$ such that $\forall n \in \mathbb{N}.\ \Phi(d_n)$, we have $\Phi\left(\bigsqcup_n d_n\right)$, in particular we get $\Phi(\bigsqcup_n f^n(\bot))$, *i.e.* $\Phi(\mathrm{fix}(f))$.

We can package this common reasoning into a form of induction principle, called 'Scott induction'.

**Theorem 17 (Scott induction)** *Let $D$ be a domain, $f\colon D \to D$ be a continuous function and $S \subseteq D$ be a subset of $D$. If the set $S$*

*(i) contains $\bot$,*

*(ii) is stable under $f$, i.e. $f(S) \subseteq S$,*

*(iii) is chain-closed, i.e. the lub of any chain of elements of $S$ is also in $S$,*

*then* $\mathrm{fix}(f) \in S$. $\ast$

**Remark 4** A set that satisfies the first and third item, *i.e.* that contains $\bot$ and is chain-closed, is sometimes called an *admissible* set.

We expressed Scott induction in terms of a subset, but it can be alternatively phrased in terms of a property $\Phi$, by taking $S$ to be $\{d \in D \mid \Phi(d)\}$. Accordingly, we will use the terms chain-closed, admissible and stable under $f$ for properties, too. $\ast$

**Example 14** Consider the domain $\Omega$ of 'vertical natural numbers' pictured in Example 6. Then

- any *finite* subset of $\Omega$ is chain-closed;
- $\{0, 2, 4, 6, \dots\}$ is not a chain-closed subset of $\Omega$;
- $\{0, 2, 4, 6, \dots\} \cup \{\omega\}$ is a chain-closed (indeed, is an admissible) subset of $\Omega$.

## 4.2 Building chain-closed subsets

The difficulty with applying Scott induction usually lies in identifying an appropriate subset $S$; *i.e.* in finding a suitably strong 'induction hypothesis'. Luckily, we can show that a large family of sets are at least chain-closed.

**Proposition 18 (Basic relations)** *Let $D$ be a cpo. The subsets*

$$\{(x, y) \in D \times D \mid x \sqsubseteq y\} \qquad \textit{and} \qquad \{(x, y) \in D \times D \mid x = y\}$$

*of $D \times D$ are chain-closed.*

*Said otherwise, the predicates $x \sqsubseteq y$ and $x = y$ on $D \times D$ determine chain-closed sets.* $\ast$

**Proposition 19 (Inverse image and substitution)** *Let $f : D \to E$ be a continuous function between cpos $D$ and $E$. Suppose $S$ is a chain-closed subset of $E$. Then the inverse image*

$$f^{-1}S = \{x \in D \mid f(x) \in S\}$$

*is a chain-closed subset of $D$.*

*Said otherwise, if a property $P(y)$ on $E$ determines a chain-closed subset of $E$ and $f : D \to E$ is a continuous function, then the property $P(f(x))$ on $D$ determines a chain-closed subset of $D$.* $\ast$

**Proposition 20 (Logical operations)** *Let $D$ be a cpo. Let $S \subseteq D$ and $T \subseteq D$ be chain-closed subsets of $D$. Then $S \cup T$ and $S \cap T$ are chain-closed subsets.*

*In terms of properties, if $P(x)$ and $Q(x)$ determine chain-closed subsets of $D$, then so do $P \vee Q$ and $P \wedge Q$.* $\ast$

Actually, if more generally $(S_i)_{i \in I}$ is a family of chain-closed subsets of $D$ indexed by a set $I$, then $\bigcap_{i \in I} S_i$ is a chain-closed subset of $D$. As a consequence, we get the following.

**Proposition 21 (Universal quantification)** *If a property $P(x, y)$ determines a chain-closed subset of $D \times E$, then the property $\forall x \in D.\ P(x, y)$ determines a chain-closed subset of $E$.* $\ast$

Proof This is because

$$\begin{aligned}\{y \in E \mid \forall x \in D.\ P(x, y)\} &= \bigcap_{d \in D} \{y \in E \mid P(d, y)\} \\ &= \bigcap_{d \in D} {f_d}^{-1}\{(x, y) \in D \times E \mid P(x, y)\}\end{aligned}$$

where $f_d : E \to D \times E$ is the continuous function defined as $f_d(y) = (d, y)$ for every $d \in D$. □

Combining these properties, we obtain that any formula built-up as a universal quantification over several variables of conjunctions and disjunctions of basic properties of the form $f(x_1, \cdots, x_k) \sqsubseteq g(x_1, \cdots, x_l)$ or $f(x_1, \cdots, x_k) = g(x_1, \cdots, x_l)$, where $f$ and $g$ are continuous, will determine a chain-closed subset of the product cpo appropriate to the non-quantified variables. Some $x_i$ can also be constants, as was used in the proof of Proposition 21.

Note, however, that infinite unions of chain-closed subsets need not be chain-closed. Indeed, any set is a union of finite subsets, which are always chain-closed – so if infinite unions of chain-closed subsets were chain-closed, all sets would be chain-closed. Accordingly, we cannot in general build chain-closed subsets with existential quantifications. Similarly, the complement of a chain-closed set (or the logical negation of a formula), also need not be chain-closed.

## 4.3 Using Scott induction

**Example 15 (Revisiting the least fixed point property)** Let $D$ be a domain and let $f : D \to D$ be a continuous function, $d \in D$, and assume $f(d) \sqsubseteq d$, *i.e.* $d$ is a pre-fixed point of $f$.

Define the *downset* of $d$ as follows:

$$d \downarrow \overset{\text{def}}{=} \{x \in D \mid x \sqsubseteq d\}.$$

By the properties of the previous section, $d \downarrow$ is a chain-closed subset, which contains $\bot$. Moreover, we have

$$\begin{aligned} x \in d \downarrow &\Leftrightarrow x \sqsubseteq d \\ &\Rightarrow f(x) \sqsubseteq f(d) \\ &\Rightarrow f(x) \sqsubseteq d \\ &\Rightarrow f(x) \in d \downarrow \end{aligned}$$

Thus, $d \downarrow$ is stable under $f$. By Scott induction, $\mathrm{fix}(f) \in d \downarrow$, *i.e.* $\mathrm{fix}(f) \sqsubseteq d$. ⋆

The next example shows that Scott's Induction Principle can be used for proving (the denotational version of) *partial correctness* assertions about programs, *i.e.* assertions of the form 'if the program terminates, then such-and-such a property holds of the results'. By contrast, a *total* correctness assertion would be 'the program does terminate and such-and-such a property holds of the results'. Because Scott Induction can only be applied for properties $\Phi$ for which $\Phi(\bot)$ holds, it is not so useful for proving total correctness.

**Example 16** Let $F$ be the continuous function defined in Section 1.3, whose least fixed point is the denotation of the command

$$C \overset{\text{def}}{=} \texttt{while } X > 0 \text{ do } (Y := X * Y; X := X - 1)$$

We will use Scott induction to prove

$$\forall x.\ \forall y \geq 0.\ \mathrm{fix}(F)[X \mapsto x, Y \mapsto y] \Downarrow \implies (\mathrm{fix}(F)[X \mapsto x, Y \mapsto y])(Y) \geq 0$$

where for $w \in D = \mathrm{State} \rightharpoonup \mathrm{State}$ we write $w(s) \Downarrow$ to mean 'the partial function $w$ is defined at the state $s$'. In words, we want to prove that if the command $C$ is run in a state where the variable $Y$ has a non-negative value, and the execution terminates, then the value of $Y$ at the end of the execution is still non-negative. $*$

Proof Let $D = \mathrm{State} \rightharpoonup \mathrm{State}$ and $S$ be the subset given by

$$S \overset{\text{def}}{=} \{w \in D \mid \forall x. \forall y \geq 0. (w[X \mapsto x, Y \mapsto y] \Downarrow) \Rightarrow (w[X \mapsto x, Y \mapsto y])(Y) \geq 0\}$$

Since the precondition of the implication always holds for $\bot_D$ which is the nowhere-defined function, $\bot_D \in S$.

Moreover, we have that

$$S = \bigcap_{x \in \mathbb{Z}} \bigcap_{y \in \mathbb{N}} (w[X \mapsto x, Y \mapsto Y] \Downarrow) \Rightarrow w[X \mapsto x, Y \mapsto y](Y) \geq 0$$

Given a fixed $x, y$, the set of $w$ such that

$$(w[X \mapsto x, Y \mapsto Y] \Downarrow) \Rightarrow w[X \mapsto x, Y \mapsto y](Y) \geq 0$$

is chain-closed. Indeed, given a chain $(w_i)_{i \in \mathbb{N}}$ in that set, there are two possibilities. Either the value of all $w_i[X \mapsto x, Y \mapsto y]$ is undefined, in which case this is also true of their lub. Or for some index $i$ the state $w_i[X \mapsto x, Y \mapsto y]$ is defined, say some state $s$ such that $s(Y) \geq 0$. But then for all $j > i$, also $w_j[X \mapsto x, Y \mapsto y] = s$, and so this is also true of the lub. Thus, $S$ is an intersection of chain-closed sets, and is chain-closed.

Finally, $S$ is stable under $F$. Indeed, let us thus assume we are given $w \in S$, and suppose moreover that $x \in \mathbb{Z}$, $y \geq 0$ and $F(w)[X \mapsto x, Y \mapsto y] \Downarrow$. In the case where $x \leq 0$, we simply have

$$F(w)[X \mapsto x, Y \mapsto y](Y) = y \geq 0.$$

Otherwise, $x > 0$, and we have

$$F(w)[X \mapsto x, Y \mapsto y](Y) = x \cdot y \geq 0$$

since by assumption $x, y \geq 0$. Thus, $F(w) \in S$, as claimed.

Since $S$ is admissible and stable under $F$, we can conclude by Scott induction that $\mathrm{fix}(F) \in S$, as desired. □

**Example 17** Let $D$ be a domain and let $f, g : D \to D$ be continuous functions such that $f \circ g \sqsubseteq g \circ f$. Then,

$$f(\bot) \sqsubseteq g(\bot) \implies \mathrm{fix}(f) \sqsubseteq \mathrm{fix}(g) \ . \qquad *$$

Proof Consider the property $\Phi(x) \equiv \big(f(x) \sqsubseteq g(x)\big)$ on $D$. By assumption, $\Phi(\bot)$ holds. Moreover, by the properties of Section 4.2, $\Phi$ is chain-closed. Since

$$f(x) \sqsubseteq g(x) \Rightarrow g(f(x)) \sqsubseteq g(g(x)) \Rightarrow f(g(x)) \sqsubseteq g(g(x))$$

$\Phi$ is also stable under $g$.

Thus, by Scott induction, we have that

$$f(\mathrm{fix}(g)) \sqsubseteq g(\mathrm{fix}(g)) = \mathrm{fix}(g) \ .$$

Hence, $\mathrm{fix}(g)$ is a pre-fixed point of $f$, and $\mathrm{fix}(f) \sqsubseteq \mathrm{fix}(g)$ as claimed. □

## 4.4 Exercises

**Exercise 8** Show the properties of Section 4.2, *i.e.* that equality and $\sqsubseteq$ give basic chain-closed sets, that chain-closed sets are stable under inverse image (by continuous functions), and that binary union and arbitrary intersection of chain-closed sets are again chain-closed.

**Exercise 9** Give an example of a subset $S \subseteq D \times D'$ of a product cpo that is not chain-closed, but which satisfies both of the following:

(i) for all $d \in D$, $\{d' \mid (d, d') \in S\}$ is a chain-closed subset of $D'$; and

(ii) for all $d' \in D'$, $\{d \mid (d, d') \in S\}$ is a chain-closed subset of $D$.

[Hint: consider $D = D' = \Omega$, the cpo in Example 6.]

(Compare this with the property of continuous functions given in Proposition 12, *i.e.* that continuity of functions from $D \times D'$ is equivalent to continuity in each argument separately.)

# 5 PCF

The language PCF ('Programming Computable Functions') is a simple functional programming language that has been used extensively as an example language in the development of the theory of both denotational and operational semantics (and the relationship between the two). Its syntax was introduced by Dana Scott *circa* 1969 as part of a 'Logic of Computable Functions'[2] and was studied as a programming language in a highly influential paper by Plotkin [3].

## 5.1 Terms and types

### Syntax

Types: $\tau ::= \mathtt{nat} \mid \mathtt{bool} \mid \tau \to \tau$

Terms:

$$\begin{array}{lll} t & ::= & 0 \mid \mathtt{succ}(t) \mid \mathtt{pred}(t) \mid \\ & & \mathtt{true} \mid \mathtt{false} \mid \mathtt{zero?}(t) \mid \mathtt{if}\ t\ \mathtt{then}\ t\ \mathtt{else}\ t \\ & & x \mid \mathtt{fun}\, x{:}\,\tau.\, t \mid t\, t \mid \mathtt{fix}(t) \end{array}$$

Figure 1: Syntax of PCF

The *types* and *terms* of the PCF language are defined in Fig. 1. The intended meaning of the syntactic constructions is as follows.

- `nat` is the type of the natural numbers, $0, 1, 2, 3, \ldots$. In PCF these are generated from 0 by repeated application of the successor operation, `succ`, which adds 1 to its argument. The predecessor operation `pred` subtracts 1 from strictly positive natural numbers (and is undefined at 0).
- `bool` is the type of booleans, `true` and `false`. The operation `zero?` tests whether its argument is zero or strictly positive and returns `true` or `false` accordingly. The *conditional* expression $\mathtt{if}\ b\ \mathtt{then}\ t\ \mathtt{else}\ t'$ behaves like either $t$ or $t'$ depending upon whether $b$ evaluates to `true` or `false` respectively.
- A PCF variable, $x$, stands for an expression.
- $\tau \to \tau'$ is the type of (partial) functions taking a single argument of type $\tau$ and (possibly) returning a result of type $\tau'$. $\mathtt{fun}\, x{:}\,\tau.\, t$ is the notation we will use for function abstraction (*i.e.* $\lambda$-abstraction) in PCF. The application of function $f$ to

[2] This logic was the stimulus for the development of the ML language and LCF system for machine-assisted proofs by Milner, Gordon *et al.*—see Paulson [1]; Scott's original work was eventually published as Scott [2].

argument $u$ is indicated by $t\ u$. The scope of a function abstraction extends as far to the right of the dot as possible and function application associates to the left (*i.e.* $f\ t\ u$ means $(f\ t)\ u$, not $f\ (t\ u)$).

- The expression $\mathtt{fix}(t)$ indicates an element $x$ *recursively defined* by $x = t\ x$. Thus, the following recursive OCaml function

  ```
  let rec f (x1 : α1) ... (xn : αn) : τ := p
  ```

  corresponds to $\mathtt{fix}(\mathtt{fun}\, x_1{:}\, \alpha_1.\ (\ldots(\mathtt{fun}\, x_n{:}\, \alpha_n.\ p)))$. The fix syntax has the advantage of being as expressive, but easier to manipulate in theory. The λ-calculus equivalent to $\mathtt{fix}(f)$ is $Y\ f$, where $Y$ is a suitable fixed point combinator.

All in all, PCF is basically a very toy version of a language from the ML family. The main difference is that PCF is *pure*, meaning that there is no state that changes during expression evaluation. So in particular variables are 'identifiers' standing for a fixed expression to be manipulated and passed around, rather than 'program variables' whose contents may get mutated during evaluation.

### Variables and substitution

The fact that $\mathtt{fun}\, x{:}\, \tau.\ t$ binds the variable $x$ means that the usual phenomena around variable binding, that were already covered in Part IB – Computation Theory (for λ-calculus) and Part IB – Semantics of Programming Languages (for other functional languages).

Just as in these courses, we consider PCF terms up to α-equivalence of bound variables. That is, $\mathtt{fun}\, x{:}\, \tau.\ x$ and $\mathtt{fun}\, y{:}\, \tau.\ y$ denote the *same* PCF program. We will also use the substitution operation, and we will denote the substitution of $u$ for $x$ as $t[u/x]$.

Since these are not the main focus here, we refer to these courses for details.

### Typing

**Definition 16** Contexts A context, usually denoted $\Gamma$, that is, a partial function from variables to types. The empty context is denoted $\cdot$, and context extension $\Gamma, x{:}\, \tau$ is the context that maps $x$ to $\tau$ and acts on other variables like $\Gamma$. ⋆

**Remark 5** Alternatively, we can see contexts as (finite) lists of pairs of a variable and a type, that is, as given by the following grammar:

$$\Gamma ::= \cdot \mid \Gamma, x{:}\, \tau$$

The view of contexts as partial functions is slightly easier to manipulate informally, which is why we stick with this presentation. ⋆

$\boxed{\Gamma \vdash t : \tau}$ The term $t$ has type $\tau$ in context $\Gamma$

$$\textsc{Zero}\ \frac{}{\Gamma \vdash \mathtt{0} : \mathtt{nat}} \qquad \textsc{Succ}\ \frac{\Gamma \vdash t : \mathtt{nat}}{\Gamma \vdash \mathtt{succ}(t) : \mathtt{nat}} \qquad \textsc{Pred}\ \frac{\Gamma \vdash t : \mathtt{nat}}{\Gamma \vdash \mathtt{pred}(t) : \mathtt{nat}}$$

$$\textsc{True}\ \frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}} \qquad \textsc{False}\ \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}}$$

$$\textsc{IsZ}\ \frac{\Gamma \vdash t : \mathtt{nat}}{\Gamma \vdash \mathtt{zero?}(t) : \mathtt{bool}} \qquad \textsc{If}\ \frac{\begin{array}{c}\Gamma \vdash b : \mathtt{bool} \\ \Gamma \vdash t : \tau \qquad \Gamma \vdash t' : \tau\end{array}}{\Gamma \vdash \mathtt{if}\ b\ \mathtt{then}\ t\ \mathtt{else}\ t' : \tau}$$

$$\textsc{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{Fun}\ \frac{\Gamma, x{:}\sigma \vdash t : \tau}{\Gamma \vdash \mathtt{fun}\, x{:}\sigma.\, t : \sigma \to \tau}$$

$$\textsc{App}\ \frac{\Gamma \vdash f : \sigma \to \tau \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash f\ u : \tau} \qquad \textsc{Fix}\ \frac{\Gamma \vdash f : \tau \to \tau}{\Gamma \vdash \mathtt{fix}(f) : \tau}$$

Figure 2: Typing for PCF

PCF is a typed language: types are assigned to terms via the relation $\Gamma \vdash t : \tau$ defined in Fig. 2.

There is a subtle, but important difference with Part IB – Semantics of Programming Languages, to how we think of types. In Part IB, types were a way to ensure that "programs do not get stuck": the operational semantics was defined for *all terms*, and the safety property was proven afterwards, showing that all well-typed programs have a meaningful operational semantics, in that they either reduce to a value, or reduce forever. Here, we will *only* define the semantics of *well-typed terms*. The philosophy is that an ill-typed term does not really qualify as a program: it should be rejected by the compiler, and as such never executed. Therefore, there is no point in defining its semantics.

**Definition 17** We will write $\mathrm{PCF}_{\Gamma,\tau}$ for the set of terms of type $\tau$ in context $\Gamma$, *i.e.*

$$\mathrm{PCF}_{\Gamma,\tau} \overset{\mathrm{def}}{=} \{t \mid \Gamma \vdash t : \tau\}$$

and we simply write $\mathrm{PCF}_{\tau}$ for $\mathrm{PCF}_{\cdot,\tau}$ for terms of type $\tau$ in the empty context, *i.e.* well-typed terms closed terms. $\star$

**Proposition 22 (Typing is stable under substitution)** *If* $\Gamma \vdash t : \tau$ *and* $\Gamma, x{:}\tau \vdash t' : \tau'$ *both hold, then so does* $\Gamma \vdash t'[t/x] : \tau'$.

Proof This is a direct induction on typing derivations. □

## 5.2 Operational Semantics

We give the operational semantics of PCF in terms of an inductively defined relation of evaluation, given in Fig. 3. The results of evaluation are PCF terms of a particular form, called *values* (or 'canonical forms').

The only values of type `bool` are `true` and `false`. The values of type `nat` are unary representations of natural numbers, $\underline{n}$ ($n \in \mathbb{N}$), where

$$\begin{cases} \underline{0} & \overset{\text{def}}{=} 0 \\ \underline{n+1} & \overset{\text{def}}{=} \mathtt{succ}(\underline{n}). \end{cases}$$

Values at function types, being function abstractions $\mathtt{fun}\, x{:}\,\tau.\, t$, are more 'intensional' than those at the ground data types, since the body $t$ is an unevaluated PCF term.

**Remark 6 (Small and big step semantics)** This is the "big-step" presentation, which directly relates a program with a value, representing the possible results of evaluation, rather than relating programs via a "small-step" transition relation.[3] In our context this presentation is slightly easier to work with, but our proofs could be easily ported to the small-step presentation.

Let the relation $t \rightsquigarrow_\tau t'$ (for $t, t' \in \mathrm{PCF}_\tau$) be the one inductively defined in Fig. 4. Then one can show that for all $\tau$ and $t, v \in \mathrm{PCF}_\tau$ with $v$ a value

$$t \Downarrow_\tau v \Leftrightarrow t \rightsquigarrow_\tau^\star v$$

where $\rightsquigarrow_\tau^\star$ denotes the reflexive-transitive closure of the relation $\rightsquigarrow_\tau$. $*$

**Example 18 (The diverging term)** Proposition 23 shows that every closed typeable term evaluates to at most one value. Of course there are some typeable terms that do not evaluate to anything. We write $t \Uparrow_\tau$ (read '$t$ diverges') if $t : \tau$ and $\nexists V.\, t \Downarrow_\tau v$. For example

$$\Omega_\tau \overset{\text{def}}{=} \mathtt{fix}(\mathtt{fun}\, x{:}\,\tau.\, x)$$

satisfies $\Omega_\tau \Uparrow_\tau$.

[3] The kind which was used in Part IB – Semantics.

Values: $$v ::= \underbrace{\mathtt{0} \mid \mathtt{succ}(v)}_{\underline{n}} \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{fun}\, x{:}\,\tau.\, t$$

$\boxed{t \Downarrow_\tau v}$ The closed term $t \in \mathrm{PCF}_\tau$ evaluates to value $v$ at type $\tau$

$$\textsc{Val}\ \frac{\vdash v : \tau}{v \Downarrow_\tau v} \qquad \textsc{Succ}\ \frac{t \Downarrow_{\mathtt{nat}} v}{\mathtt{succ}(t) \Downarrow_{\mathtt{nat}} \mathtt{succ}(v)} \qquad \textsc{Pred}\ \frac{t \Downarrow_{\mathtt{nat}} \mathtt{succ}(v)}{\mathtt{pred}(t) \Downarrow_{\mathtt{nat}} v}$$

$$\textsc{ZeroZ}\ \frac{t \Downarrow_{\mathtt{nat}} \mathtt{0}}{\mathtt{zero?}(t) \Downarrow_{\mathtt{bool}} \mathtt{true}} \qquad \textsc{ZeroS}\ \frac{t \Downarrow_{\mathtt{nat}} \mathtt{succ}(v)}{\mathtt{zero?}(t) \Downarrow_{\mathtt{bool}} \mathtt{false}}$$

$$\textsc{IfT}\ \frac{b \Downarrow_{\mathtt{bool}} \mathtt{true} \qquad t_1 \Downarrow_\tau v}{\mathtt{if}\ b\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \Downarrow_\tau v} \qquad \textsc{IfF}\ \frac{b \Downarrow_{\mathtt{bool}} \mathtt{false} \qquad t_2 \Downarrow_\tau v}{\mathtt{if}\ b\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \Downarrow_\tau v}$$

$$\textsc{Fun}\ \frac{t \Downarrow_{\sigma \to \tau} \mathtt{fun}\, x{:}\,\sigma.\, t' \qquad t'[u/x] \Downarrow_\tau v}{t\ u \Downarrow_\tau v} \qquad \textsc{Fix}\ \frac{t\ (\mathtt{fix}(t)) \Downarrow_\tau v}{\mathtt{fix}(t) \Downarrow_\tau v}$$

Figure 3: Evaluation for PCF

$\boxed{t \rightsquigarrow_\tau t'}$ Closed term $t \in \mathrm{PCF}_\tau$ reduces to $t' \in \mathrm{PCF}_\tau$ at type $\tau$

$$\frac{t \rightsquigarrow_{\mathtt{nat}} t'}{\mathtt{op}(t) \rightsquigarrow_\tau \mathtt{op}(t')} \qquad \left(\begin{array}{rl}\text{where} & \mathtt{op}\text{ is }\mathtt{succ}\text{ or }\mathtt{pred}\text{ and }\tau\text{ is }\mathtt{nat}\\ \text{or} & \mathtt{op}\text{ is }\mathtt{zero?}\text{ and }\tau\text{ is }\mathtt{bool}\end{array}\right)$$

$$\mathtt{pred}(\mathtt{succ}(v)) \rightsquigarrow_{\mathtt{nat}} v \qquad\qquad \mathtt{zero?}(0) \rightsquigarrow_{\mathtt{bool}} \mathtt{true}$$

$$\mathtt{zero?}(\mathtt{succ}(v)) \rightsquigarrow_{\mathtt{bool}} \mathtt{false}$$

$$\frac{b \rightsquigarrow_{\mathtt{bool}} b'}{\mathtt{if}\ b\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \rightsquigarrow_\tau \mathtt{if}\ b'\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2}$$

$$\mathtt{if\ true\ then}\ t_1\ \mathtt{else}\ t_2 \rightsquigarrow_\tau t_1 \qquad\qquad \mathtt{if\ false\ then}\ t_1\ \mathtt{else}\ t_2 \rightsquigarrow_\tau t_2$$

$$\frac{t \rightsquigarrow_{\sigma\to\tau} t'}{t\ u \rightsquigarrow_\tau t'\ u} \qquad\qquad (\mathtt{fun}\,x{:}\,\sigma.\ t)\ u \rightsquigarrow_\tau t[u/x] \qquad\qquad \mathtt{fix}(t) \rightsquigarrow_\tau t\ (\mathtt{fix}(t))$$

Figure 4: Transition for PCF

For if for some $v$ there were a proof of $\mathtt{fix}(\mathtt{fun}\,x{:}\,\tau.\ x) \Downarrow_\tau v$, choose one of minimal height. This proof, call it $\mathcal{P}$, must look like

$$\cfrac{\cfrac{\mathtt{fun}\,x{:}\,\tau.\ x \Downarrow \mathtt{fun}\,x{:}\,\tau.\ x \qquad \overset{\mathcal{P}'}{\mathtt{fix}(\mathtt{fun}\,x{:}\,\tau.\ x) \Downarrow v}}{(\mathtt{fun}\,x{:}\,\tau.\ x)\ (\mathtt{fix}(\mathtt{fun}\,x{:}\,\tau.\ x)) \Downarrow v}}{\mathtt{fix}(\mathtt{fun}\,x{:}\,\tau.\ x) \Downarrow v}$$

where $\mathcal{P}'$ is a strictly shorter proof of $\mathtt{fix}(\mathtt{fun}\,x{:}\,\tau.\ x) \Downarrow_\tau v$, which contradicts the minimality of $\mathcal{P}$. $\ast$

**Example 19 (Partial recursive functions in PCF)** Although the PCF syntax is rather terse, the combination of increment, decrement, test for zero, conditionals, function abstraction and application, and fixed point recursion makes it Turing complete – in the sense that all partial recursive functions[4] can be coded. More precisely, for every partial recursive function $\phi$, there is a PCF term $\underline{\phi}$ such that for all $n \in \mathbb{N}$, if $\phi(n)$ is defined then $\underline{\phi}\ \underline{n} \Downarrow_{\mathtt{nat}} \underline{\phi(n)}$.

[4] See Part IB – Computation Theory.

For example, recall that the partial function $h\colon \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ defined by *primitive recursion* from $f\colon \mathbb{N} \rightharpoonup \mathbb{N}$ and $g\colon \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ satisfies that for all $x, y \in \mathbb{N}$

$$\begin{cases} h(x,0) & = f(x) \\ h(x,y+1) & = g(x,y,h(x,y)). \end{cases}$$

Thus, if the function $f$ has been coded in PCF by a term $f' : \mathtt{nat} \to \mathtt{nat}$ and the function $g$ by a term $g' : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$, then $h$ can be coded by

$$\begin{aligned} H \overset{\text{def}}{=} {} & \mathtt{fix}(\mathtt{fun}\, h{:}\,\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}.\ \mathtt{fun}\, x{:}\,\mathtt{nat}.\ \mathtt{fun}\, y{:}\,\mathtt{nat}. \\ & \qquad \mathtt{if\ zero?}(y)\ \mathtt{then}\ f'\ x\ \mathtt{else}\ g'\ x\ (\mathtt{pred}(y))\ (h\ x\ (\mathtt{pred}(y)))). \end{aligned}$$

Apart from primitive recursion, and the base cases, the other construction needed for defining partial recursive functions is *minimisation*. For example, the partial function $m\colon \mathbb{N} \rightharpoonup \mathbb{N}$ defined from $k\colon \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ by minimisation satisfies that for all $x \in \mathbb{N}$, $m(x)$ is the least $y \geq 0$ such that $k(x,y) = 0$ and $\forall z.\, 0 \leq z < y \Rightarrow k(x,z) > 0$. This can also be expressed using fixed points. For if $k$ has been coded in PCF by a term $k' : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$, then in fact $m$ can be coded as $\mathtt{fun}\, x{:}\,\mathtt{nat}.\ m'\ x\ 0$ where

$$\begin{aligned} m' \overset{\text{def}}{=} {} & \mathtt{fix}(\mathtt{fun}\, m'{:}\,\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}.\ \mathtt{fun}\, x{:}\,\mathtt{nat}.\ \mathtt{fun}\, y{:}\,\mathtt{nat}. \\ & \qquad \mathtt{if\ zero?}(k'\ x\ y)\ \mathtt{then}\ y\ \mathtt{else}\ (m'\ x\ \mathtt{succ}(y))). \end{aligned}$$ *

**Proposition 23 (Determinism)** *Evaluation in PCF is* deterministic*: if both $t \Downarrow_\tau v$ and $t \Downarrow_\tau v'$ hold, then $v = v'$.* *

Proof By rule induction: one shows that

$$\{(t,\tau,v) \mid t \Downarrow_\tau v \wedge \forall v'.(t \Downarrow_\tau v' \Rightarrow v = v')\}$$

is closed under the axioms and rules defining $\Downarrow$. □

## 5.3 Contextual equivalence

Recall (from Part IB – Semantics) the general notion of contextual equivalence.

**Definition 18 (Contextual equivalence – informal)** Two phrases of a programming language are *contextually equivalent* if any occurrences of the first phrase in a *complete program* can be replaced by the second phrase without affecting the *observable results* of executing the program. *

It is really a family of notions, parameterised by the particular choices one takes for what constitutes a '*complete program*' in the language and what are the '*observable results*' of executing such programs. For PCF it is reasonable to take the programs to be closed terms of type nat or bool, and to observe the values (or divergence) that result from evaluating such terms. Open terms are incomplete, in the sense that they are missing the values for which their variables stand. Function types $\sigma \to \tau$ do not give sensible observable results: since values at function types are intentional, observing function types would lead us to distinguish functions which have different source code, which is too fine-grained.

First, we need to define contexts. There is an unfortunate clash of terminology between typing contexts $\Gamma$ used to define typing and evaluation contexts $\mathcal{C}$ used to define contextual equivalence. We will use context alone when it is clear what kind of context we mean, and talk about typing/evaluation contexts when ambiguity might arise.

**Definition 19 (Evaluation contexts)** An *evaluation context* is a term with a hole, written $-$, to be filled by a PCF term. Formally, it is given by the following grammar:

$$\begin{array}{rcl}\mathcal{C} & ::= & - \mid \mathtt{succ}(\mathcal{C}) \mid \mathtt{pred}(\mathcal{C}) \mid \mathtt{zero?}(\mathcal{C}) \mid \\ & & \mathtt{if}\ \mathcal{C}\ \mathtt{then}\ t\ \mathtt{else}\ t \mid \mathtt{if}\ t\ \mathtt{then}\ \mathcal{C}\ \mathtt{else}\ t \mid \mathtt{if}\ t\ \mathtt{then}\ t\ \mathtt{else}\ \mathcal{C} \mid \\ & & \mathtt{fun}\, x{:}\, \tau.\, \mathcal{C} \mid \mathcal{C}\ t \mid t\ \mathcal{C} \mid \mathtt{fix}(\mathcal{C})\end{array}$$

Given such a context $\mathcal{C}$,[5] we write $\mathcal{C}[t]$ for the PCF expression that results from replacing $-$ in $\mathcal{C}$ by $t$. *

Note that this form of substitution may well involve the capture of free variables in $t$ by binders in $\mathcal{C}$. For example, if $\mathcal{C}$ is $\mathtt{fun}\, x{:}\, \tau.\, -$, then $\mathcal{C}[x]$ is $\mathtt{fun}\, x{:}\, \tau.\, x$.

**Definition 20 (Typing for evaluation contexts)** Typing is extended straightforwardly to contexts: we write $\Gamma \vdash_{\Delta,\sigma} \mathcal{C} : \tau$ to mean that assuming that the hole has type $\sigma$ in typing context $\Delta$, the whole evaluation context has type $\tau$ in typing context $\Delta$. The only new rule is

$$\frac{}{\Gamma \vdash_{\Gamma,\tau} - : \tau}$$

All other rules from Fig. 2 are adapted to type the evaluation context using this new relation, so for instance the rule for application of a context is

$$\frac{\Gamma \vdash_{\Delta,\sigma} \mathcal{C} : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash_{\Delta,\sigma} \mathcal{C}\ u : \tau_2}$$

*

[5] It is common practice to write $\mathcal{C}[-]$ instead of $\mathcal{C}$ to indicate the symbol being used to mark the 'hole' in $\mathcal{C}$.

**Definition 21 (Contextual equivalence)** Given a type $\tau$, a typing context $\Gamma$ and terms $t, t' \in \mathrm{PCF}_{\Gamma,\tau}$, *contextual equivalence*, written $\Gamma \vdash t \cong_{\mathrm{ctx}} t' : \tau$ is defined to hold if for all evaluation contexts $\mathcal{C}$ such that $\cdot \vdash_{\Gamma,\tau} \mathcal{C} : \gamma$, where $\gamma$ is `nat` or `bool`, and for all values $v \in \mathrm{PCF}_\gamma$,

$$\mathcal{C}[t] \Downarrow_\gamma v \Leftrightarrow \mathcal{C}[t'] \Downarrow_\gamma v.$$

⋆

When $\Gamma$ is the empty context, we simply write $t \cong_{\mathrm{ctx}} t' : \tau$ for $\cdot \vdash t \cong_{\mathrm{ctx}} t' : \tau$.

**Remark 7** Note that divergence is covered by this definition. Indeed, if $\Gamma \vdash t \cong_{\mathrm{ctx}} t' : \tau$, by contrapositive if $t \Uparrow_\tau$, then also $t'$ must diverge, because if $t'$ would evaluate to some $v$ then $t$ should do so too. ⋆

## 5.4 Introducing denotational semantics

Contextual equivalence is the natural notion of equivalence between programs. However, it is generally very hard to work with, because of the universal quantification over all evaluation contexts. Thus, we would like to obtain another form of equivalence, which avoids this difficulty and is thus easier to handle. Denotational semantics provides tooling for this.

### The aims of denotational semantics

More precisely, our goals are to define

- a mapping of PCF types $\tau$ to domains $[\![\tau]\!]$;
- a mapping of closed, well-typed PCF terms $\cdot \vdash t : \tau$ to elements $[\![t]\!] \in [\![\tau]\!]$;
- denotation of open terms will be continuous functions.

And we moreover want to ensure that the following properties hold.

*Compositionality:* $[\![t]\!] = [\![t']\!] \Rightarrow [\![\mathcal{C}[t]]\!] = [\![\mathcal{C}[t']]\!]$.

*Soundness:* for any type $\tau$, $t \Downarrow_\tau v \Rightarrow [\![t]\!] = [\![v]\!]$.

*Adequacy:* for $\gamma =$ `bool` or `nat`, if $t \in \mathrm{PCF}_\gamma$ and $[\![t]\!] = [\![v]\!]$ then $t \Downarrow_\gamma v$.

The *soundness* and *adequacy* properties make precise the connection between the operational and denotational semantics for which we are aiming. Note that the adequacy property only involves the ‘ground’ datatypes $\mathbb{N}$ and $\mathbb{B}$. One cannot expect such a property to hold at function types because of the ‘intensional’ nature of values at such types we already mentioned. Indeed, such an adequacy property at function types would contradict the compositionality and soundness properties we want for $[\![-]\!]$, as the following example shows.

**Example 20** Consider the following two PCF value terms of type `nat` → `nat`:

$$v \overset{\text{def}}{=} \mathtt{fun}\, x{:}\, \mathtt{nat}.\, (\mathtt{fun}\, y{:}\, \mathtt{nat}.\, y)\; 0 \quad \text{and} \quad v' \overset{\text{def}}{=} \mathtt{fun}\, x{:}\, \mathtt{nat}.\, 0.$$

Now $v \not\Downarrow v'$, since $v$ is a value, so it does not evaluate further. However, the soundness and compositionality properties of $[\![-]\!]$ imply that $[\![v]\!] = [\![v']\!]$. Indeed, we have

$$\texttt{fun}\, y\colon \texttt{nat}.\ y\ 0 \Downarrow_{\texttt{nat}} 0.$$

So by soundness $\texttt{fun}\, y\colon \texttt{nat}.\ y\ 0 = [\![0]\!]$. Therefore, by compositionality for $\mathcal{C}[-] \stackrel{\text{def}}{=} \texttt{fun}\, x\colon \texttt{nat}.\ -$ we have

$$[\![\mathcal{C}[(\texttt{fun}\, y\colon \texttt{nat}.\ y)\ 0]]\!] = [\![\mathcal{C}[0]]\!]$$

*i.e.* $[\![v]\!] = [\![v']\!]$. *

The value of denotational semantics comes from the following theorem, that we can already prove now, from the requirements we just made.

**Theorem 24 (Semantic equality implies contextual equivalence)** *For all types $\tau$ and closed terms $t_1, t_2 \in \mathrm{PCF}_\tau$, if $[\![t_1]\!]$ and $[\![t_2]\!]$ are equal elements of the domain $[\![\tau]\!]$, then $t_1 \cong_{\text{ctx}} t_2 : \tau$.*

Proof

$$\begin{aligned}
\mathcal{C}[t_1] \Downarrow_{\texttt{nat}} v &\Rightarrow [\![\mathcal{C}[t_1]]\!] = [\![v]\!] && \text{(soundness)}\\
&\Rightarrow [\![\mathcal{C}[t_2]]\!] = [\![v]\!] && \text{(compositionality on } [\![t_1]\!] = [\![t_2]\!]\text{)}\\
&\Rightarrow \mathcal{C}[t_2] \Downarrow_{\texttt{nat}} v && \text{(adequacy)}
\end{aligned}$$

and symmetrically for $\mathcal{C}[t_2] \Downarrow_{\texttt{nat}} v \Rightarrow \mathcal{C}[t_1] \Downarrow_{\texttt{nat}} v$, and similarly for `bool`. □

This means that we can use denotational semantics to establish instances of contextual equivalence, by showing that terms have equal denotation. In many cases this is an easier task than proving contextual equivalence directly from the definition. Theorem 24 generalises to open terms: if the continuous functions that are the denotations of two open terms (of the same type for some typing context) are equal, then the terms are contextually equivalent.

The question remains, though, to know if this proof technique is complete. That is, is equality in the model a necessary condition for contextual equivalence? We will come back to this question (called full abstraction), in the last chapter.

## 5.5 Exercises

**Exercise 10** Carry out the suggested proof that evaluation is deterministic (Proposition 23).

**Exercise 11** Recall that Church's fixed point combinator in the untyped lambda calculus is $Y \overset{\text{def}}{=} \lambda f.\ (\lambda x.\ f\ (x\ x))(\lambda x.\ f\ (x\ x))$. Show that there are no PCF types $\tau_1, \tau_2, \tau_3$ so that the following typing relation holds:

$$\cdot \vdash \mathtt{fun}\, f\colon \tau_1.\ (\mathtt{fun}\, x\colon \tau_2.\ f\ (x\ x))\ (\mathtt{fun}\, x\colon \tau_2.\ f\ (x\ x)) : \tau_3$$

**Exercise 12** Define the following PCF terms:

$$\begin{aligned}\mathtt{plus} \overset{\text{def}}{=} {} & \mathtt{fun}\, x\colon \mathtt{nat}.\ \mathtt{fix}(\mathtt{fun}(p\colon \mathtt{nat} \to \mathtt{nat})(y\colon \mathtt{nat}).\\ & \quad \mathtt{if}\ \mathtt{zero?}(y)\ \mathtt{then}\ x\ \mathtt{else}\ \mathtt{succ}(p\ \mathtt{pred}(y)))\end{aligned}$$

$$\begin{aligned}\mathtt{mul} \overset{\text{def}}{=} {} & \mathtt{fun}\, x\colon \mathtt{nat}.\ \mathtt{fix}(\mathtt{fun}(t\colon \mathtt{nat} \to \mathtt{nat})(y\colon \mathtt{nat}).\\ & \quad \mathtt{if}\ \mathtt{zero?}(y)\ \mathtt{then}\ 0\ \mathtt{else}\ \mathtt{plus}\ x\ (t\ \mathtt{pred}(y)))\end{aligned}$$

Show by induction on $n \in \mathbb{N}$ that for all $m \in \mathbb{N}$

$$\begin{aligned}\mathtt{plus}\ \underline{m}\,\underline{n} &\Downarrow_{\mathtt{nat}} \underline{m+n}\\ \mathtt{mul}\ \underline{m}\,\underline{n} &\Downarrow_{\mathtt{nat}} \underline{m \cdot n}.\end{aligned}$$

Using the above functions, define a factorial function and show that it does indeed compute the factorial.

# 6 Denotational Semantics for PCF

We turn now to the task of defining a denotational semantics for PCF with the properties of compositionality, soundness, and adequacy.

## 6.1 Types and contexts

**Definition 22 (Semantics of types)** For each PCF type $\tau$ we define is semantics as a domain $[\![\tau]\!]$ by induction on its structure:

$$
\begin{aligned}
[\![\texttt{nat}]\!] &\overset{\text{def}}{=} \mathbb{N}_\perp && \text{(flat domain)} \\
[\![\texttt{bool}]\!] &\overset{\text{def}}{=} \mathbb{B}_\perp && \text{(flat domain)} \\
[\![\tau \to \tau']\!] &\overset{\text{def}}{=} [\![\tau]\!] \to [\![\tau']\!] && \text{(function domain)}
\end{aligned}
$$

We use of flat domains (Definition 12) and function domains (Definition 15). ⋆

**Definition 23 (Semantics of context)** The semantics of a context $\Gamma$ is an environment, *i.e.* a mapping from variables to values in the relevant domain:

$$[\![\Gamma]\!] \overset{\text{def}}{=} \prod_{x \in \text{dom}(\Gamma)} [\![\Gamma(x)]\!]$$

That is, $[\![\Gamma]\!]$ is the domain of partial functions $\rho$ from variables to domains such that $\text{dom}(\rho) = \text{dom}(\Gamma)$ and $\rho(x) \in [\![\Gamma(x)]\!]$ for all $x \in \text{dom}(\Gamma)$. ⋆

**Remark 8** Unfolding the definition, we get that

- for the empty context, $[\![\cdot]\!] = \mathbb{1}$, *i.e.* a type with a single element (technically, the nowhere-defined partial function);
- for a context with only one variable $[\![x\colon \tau]\!] = (\{x\} \to [\![\tau]\!]) \cong [\![\tau]\!]$;
- more generally, $[\![x_1\colon \tau_1, \dots, x_n\colon \tau_n]\!] = [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$.

Given these isomorphisms, we will think of environments both as iterated products and as partial maps, depending on what is most useful. ⋆

## 6.2 Terms

To every typing judgement

$$\Gamma \vdash t : \tau$$

we associate a continuous function

$$[\![\Gamma \vdash t : \tau]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$$

between domains. In other words,

$$[\![-]\!] : \mathrm{PCF}_{\Gamma,\tau} \to [\![\Gamma]\!] \to [\![\tau]\!]$$

**Remark 9**

- Just as in Section 1.1, we use $[\![-]\!]$ for the three different functions computing the denotation of a type, a context and a term.
- Because terms have at most one typing derivation (and well-typed terms exactly one), defining the denotation on well-typed terms or on typing derivation is equivalent, and we will conflate the two. This abuse would not be so benign if we had more than one typing derivation! In that case we could have different semantics for different derivations for the same term, and so talking about "the" semantics of a term would be ambiguous. $\ast$

The continuous function is defined by induction on the structure of $t$ (or, equivalently, on its typing derivation).

**Definition 24 (Denotation of operations on $\mathbb{B}$ and $\mathbb{N}$)** Let succ, pred and zero? be the functions respectively defined as follows:begin

$$\mathrm{succ} : \begin{array}{rcl} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & n+1 \end{array} \qquad \mathrm{pred} : \begin{array}{rcl} \mathbb{N} & \to & \mathbb{N} \\ 0 & \mapsto & \text{undefined} \\ n+1 & \mapsto & n \end{array}$$

$$\mathrm{zero?} : \begin{array}{rcl} \mathbb{N} & \to & \mathbb{B} \\ 0 & \mapsto & \text{true} \\ n+1 & \mapsto & \text{false} \end{array}$$

We define the following:

$$\begin{array}{rcll}
[\![\mathtt{0}]\!]\,(\rho) & \overset{\text{def}}{=} & 0 & \in \mathbb{N}_\bot \\
[\![\mathtt{true}]\!]\,(\rho) & \overset{\text{def}}{=} & \text{true} & \in \mathbb{B}_\bot \\
[\![\mathtt{false}]\!]\,(\rho) & \overset{\text{def}}{=} & \text{false} & \in \mathbb{B}_\bot \\
\\
[\![\mathtt{succ}(t)]\!]\,(\rho) & \overset{\text{def}}{=} & \mathrm{succ}_\bot([\![t]\!]\,(\rho)) & \in \mathbb{N}_\bot \\
[\![\mathtt{pred}(t)]\!]\,(\rho) & \overset{\text{def}}{=} & \mathrm{pred}_\bot([\![t]\!]\,(\rho)) & \in \mathbb{N}_\bot \\
[\![\mathtt{zero?}(t)]\!]\,(\rho) & \overset{\text{def}}{=} & \mathrm{zero?}_\bot([\![t]\!]\,(\rho)) & \in \mathbb{B}_\bot \\
\\
[\![\mathtt{if}\ b\ \mathtt{then}\ t\ \mathtt{else}\ t']\!] & \overset{\text{def}}{=} & \mathrm{if}([\![b]\!]\,(\rho), [\![t]\!]\,(\rho), [\![t']\!]\,(\rho)) & \in [\![\tau]\!]
\end{array}$$

Where $f_\perp$ is the flat domain lifting, defined in Proposition 7, and the semantic conditional function $\mathrm{if} : \mathbb{B}_\perp \times (D \times D) \to D$ is defined in Proposition 10 – here we apply it with $[\![\tau]\!]$, where $\tau$ is the common type of $t$ and $t'$. $\ast$

**Remark 10** We have already done all the work necessary to show this indeed defines continuous functions. By Example 10, the constant functions interpreting 0, `true` and `false` are continuous. By Proposition 7 and continuity of composition (Proposition 15), if $[\![t]\!]$ is continuous, then so is $[\![\mathtt{succ}(t)]\!] = \mathrm{succ}_\perp \circ [\![t]\!]$, and similarly for `pred` and `zero`?. Finally, for the conditional, we rely on Proposition 10 telling us that if is continuous, and on Proposition 9 for continuity of pairing. $\ast$

**Definition 25 (Denotation of the λ-calculus operations)** We define the following:

$$\begin{aligned}
[\![x]\!]\,(\rho) &\overset{\mathrm{def}}{=} \rho(x) && \in [\![\Gamma(x)]\!] \quad (\text{for } x \in \mathrm{dom}(\Gamma)) \\
[\![t_1\, t_2]\!]\,(\rho) &\overset{\mathrm{def}}{=} ([\![t_1]\!]\,(\rho))\ ([\![t_2]\!]\,(\rho)) \\
[\![\mathtt{fun}\, x{:}\,\tau.\, t]\!]\,(\rho) &\overset{\mathrm{def}}{=} \lambda d \in [\![\tau]\!]\,.\ [\![t]\!]\,(\rho[x \mapsto d])
\end{aligned}$$

The interpretation of variable is the projection from a general product (defined in Proposition 11), that of an application is eval as defined in Proposition 13, and that of abstraction is currying, defined in Proposition 14. $\ast$

**Definition 26 (Denotation of fixed points)** Finally, we set

$$[\![\mathtt{fix}\, f]\!]\,(\rho) \overset{\mathrm{def}}{=} \mathrm{fix}([\![f]\!]\,(\rho))$$

$\ast$

**Theorem 25 (Denotation is well-defined)** *For any PCF term $t$ such that $\Gamma \vdash t{:}\,\tau$, the object $[\![t]\!]$ is well-defined and a continuous function $[\![t]\!] : [\![\Gamma]\!] \to \tau$.*

Proof The proof is by induction on the typing derivation.

We have already explained in Remark 10 that the interpretation of all the operations on booleans and natural numbers are continuous or preserve continuity obtained from induction hypothesis.

Similarly, the interpretation of a variable is continuous by Proposition 11, that of application by Proposition 13 and continuity of pairing. For abstraction, assume we have $\Gamma, x{:}\,\sigma \vdash t : \tau$. By induction hypothesis, $[\![t]\!] : [\![\Gamma, x{:}\,\sigma]\!] \to [\![\tau]\!]$. But $[\![\Gamma, x{:}\,\sigma]\!] = [\![\Gamma]\!] \times [\![\sigma]\!]$, and so we get that $\mathrm{cur}([\![t]\!]) : [\![\Gamma]\!] \to ([\![\sigma]\!] \to [\![\tau]\!]) = [\![\Gamma]\!] \to [\![\sigma \to \tau]\!]$ as necessary.

Finally, continuity of the interpretation of `fix` is exactly Proposition 16. □

**Remark 11 (Denotation of closed terms)** If $t \in \mathrm{PCF}_\tau$, then by definition $\cdot \vdash t : \tau$ holds, so we get $[\![t]\!] : [\![\cdot]\!] \to [\![\tau]\!]$. Recall from Remark 8 that the interpretation of the empty context is a singleton set $\mathbb{1}$. Thus, $[\![\cdot]\!] \to [\![\tau]\!]$ is in bijection with $\tau$. So we can identify the denotation of closed PCF terms with elements of the domain denoting their type, and consider that $[\![t]\!] \in [\![\tau]\!]$.

## 6.3 Compositionality

The fact that the denotational semantics of PCF terms is *compositional* – *i.e.* that the denotation of a compound term is a function of the denotations of its immediate subterms – is part and parcel of the definition of $[\![t]\!]$ by induction on the structure of $t$: the denotation of each term constructor is defined by combining the denotation of their immediate subterms.

**Theorem 26 (Compositionality)** *Suppose $t, u \in \mathrm{PCF}_{\Gamma,\tau}$, such that*

$$[\![t]\!] = [\![u]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$$

*Suppose moreover that $\mathcal{C}[-]$ is a PCF context such that $\Gamma' \vdash_{\Gamma,\tau} \mathcal{C} : \tau'$. Then*

$$[\![\mathcal{C}[t]]\!] = [\![\mathcal{C}[u]]\!] : [\![\Gamma']\!] \to [\![\tau']\!] \,.$$

*

Proof The proof is by induction on the typing derivation for $\mathcal{C}[-]$. In the base case of the typing rule for $-$, we have that $-[t] = t$ and $-[u] = u$, so we use the fact that the denotation of $t$ and $u$ are equal. In all other case, we use the induction hypothesis, together with the fact that the denotation of a term is defined in terms of that of its subterm.

For example, let us consider in detail the case of succ, so assume $\mathcal{C} = \mathtt{succ}(\mathcal{C}')$. We have $\mathcal{C}[t] = \mathtt{succ}(\mathcal{C}'[t])$, and similarly for $u$. By induction hypothesis, $[\![\mathcal{C}'[t]]\!] = [\![\mathcal{C}'[u]]\!]$. But then, for any $\rho \in [\![\Gamma]\!]$,

$$\begin{aligned} [\![\mathcal{C}[t]]\!]\,(\rho) &= [\![\mathtt{succ}(\mathcal{C}'[t])]\!]\,(\rho) \\ &= \mathrm{succ}_\bot([\![\mathcal{C}'[t]]\!]\,(\rho)) \\ &= \mathrm{succ}_\bot([\![\mathcal{C}'[u]]\!]\,(\rho)) \\ &= [\![\mathtt{succ}(\mathcal{C}'[u])]\!]\,(\rho) \\ &= [\![\mathcal{C}[u]]\!]\,(\rho) \end{aligned}$$

And so $[\![\mathcal{C}[t]]\!] = [\![\mathcal{C}[u]]\!]$. □

As a special case for closed $t$ and $u$, we get the requirement of compositionality as stated in Section 5.4.

**Remark 12** We can even go one step further, and *define* the denotation of a context directly: if $\Gamma \vdash_{\Delta,\sigma} \mathcal{C} : \tau$, then $[\![\mathcal{C}]\!]$ should be an element of $([\![\Delta]\!] \to [\![\sigma]\!]) \to [\![\Gamma]\!] \to [\![\tau]\!]$. Intuitively, a context takes something of the "type of the hole" – since that hole lives in a context, this is a continuous function – and an environment for the context, and gives back a semantic value of the type of the whole context. To obtain this, set

$$\begin{aligned} [\![-]\!]\,(d) &= d \\ [\![\mathcal{C}\; t]\!]\,(d)(\rho) &= ([\![\mathcal{C}]\!]\,(d)(\rho))([\![t]\!]\,(\rho)) \\ &\vdots \end{aligned}$$

That is, define the denotation of the hole to simply be the identity, and on all other context former, to mimick the denotation of terms.

By a direct induction on the context typing, if $\Gamma \vdash_{\Delta,\sigma} \mathcal{C} : \tau$ and $\Delta \vdash t : \sigma$, we have

$$[\![\mathcal{C}[t]]\!] = [\![\mathcal{C}]\!]\,([\![t]\!])$$

This gives us a more conceptual proof of compositionality, exposing its essence: given the hypothesis of Theorem 26, we have

$$[\![\mathcal{C}[t]]\!] = [\![\mathcal{C}]\!]\,([\![t]\!]) = [\![\mathcal{C}]\!]\,([\![t']\!]) = [\![\mathcal{C}[t']]\!]$$

*

The following substitution property gives another aspect of the compositional nature of the denotational semantics of PCF. It can be proven by induction on the structure of the term $t$.

**Proposition 27 (Substitution property of the semantic function)** *Assume*

$$\begin{aligned} \Gamma &\vdash u : \sigma \\ \Gamma, x{:}\sigma &\vdash t : \tau \end{aligned}$$

*(so that by Proposition 22 we also have* $\Gamma \vdash t[u/x] : \tau$*). Then for all* $\rho \in [\![\Gamma]\!]$

$$[\![t[u/x]]\!]\,(\rho) = [\![t]\!]\,(\rho[x \mapsto [\![u]\!]\,(\rho)]).$$

*In particular when* $\Gamma = \cdot$, $[\![t]\!] : [\![\sigma]\!] \to [\![\tau]\!]$ *and*

$$[\![t[u/x]]\!] = [\![t]\!]\,([\![u]\!])$$

*

## 6.4 Soundness

The second of the aims mentioned in Section 5.4 is soundness: if a closed PCF term $t$ evaluates to a value $v$ in the operational semantics, then $t$ and $v$ have the same denotation.

**Theorem 28 (Soundness)** *For all PCF types $\tau$ and all closed terms $t, v \in \mathrm{PCF}_\tau$ with $v$ a value, if $t \Downarrow_\tau v$ is derivable from the axioms and rules in Fig. 3, then*

$$[\![t]\!] = [\![v]\!] \in [\![\tau]\!]$$

*that is, $[\![t]\!]$ and $[\![v]\!]$ are equal elements of the domain $[\![\tau]\!]$.* ⁎

Proof By induction on the (inductively defined) relation $\Downarrow$. Specifically, defining

$$\Phi(t, \tau, v) \overset{\text{def}}{\Leftrightarrow} [\![t]\!] = [\![v]\!] \in [\![\tau]\!]$$

we need to show that the property $\Phi(t, \tau, v)$ is closed under the axioms and rules in Fig. 3. We give the argument for rules Fun and Fix, and leave the others as easy exercises.

*Case Fun.* Suppose

$$[\![t_1]\!] = [\![\mathtt{fun}\, x{:}\,\tau.\, t_1']\!] \in [\![\tau \to \tau']\!] \tag{4}$$
$$[\![t_1'[t_2/x]]\!] = [\![v]\!] \in [\![\tau']\!]\,. \tag{5}$$

We have to prove that $[\![t_1\ t_2]\!] = [\![v]\!] \in [\![\tau']\!]$. But

$$\begin{aligned}
[\![t_1\ t_2]\!] &= [\![t_1]\!]\,([\![t_2]\!]) && \text{by definition of } [\![-]\!] \text{ (Definition 25)}\\
&= [\![\mathtt{fun}\, x{:}\,\tau.\, t_1']\!]\,([\![t_2]\!]) && \text{by (4)}\\
&= (\lambda d \in [\![\tau]\!]\,.\ [\![t_1']\!]\,(d))([\![t_2]\!]) && \text{by definition of } [\![-]\!] \text{ (Definition 25)}\\
&= [\![t_1']\!]\,([\![t_2]\!]) &&\\
&= [\![t_1'[t_2/x]]\!] && \text{by Proposition 27}\\
&= [\![v]\!] && \text{by (5).}
\end{aligned}$$

*Case Fix.* Suppose

$$[\![t\ \mathtt{fix}(t)]\!] = [\![v]\!] \in [\![\tau]\!]\,. \tag{6}$$

We have to prove that $[\![\texttt{fix}(t)]\!] = [\![v]\!] \in [\![\tau]\!]$. But

$$\begin{aligned}
[\![\texttt{fix}(t)]\!] &= \mathrm{fix}([\![t]\!]) && \text{by definition of } [\![-]\!] \text{ (Definition 26)}\\
&= [\![t]\!]\,(\mathrm{fix}([\![t]\!])) && \text{by fixed point property of fix}\\
&= [\![t]\!]\,([\![\texttt{fix}(t)]\!]) && \text{by definition of } [\![-]\!] \text{ (Definition 26)}\\
&= [\![t\ \texttt{fix}(t)]\!] && \text{by definition of } [\![-]\!] \text{ (Definition 25)}\\
&= [\![v]\!] && \text{by (6).}
\end{aligned}$$

□

We have now established two of the three properties of the denotational semantics of PCF stated in Section 5.4 (and which are in particular needed to use denotational equality to prove PCF contextual equivalences). The third property, *adequacy*, is not so easy to prove as are the first two. Its proof is the subject of the next section.

## 6.5 Exercises

**Exercise 13** Prove the substitution property of semantic (Proposition 27).

**Exercise 14** Defining $\Omega_\tau \overset{\text{def}}{=} \texttt{fix}(\texttt{fun}\,x\colon\tau.\,x)$, show that $[\![\Omega_\tau]\!]$ is the least element $\bot$ of the domain $[\![\tau]\!]$. Deduce that $[\![\texttt{fun}\,x\colon\tau.\,\Omega_\tau]\!] = [\![\Omega_{\tau\to\tau}]\!]$.

# 7 Relating Denotational and Operational Semantics

We have already seen (in Section 6.4) that the denotational semantics of PCF given in Section 6 is *sound* for the operational semantics, in the sense defined in Section 5.4: if $t \Downarrow_\tau v$ then $[\![t]\!] = [\![v]\!]$. But we want more: we should be able to get back from denotational to operational properties.

To this aim, we prove the property of *adequacy*: on closed terms of the base types `bool` and `nat`, denotational and operational semantics agree. More precisely, we have to prove for any closed PCF term $t$ and value $v$ of type $\gamma = \texttt{nat}$ or `bool`, that

$$[\![t]\!] = [\![v]\!] \Rightarrow t \Downarrow_\tau v.$$

For any closed PCF term $t$ and value $v$ of ground type $\gamma \in \{\texttt{nat}, \texttt{bool}\}$

$$[\![t]\!] = [\![v]\!] \in [\![\gamma]\!] \Rightarrow t \Downarrow_\gamma v$$

Adequacy does not hold at function types or for open terms

$$[\![\texttt{fun}\, x{:}\, \tau.\, (\texttt{fun}\, y{:}\, \tau.\, y)\, x]\!] \quad = \quad [\![\texttt{fun}\, x{:}\, \tau.\, x]\!] \quad : [\![\tau]\!] \to [\![\tau]\!]$$

but

$$\texttt{fun}\, x{:}\, \tau.\, (\texttt{fun}\, y{:}\, \tau.\, y)\, x \not\Downarrow_{\tau\to\tau} \texttt{fun}\, x{:}\, \tau.\, x$$

Perhaps surprisingly, this is not so easy to prove. We will employ a method due to Plotkin (although not quite the one used in his original paper on PCF [3]) and Mulmuley [4] making use of the following 'formal approximation' relations. This is a logical relation, somewhat similar to those seen in Part II – Types to show termination of simply-typed λ-calculus.

## 7.1 Formal approximation relation

We define a family of binary relations

$$\lhd_\tau \subseteq [\![\tau]\!] \times \mathrm{PCF}_\tau$$

indexed by the PCF type $\tau$. For each $\tau$, $\lhd_\tau$ relates elements of the domain $[\![\tau]\!]$ to *closed* PCF terms of type $\tau$. We use infix notation and write $d \lhd_\tau t$ instead of $(d, t) \in \lhd_\tau$. The definition of these relations $\lhd_\tau$ proceeds *by induction on the structure of the type* $\tau$. (Read the definition in conjunction with the definition of $[\![\tau]\!]$ given in Definition 22.)

**Definition 27 (Formal approximation)** Given a PCF type $\tau$, a semantic value $d \in [\![\tau]\!]$ and a closed term $t \in \mathrm{PCF}_\tau$, the *formal approximation* relation $\lhd_\tau$ is defined as follows:

$$
\begin{array}{rcl}
d \lhd_{\mathtt{nat}} t & \overset{\text{def}}{\Leftrightarrow} & (d \in \mathbb{N} \Rightarrow t \Downarrow_{\mathtt{nat}} \underline{d}) \\
d \lhd_{\mathtt{bool}} t & \overset{\text{def}}{\Leftrightarrow} & (d = \text{true} \Rightarrow t \Downarrow_{\mathtt{bool}} \mathtt{true}) \\
 & & \wedge (d = \text{false} \Rightarrow t \Downarrow_{\mathtt{bool}} \mathtt{false}) \\
d \lhd_{\tau \to \tau'} t & \overset{\text{def}}{\Leftrightarrow} & \forall e \in [\![\tau]\!], u \in \mathrm{PCF}_\tau . (e \lhd_\tau u \Rightarrow d(e) \lhd_{\tau'} t\, u)
\end{array}
$$

*

The key property of the relations $\lhd_\tau$ is that they are respected by all operations of the PCF language. But to be able to state this, we need to extend the relation to open contexts, for which we need a few definitions.

**Definition 28 (Parallel closed substitution)** Given a typing context $\Gamma$, a parallel closed substitution $\sigma$ for $\Gamma$ is a function mapping each variable $x \in \operatorname{dom}\Gamma$ to a closed PCF term $\sigma(x) \in \mathrm{PCF}_{\Gamma(x)}$.

We write $\vdash \sigma : \Gamma$ to express that $\sigma$ is such a parallel closed substitution, and $t[\sigma]$ for the action of such a substitution on terms, simultaneously replacing each variable $x$ appearing in $t$ by $\sigma(x)$.

**Remark 13** Just like unary substitution, n-ary substitution preserve typing: if $\vdash \sigma : \Gamma$ and $\Gamma \vdash t : \tau$, then $\vdash t[\sigma] : \tau$.

Actually, we can more generally define parallel open substitutions $\Delta \vdash \sigma : \Gamma$, but we will not use such substitutions in this course. *

**Definition 29 (Formal approximation for substitution)** Given a context $\Gamma$, a substitution $\sigma$ such that $\vdash \sigma : \Gamma$ and an environment $\rho \in [\![\Gamma]\!]$, we extend the formal approximation as follows:

$$\rho \lhd_\Gamma \sigma \overset{\text{def}}{\Leftrightarrow} \forall x \in \operatorname{dom}(\Gamma).\ \rho(x) \lhd_{\Gamma(x)} \sigma(x).$$

*

We can now finally state the fundamental property of the logical relation.

**Theorem 29 (Fundamental property of formal approximation)** *Given a term t such that $\Gamma \vdash t : \tau$ for some $\Gamma$ and $\tau$, for any environment $\rho$ and substitution $\sigma$ such that $\rho \lhd_\Gamma \sigma$, we have $[\![t]\!]\,(\rho) \lhd_\tau t[\sigma]$.* *

Note that this fundamental property specialises in case $\Gamma = \emptyset$ to give

$$[\![t]\!] \lhd_\tau t$$

for all types $\tau$ and all closed PCF terms $t : \tau$. (Here we are using the notation for denotations of closed terms introduced in Remark 11.) Using this, we can complete the proof of adequacy.

**Theorem 30 (Adequacy)** *For any closed PCF term $t$ and value $v$ of ground type $\gamma \in \{\mathtt{nat}, \mathtt{bool}\}$*

$$[\![t]\!] = [\![v]\!] \in [\![\gamma]\!] \Rightarrow t \Downarrow_\gamma v$$

Proof (Adequacy) We give the proof for `nat`, the `bool` case is entirely similar. Because $v$ is a ground value of type `nat`, it must be the case that $v = \underline{n}$ for some $n \in \mathtt{nat}$. Hence

$$\begin{array}{lll} [\![t]\!] = [\![\underline{n}]\!] = n & & \text{by assumption and definition of } [\![-]\!] \\ \quad \Rightarrow n = [\![t]\!] \lhd_\tau t & & \text{by the fundamental property} \\ \quad \Rightarrow t \Downarrow \underline{n} & & \text{by definition of } \lhd_{\mathtt{nat}} \end{array}$$

□

## 7.2 Proof of the fundamental property of formal approximation

We first need some preliminary lemmas on formal approximation, all proven by induction on $\tau$.

**Lemma 31** *The least element approximates any program: for any $\tau$ and $t \in \mathrm{PCF}_\tau$, $\bot_{[\![\tau]\!]} \lhd_\tau t$.* ⋆

Proof At ground type `nat`, we must show that if $\bot_{\mathbb{N}_\bot} \in \mathbb{N}$ then a certain condition holds. But this is vacuously true, since $\bot \notin \mathbb{N}$. The same reasoning goes for `bool`.

For $\tau \to \tau'$, we have

$$\bot_{[\![\tau\to\tau']\!]} = \bot_{[\![\tau]\!]\to[\![\tau']\!]} = \lambda e \in [\![\tau]\!] \,.\, \bot_{[\![\tau']\!]}$$

Now, assuming $e, u$ such that $e \lhd_\tau u$, we must show $\bot_{[\![\tau\to\tau']\!]}(e) \lhd_{\tau'} t\ u$. But this amounts to $\bot_{[\![\tau']\!]} \lhd_{\tau'} t\ u$, which is true by induction hypothesis on $\tau'$. □

**Lemma 32** *Given $t \in \mathrm{PCF}_\tau$, the set $\{d \in [\![\tau]\!] \mid d \lhd_\tau t\}$ is a chain-closed subset of the domain $[\![\tau]\!]$.* ⋆

**Lemma 33** *If $d' \sqsubseteq d$ and $d \lhd_\tau t$, then $d' \lhd_\tau t$.* ⋆

Proof At base types, if $d' \sqsubseteq d$ either $d' = d$ or $d' = \bot$. In the first case, the conclusion is direct, and in the second we can apply Lemma 31.

At the function type $\tau \to \tau'$, as for the previous two lemmas we use the induction hypothesis on $\tau'$ to conclude. □

**Lemma 34** *If $t, t' \in \mathrm{PCF}_\tau$ are such that $\forall v.\ t \Downarrow_\tau v \Rightarrow t' \Downarrow_\tau v$, and $d \lhd_\tau t$, then $d \lhd_\tau t'$.* ⋆

Now we have all we need to look at the fundamental property.

PROOF (THEOREM 29, FUNDAMENTAL PROPERTY OF FORMAL APPROXIMATION) We proceed by induction on typing, to show that

$$\Phi(\Gamma, t, \tau) \overset{\text{def}}{\Leftrightarrow} \forall \rho, \sigma.\, (\rho \lhd_\Gamma \sigma \;\Rightarrow\; [\![t]\!]\,(\rho) \lhd_\tau t[\sigma])$$

*Case ZERO.* $\Phi(\Gamma, 0, \mathtt{nat})$ holds because $0 \Downarrow_{\mathtt{nat}} 0 = \underline{0}$.

*Case SUCC.* We have to prove that $\Phi(\Gamma, t, \mathtt{nat})$ implies $\Phi(\Gamma, \mathtt{succ}(t), \mathtt{nat})$, which amounts to showing that for all $d' \in [\![\mathtt{nat}]\!]$, $t' \in \mathrm{PCF}_{\mathtt{nat}}$,

$$d' \lhd_{\mathtt{nat}} t' \Rightarrow \mathrm{succ}_\bot(d') \lhd_{\mathtt{nat}} \mathtt{succ}(t') \tag{7}$$

That is, we can restrict to proving the statement on closed terms. Indeed, if we are given $\rho, \sigma$ such that $\rho \lhd_\Gamma \sigma$, $[\![\mathtt{succ}(t)]\!]\,(\rho) = \mathrm{succ}_\bot([\![t]\!]\,(\rho))$ and $(\mathtt{succ}(t))[\sigma] = \mathtt{succ}(t[\sigma])$, we can apply 7 with $d' = [\![t]\!]\,(\rho)$ and $t' = t[\sigma]$, since the right-hand side then becomes exactly our induction hypothesis.

To show this, assume $succ_\bot(d') \in \mathbb{N}$. This implies that $d' \in \mathbb{N}$, and so by induction hypothesis that $t' \Downarrow_{\mathtt{nat}} \underline{d'}$ We can then use rule SUCC to conclude

$$\mathtt{succ}(t') \Downarrow_{\mathtt{nat}} \mathtt{succ}(\underline{d'}) = \underline{d' + 1} = \underline{\mathrm{succ}_\bot(d')}$$

*Cases PRED, IsZ.* These cases are similar to the previous one, for the functions pred and zero?.

*Cases TRUE, FALSE.* These cases are similar to those for 0.

*Case IF.* Just as for $\mathtt{succ}$, it is enough to consider closed terms, *i.e.* to show that if $d_1 \lhd_{\mathtt{bool}} t_1$, $d_2 \lhd_\tau t_2$, and $d_3 \lhd_\tau t_3$, then

$$\mathrm{if}(d_1, d_2, d_3) \lhd_\tau \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 \tag{8}$$

where if is the continuous function if : $\mathbb{B}_\bot \times ([\![\tau]\!] \times [\![\tau]\!]) \to [\![\tau]\!]$ of Proposition 10 that was used in Definition 24 as the denotation of the conditional. If $d_1 = \bot \in \mathbb{B}_\bot$, then $\mathrm{if}(d_1, d_2, d_3) = \bot$ and (8) holds by Lemma 31. So we are left with the cases $d_1 = \mathrm{true}$ or $d_1 = \mathtt{false}$. We consider the case $d_1 = \mathrm{true}$; the argument for the other case is similar.

Since true $= d_1 \lhd_{\mathtt{bool}} t_1$, by the definition of $\lhd_{\mathtt{bool}}$ we have $t_1 \Downarrow_{\mathtt{bool}} \mathtt{true}$. It follows from rule IFT for evaluation that

$$\forall v.\, (t_2 \Downarrow_\tau v \Rightarrow \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 \Downarrow_\tau v).$$

So Lemma 34 applied to $d_2 \lhd_\tau t_2$ yields that

$$d_2 \lhd_\tau \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$$

and then since $d_2 = \mathrm{if}(\mathrm{true}, d_2, d_3) = \mathrm{if}(d_1, d_2, d_3)$, we get (8), as required.

*Case Var.* $\Phi(\Gamma, x, \Gamma(x))$ holds because if $\rho \lhd_\Gamma \sigma$, then for all $x \in \mathrm{dom}(\Gamma)$ we have

$$[\![x]\!]\,(\rho) \overset{\text{def}}{=} \rho(x) \lhd_{\Gamma(x)} \sigma(x) \overset{\text{def}}{=} x[\sigma]$$

*Case Fun.* By induction hypothesis, $\Phi(\Gamma, x{:}\,\tau, t, \tau')$ holds. We moreover assume that $\rho \lhd_\Gamma \sigma$ holds, and we have to show that $[\![\texttt{fun}\,x{:}\,\tau.\,t]\!]\,(\rho) \lhd_{\tau\to\tau'} (\texttt{fun}\,x{:}\,\tau.\,t)[\sigma]$, *i.e.* that $d \lhd_\tau u$ implies

$$[\![\texttt{fun}\,x{:}\,\tau.\,t]\!]\,(\rho)(d) \lhd_{\tau'} ((\texttt{fun}\,x{:}\,\tau.\,t)[\sigma])\,u. \tag{9}$$

From Definition 25, we have

$$[\![\texttt{fun}\,x{:}\,\tau.\,t]\!]\,(\rho)(d) = [\![t]\!]\,(\rho[x \mapsto d]). \tag{10}$$

Since $(\texttt{fun}\,x{:}\,\tau.\,t)[\sigma] = \texttt{fun}\,x{:}\,\tau.\,(t[\sigma])$ and $(t[\sigma])[u/x] = t[\sigma[x \mapsto u]]$, by rule Fun for evaluation,

$$\forall v.\ (t[\sigma[x \mapsto u]] \Downarrow_{\tau'} v \Rightarrow ((\texttt{fun}\,x{:}\,\tau.\,t)[\sigma])\,u \Downarrow_{\tau'} v)\,. \tag{11}$$

Since $\rho \lhd_\Gamma \sigma$ and $d \lhd_\tau u$, we have $\rho[x \mapsto d] \lhd_{\Gamma, x:\tau} \sigma[x \mapsto u]$; so by $\Phi(\Gamma, x{:}\,\tau, t, \tau')$ we obtain

$$[\![t]\!]\,(\rho[x \mapsto d]) \lhd_{\tau'} t[\sigma[x \mapsto u]].$$

Then (9) follows from this by using (10) and applying Lemma 34 with (11).

*Case App.* It suffices to show that if $d_1 \lhd_{\tau\to\tau'} t_1$ and $d_2 \lhd_\tau t_2$, then $d_1(d_2) \lhd_{\tau'} t_1\,t_2$. But this follows immediately from the definition of $\lhd_{\tau\to\tau'}$.

*Case Fix.* As in the case of Succ, it is enough to show that

$$d \lhd_{\tau\to\tau} f \Rightarrow \mathrm{fix}\,d \lhd_\tau \texttt{fix}\,f$$

To show this, we use Scott induction (Theorem 17) on the set

$$\{e \in [\![\tau]\!] \mid e \lhd_\tau \texttt{fix}\,f\}$$

By Lemmas 31 and 32, this set contains $\bot_{[\![\tau]\!]}$ and is chain-closed. It thus suffices to prove that it is stable under $f$, *i.e.* that

$$\forall e \in [\![\tau]\!] \,.\, (e \in S \Rightarrow d(e) \in S).$$

Now, by definition of $\lhd_{\tau \to \tau}$, it is the case that

$$d(e) \lhd_\tau f\,(\mathtt{fix}\,f). \tag{12}$$

Rule Fix for evaluation implies

$$\forall v.\,(f\,(\mathtt{fix}\,f))\Downarrow_\tau v \Rightarrow \mathtt{fix}\,f \Downarrow_\tau v). \tag{13}$$

Then applying Lemma 34 to (12) and (13) yields $d(e) \lhd_\tau \mathtt{fix}\,f$, *i.e.* $d(e) \in S$, as required to complete Scott induction, this case and the whole proof. □

## 7.3 Extensionality

The formal approximation relations $\lhd_\tau$ is not just any relation. It actually corresponds to a one-sided version of contextual equivalence.

**Definition 30 (Contextual preorder)** Given a type $\tau$, a typing context $\Gamma$ and terms $t, t' \in \mathrm{PCF}_{\Gamma,\tau}$, the *contextual preorder*, written $\Gamma \vdash t \leq_{\mathrm{ctx}} t' : \tau$ is defined to hold if for all evaluation contexts $\mathcal{C}$ such that $\cdot \vdash_{\Gamma,\tau} \mathcal{C} : \gamma$, where $\gamma$ is $\mathtt{nat}$ or $\mathtt{bool}$, and for all values $v \in \mathrm{PCF}_\gamma$,

$$\mathcal{C}[t] \Downarrow_\gamma v \Rightarrow \mathcal{C}[t'] \Downarrow_\gamma v.$$

As for contextual equivalence, we write $t \leq_{\mathrm{ctx}} t' : \tau$ for $\cdot \vdash t \leq_{\mathrm{ctx}} t' : \tau$, *i.e.* if $t$ and $t'$ are closed. *

**Proposition 35 (Contextual preorder corresponds to formal approximation)** *For all PCF types $\tau$ and all closed terms $t_1, t_2 \in \mathrm{PCF}_\tau$*

$$t_1 \leq_{\mathrm{ctx}} t_2 : \tau \Leftrightarrow [\![t_1]\!] \lhd_\tau t_2.$$

*

Before proving this property, we need another characterisation of contextual preorder.

**Lemma 36 (Application contexts)** *For contextual preorder between closed terms,*

*Let $t_1, t_2$ be closed terms of type $\tau$. Then $t_1 \leq_{\mathrm{ctx}} t_2 : \tau$ if and only if, for every term $f : \tau \to \mathtt{bool}$,*

$$f\,t_1 \Downarrow_{\mathtt{bool}} \mathtt{true} \Rightarrow f\,t_2 \Downarrow_{\mathtt{bool}} \mathtt{true}.$$

*

Proof For the "only if" direction, simply note that

$$\cdot \vdash_{\cdot,\tau} f\, - : \texttt{bool}$$

and so by the definition of contextual preorder,

$$f\, t_1 = (f\, -)[t_1] \Downarrow_{\texttt{bool}} \texttt{true} \Rightarrow f\, t_2 = (f\, -)[t_2] \Downarrow_{\texttt{bool}} \texttt{true}$$

For the other direction, assume we are given an arbitrary context $\mathcal{C}$ and a value $v$, and assume $\cdot \vdash_{\Gamma,\tau} \mathcal{C} : \gamma$ and $\mathcal{C}[t] \Downarrow_{\gamma} v$. Let us build the function $f$ as follows. First, define $c : \tau \to \gamma$ as $\texttt{fun}\, x{:}\tau.\ \mathcal{C}[x]$. Then take $g : \gamma \to \tau$ which returns $\texttt{true}$ if and only if its argument is equal to $v$. This function is easily defined in PCF – after all, the language is Turing-complete, so we can certainly code a function that tests whether its argument is a given boolean $\texttt{true}$ or $\texttt{false}$, or a given natural number $\underline{n}$. Given these, let $f \overset{\text{def}}{=} \texttt{fun}\, x{:}\tau.\ g\,(c\, x)$. Then, $f\, t \Downarrow_{\texttt{bool}} \texttt{true}$ if and only if $c\, t \Downarrow_{\gamma} v$, *i.e.* if and only if $\mathcal{C}[t] \Downarrow_{\gamma} v$. Now we can use our assumption for this $f$, and conclude. □

Proof (Proposition 35) Assume $[\![t_1]\!] \lhd_{\tau} t_2$. For any $f \in \mathrm{PCF}_{\tau\to\texttt{bool}}$, by the fundamental property of $\lhd$ we have $[\![f]\!] \lhd_{\tau\to\texttt{bool}} t$, which by definition of $\lhd_{\tau\to\texttt{bool}}$ implies that

$$[\![t\, t_1]\!] = [\![t]\!]\,([\![t_1]\!]) \lhd_{\texttt{bool}} t\, t_2. \tag{14}$$

So if $t\, t_1 \Downarrow_{\texttt{bool}} \texttt{true}$, then $[\![t\, t_1]\!] = \texttt{true}$ (by soundness) and hence by definition of $\lhd_{\texttt{bool}}$ from (14) we get $t\, t_2 \Downarrow_{\texttt{bool}} \texttt{true}$. Using the characterisation of Lemma 36, we finally obtain $t_1 \leq_{\text{ctx}} t_2 : \tau$.

This establishes the right-to-left implication. For the converse, it is enough to prove

$$(d \lhd_{\tau} t_1 \wedge t_1 \leq_{\text{ctx}} t_2 : \tau) \Rightarrow d \lhd_{\tau} t_2. \tag{15}$$

For then if $t_1 \leq_{\text{ctx}} t_2 : \tau$, since $[\![t_1]\!] \lhd_{\tau} t_1$ (by the fundamental property), (15) implies $[\![t_1]\!] \lhd_{\tau} t_2$. Property (15) follows by induction on the structure of the type $\tau$. Indeed, if $\tau = \texttt{nat}$ or $\texttt{bool}$, then $t_1 \leq_{\text{ctx}} t_2 : \tau$ implies, using the context $-$, that

$$\forall v. : \tau\, (t_1 \Downarrow_{\tau} v \Rightarrow t_2 \Downarrow_{\tau} v)$$

from which (15) follows by Lemma 34. If $t_1 \leq_{\text{ctx}} t_2 : \tau \to \tau'$, then also

$$t_1\, t \leq_{\text{ctx}} t_2\, t : \tau'$$

for any $t : \tau$, by taking an evaluation context $\mathcal{C}$ to $\mathcal{C}[-\ t]$. Thus, we can apply the induction hypothesis for $\tau'$ to conclude. □

This equivalence allows us to transfer the extensionality properties enjoyed by the domain partial orders $\sqsubseteq$ to the contextual preorder, as follows.

**Proposition 37 (Extensionality properties of contextual preorder)** *For* $\gamma = \mathtt{bool}$ *or* $\mathtt{nat}$, $t_1 \leq_{\mathrm{ctx}} t_2 : \tau$ *holds if and only if*

$$\forall v.\, (t_1 \Downarrow_\gamma v \Rightarrow t_2 \Downarrow_\gamma v).$$

*At a function type* $\tau \to \tau'$, $t_1 \leq_{\mathrm{ctx}} t_2 : \tau \to \tau'$ *holds if and only if*

$$\forall t \in \mathrm{PCF}_\tau\,.\, (t_1\, t \leq_{\mathrm{ctx}} t_2\, t : \tau'). \qquad *$$

Proof The 'only if' directions are easy consequences of the definition of the contextual preorder.

For the 'if' direction in case $\tau = \mathtt{bool}$ or $\mathtt{nat}$, for any value $v$ we have

$$\begin{aligned} [\![t_1]\!] = [\![v]\!] &\Rightarrow t_1 \Downarrow_\tau v && \text{by adequacy} \\ &\Rightarrow t_2 \Downarrow_\tau v && \text{by assumption} \end{aligned}$$

and hence $[\![t_1]\!] \lhd_\tau t_2$ by definition of $\lhd$ at these ground types. We conclude by Proposition 35.

For the 'if' direction in case of a function type $\tau \to \tau'$, we have

$$\begin{aligned} d \lhd_\tau t &\Rightarrow [\![t_1]\!](d) \lhd_{\tau'} t_1\, t && \text{since } [\![t_1]\!] \lhd_\tau t_1 \\ &\Rightarrow [\![t_1]\!](d) \lhd_{\tau'} t_2\, t && \text{by (15), since } t_1\, t \leq_{\mathrm{ctx}} t_2\, t : \tau' \text{ by assumption} \end{aligned}$$

and hence $[\![t_1]\!] \lhd_{\tau \to \tau'} t_2$ by definition of $\lhd$ at type $\tau \to \tau'$. So once again we can Proposition 35 to get the desired conclusion. □

## 7.4 Exercises

**Exercise 15** Show Lemmas 32 and 34.

**Exercise 16** For any PCF type $\tau$ and any closed terms $t_1, t_2 \in \mathrm{PCF}_\tau$, show that

$$\forall v.\, (t_1 \Downarrow_\tau v \Leftrightarrow t_2 \Downarrow_\tau v) \Rightarrow t_1 \cong_{\mathrm{ctx}} t_2 : \tau. \tag{16}$$

[Hint: combine Proposition 35 with Lemma 34.]

**Exercise 17** Use (16) to show that $\beta$-conversion is valid up to contextual equivalence in PCF, in the sense that for all $t, u$ such that $x{:}\tau \vdash t : \tau'$ and $\vdash u : \tau$, we have

$$(\mathtt{fun}\, x{:}\tau.\, t)\, u \cong_{\mathrm{ctx}} t[u/x] : \tau'.$$

**Exercise 18** Is the converse of (16) valid at ground types? At function types?
[Hint: recall the extensionality property at function types (Proposition 37) and consider the terms $\Omega_{\mathtt{nat}\to\mathtt{nat}}$ and $\mathtt{fun}\, x{:}\mathtt{nat}.\ \Omega_{\mathtt{nat}}$ (defined in Exercise 14), of type $\mathtt{nat} \to \mathtt{nat}$.]

# 8 Full abstraction

## 8.1 Failure of full abstraction

As we saw in Theorem 24, the adequacy property implies that contextual equivalence of two PCF terms can be proved by showing that they have equal denotations: $[\![t_1]\!] = [\![t_2]\!] \in [\![\tau]\!] \Rightarrow t_1 \cong_{\mathrm{ctx}} t_2 : \tau$. In general one says that a denotational semantics is said to be fully abstract if contextual equivalence *coincides* with equality of denotation.

**Definition 31 (Full abstraction)** A denotational model is *fully abstract* if

$$t_1 \cong_{\mathrm{ctx}} t_2 : \tau \Rightarrow [\![t_1]\!] = [\![t_2]\!] \in [\![\tau]\!]$$

*

Unfortunately this is not the case for the denotational semantics of PCF using domains and continuous functions: *there are contextually equivalence PCF terms with unequal denotations.* In other words, the domain model of PCF is *not fully abstract.* The classic example demonstrating this failure is due to Plotkin [3] and involves the *parallel-or* function.

**Definition 32 (Parallel or)** The *parallel or* function $\mathrm{por} : \mathbb{B}_\perp \times \mathbb{B}_\perp \to \mathbb{B}_\perp$ is defined as given by the following table:

| por | true | false | ⊥ |
|---|---|---|---|
| true | true | true | true |
| false | true | false | ⊥ |
| ⊥ | true | ⊥ | ⊥ |

*

Contrast por with the following 'sequential-or' function.

**Definition 33 (Left sequential or)** The (left) sequential or function $\mathrm{or} : \mathbb{B}_\perp \times \mathbb{B}_\perp \to \mathbb{B}_\perp$ is defined as

$$\mathrm{or} \overset{\mathrm{def}}{=} [\![\mathtt{fun}\, x\colon \mathtt{bool}.\ \mathtt{fun}\, y\colon \mathtt{bool}.\ \mathtt{if}\ x\ \mathtt{then}\ \mathtt{true}\ \mathtt{else}\ y]\!]$$

It is given by the following table:

| or | true | false | ⊥ |
|---|---|---|---|
| true | true | true | true |
| false | true | false | ⊥ |
| ⊥ | ⊥ | ⊥ | ⊥ |

*

Both functions give the usual boolean 'or' function when restricted to $\mathbb{B}$, but differ in their behaviour at arguments involving the element $\perp$ denoting 'non-termination'. Note that $\mathrm{por}(d_1, d_2) = \mathrm{true}$ if *either* of $d_1$ or $d_2$ is true even if the other argument is $\perp$; whereas $\mathrm{or}(\perp, d_2) = \perp$ irrelevant of $d_2$.

As noted in the definition, or is *definable*, in the sense that there is a closed PCF term $t : \mathtt{bool} \to (\mathtt{bool} \to \mathtt{bool})$ with $[\![t]\!] = \mathrm{or}$. This term tests whether its first argument is `true` or `false`, and so diverges if that first argument diverges, irrespective of its second argument.

By contrast, for por we have the following, first proven by Plotkin [3] – for a slightly different function.

**Theorem 38 (Undefinability of parallel or)** *There is no closed PCF term*

$$t : \mathtt{bool} \to \mathtt{bool} \to \mathtt{bool}$$

*satisfying*

$$[\![t]\!] = \mathrm{por} : \mathbb{B}_\perp \to \mathbb{B}_\perp \to \mathbb{B}_\perp \ . \qquad *$$

We will not give the proof of this proposition here. The original proof by Plotkin [3] operates via an 'Activity Lemma', but there are alternative approaches using 'stable' continuous functions [5, p 181], or using 'logical relations' [6].

In any case, the key idea is that evaluation in PCF proceeds *sequentially*. So whatever $t$ is, evaluation of $t\ u_1\ u_2$ must at some point involve full evaluation of either $u_1$ or $u_2$ ($t$ cannot ignore its arguments if it is to return `true` in some cases and `false` in others); whereas an algorithm to compute por at a pair of arguments must compute the values of those arguments 'in parallel' in case one diverges whilst the other yields the value `true`.

One can exploit the undefinability of por in PCF to manufacture a pair of contextually equivalent closed terms in PCF with unequal denotations, and thus prove that our denotational semantics is not fully abstract. Indeed, define, for $b \in \{\mathtt{true}, \mathtt{false}\}$, the following program ($\Omega$ has been defined in Example 18):

```
T_b ≝ fun f: bool → (bool → bool).
        if(f true Ω_bool) then
          if (f Ω_bool true) then
            if (f false false) then Ω_bool else b
          else Ω_bool
        else Ω_bool
```

**Theorem 39 (Failure of full abstraction)** *The denotational model given in Section 6, using domains and continuous functions, is not fully abstract.*

*More precisely, the two terms $T_{\texttt{true}}$ and $T_{\texttt{false}}$ are contextually equivalent but have different denotations:*

$$T_{\texttt{true}} \cong_{\text{ctx}} T_{\texttt{false}} : (\texttt{bool} \to \texttt{bool} \to \texttt{bool}) \to \texttt{bool}$$

$$[\![T_{\texttt{true}}]\!] \neq [\![T_{\texttt{false}}]\!] \in (\mathbb{B} \to \mathbb{B} \to \mathbb{B}) \to \mathbb{B} \qquad \star$$

Proof From the definition of por in Definition 32 and the definition of $[\![-]\!]$ in Definitions 24 to 26, it is not hard to see that

$$[\![T_b]\!]\,(\text{por}) = [\![b]\!]$$

Thus $[\![T_{\texttt{true}}]\!]\,(\text{por}) \neq [\![T_{\texttt{false}}]\!]\,(\text{por})$ and therefore $[\![T_{\texttt{true}}]\!] \neq [\![T_{\texttt{false}}]\!]$.

To see that $T_{\texttt{true}} \cong_{\text{ctx}} T_{\texttt{false}} : (\texttt{bool} \to \texttt{bool} \to \texttt{bool}) \to \texttt{bool}$ we use the extensionality results of Proposition 37. Thus, we have to show for all $t : \texttt{bool} \to \texttt{bool} \to \texttt{bool}$ and $v \in \{\texttt{true}, \texttt{false}\}$ that

$$T_{\texttt{true}}\, t \Downarrow_{\texttt{bool}} v \Leftrightarrow T_{\texttt{false}}\, t \Downarrow_{\texttt{bool}} v. \qquad (17)$$

But the definition of $T_b$ is such that $T_b\, t \Downarrow_{\texttt{bool}} v$ only holds if

$$t\,\texttt{true}\,\Omega_{\texttt{bool}} \Downarrow_{\texttt{bool}} \texttt{true} \qquad t\,\Omega_{\texttt{bool}}\,\texttt{true} \Downarrow_{\texttt{bool}} \texttt{true}$$

$$t\,\texttt{false}\,\texttt{false} \Downarrow_{\texttt{bool}} \texttt{false}.$$

By the soundness property (Theorem 28), this means that

$$[\![t]\!]\,(\text{true})(\bot) = \text{true} \qquad [\![t]\!]\,(\bot)(\text{true}) = \text{true} \qquad [\![t]\!]\,(\text{false})(\text{false}) = \text{false}.$$

(Recall from Exercise 14 that $[\![\Omega]\!] = \bot$.) It follows in that case that the continuous function $[\![t]\!] : \mathbb{B}_\bot \to \mathbb{B}_\bot \to \mathbb{B}_\bot$ coincides with por (see Exercise 19). Thus, such a $t$ cannot exist, by Theorem 38. Therefore, (17) is trivially satisfied for all $t$, and thus $T_{\texttt{true}}$ and $T_{\texttt{false}}$ are indeed contextually equivalent. □

## 8.2 Beyond full abstraction failure

This failure of full abstraction can be understood in three different ways. First, we can see it as a failure of PCF: the language is unable to express objects that naturally appear in the semantics. Second, we can see it as a failure of contexts, which are not expressive enough to distinguish terms which are semantically different. For instance, we could wish to have a context that is able to separate the two terms $T_{\texttt{true}}$ and $T_{\texttt{false}}$ above. Third, we can see it as a failure of the model, which does not adequately capture contextual equivalence. In particular, it contains "too many" elements, some of which – such as por – should be ruled out because they do not correspond to behaviour expressible by a PCF program. All three approaches are valid, and lead to different ways to "correct" the full abstraction failure.

### Full abstraction for PCF+por

The failure of full abstraction for the denotational semantics of PCF can be repaired by extending PCF with extra terms for those elements of the domain-theoretic model that are not definable in the language as originally given. We have seen that por is one such element 'missing' from PCF, and a remarkable result[6] is that this is the *only* thing we need add to PCF to obtain full abstraction. This extension is defined formally in Fig. 5.

Terms: $$t ::= \cdots \mid \texttt{por}(t, t)$$

$\boxed{\Gamma \vdash t : \tau}$

$$\ldots \qquad \textsc{Por}\ \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash \texttt{por}(t_1, t_2) : \tau}$$

$\boxed{t \Downarrow_\tau v}$

$$\textsc{PorL}\ \frac{t_1 \Downarrow_{\texttt{bool}} \texttt{true}}{\texttt{por}(t_1, t_2) \Downarrow_{\texttt{bool}} \texttt{true}} \qquad \textsc{PorR}\ \frac{t_2 \Downarrow_{\texttt{bool}} \texttt{true}}{\texttt{por}(t_1, t_2) \Downarrow_{\texttt{bool}} \texttt{true}}$$

$$\textsc{PorF}\ \frac{t_1 \Downarrow_{\texttt{bool}} \texttt{false} \qquad t_2 \Downarrow_{\texttt{bool}} \texttt{false}}{\texttt{por}(t_1, t_2) \Downarrow_{\texttt{bool}} \texttt{false}}$$

Figure 5: PCF+`por`

The proof of this result, just like that of full abstraction failure, are out of the scope of these notes, see Gunter [5] or Curien [7] for proofs.

**Theorem 40 (Full abstraction for PCF+`por`)** *If we extend the semantics of PCF to PCF+*`por` *with*

$$[\![\texttt{por}]\!] = \mathrm{por}$$

*the resulting denotational semantics is fully abstract.* $\star$

[6] Shown in the original Plotkin [3] for a more expressive 'parallel conditional', and refined later to parallel or, see *e.g.* Curien [7].

### Fully abstract semantics for PCF

The evaluation of PCF terms involves a form of 'sequentiality' which is not reflected in the denotational semantics of PCF using domains and continuous functions: the continuous function por does not denote any PCF term and this results in a mismatch between denotational equality and contextual equivalence. But what precisely does 'sequentiality' mean in general? Can we characterise it in an abstract way, independent of the particular syntax of PCF terms, and hence give a more refined form of denotational semantics that *is* fully abstract for contextual equivalence for PCF (and for other types of language besides the simple, pure functional language PCF)? These questions have driven the development of domain theory and denotational semantics since the appearance of Plotkin [3]: see the survey by Hyland and Ong [8], for example.

A first step by Berry [9] was to refine the domain model to so-called dI-domains – and stable functions –, so that por does not belong to the semantic type $\mathbb{B}_\perp \to \mathbb{B}_\perp \to \mathbb{B}_\perp$. However, other, more complicated higher-order functions are allowed in the semantics without being definable in PCF, and this model is still not fully abstract.

It is only in the 90s that definitive answers started to emerge even for such an apparently simple language as PCF. O'Hearn and Riecke [10] construct a fully abstract model of PCF by using logical relations to characterise the definable elements of the standard model. Although this does provide a solution, it does not seem to give much insight into the nature of sequential computation. By contrast, Abramsky, Jagadeesan, and Malacaria [11] and Hyland and Ong [8] solve the problem by introducing a radically different approach to giving semantics to programming languages, based upon the idea of viewing program execution as a two-player game between the program and the environment. See Abramsky et al. [12] and Hyland [13] for introductions to these game semantics.

### Undecidability of contextual equivalence

Finally, a striking negative result by Loader should be mentioned. Note that the material in Section 8.1 does not depend upon the presence of numbers and arithmetic in PCF. Let $\mathrm{PCF}_{\mathtt{bool}}$ denote the fragment of PCF only involving `bool` and the function types formed from it, `true`, `false`, conditionals, variables, function abstraction and application, and a primitive divergent term $\Omega_{\mathtt{bool}} : \mathtt{bool}$.

Since $\mathbb{B}_\perp$ is a finite domain and since the function domain formed from finite domains is again finite, the domain associated to each $\mathrm{PCF}_{\mathtt{bool}}$ type is finite. Element in these domains can be seen as some sort of higher-order "truth tables", akin to those of Section 8.1. A further simplification arises from the fact that if the domains $D$ and $D'$ are finite, then they contain no non-trivial chains and hence the continuous functions $D \to D'$ are just the monotone functions.

Since there are finitely many semantic terms at each type, there are also finitely many different equivalence classes up to contextual equivalence. Given these finiteness properties, and the terribly simple nature of the language, one might hope that the following questions are decidable (uniformly in the $\mathrm{PCF}_{\texttt{bool}}$ type $\tau$):

- Which elements of $[\![\tau]\!]$ are definable by $\mathrm{PCF}_{\texttt{bool}}$ terms?
- When are two $\mathrm{PCF}_{\texttt{bool}}$ of type $\tau$ contextually equivalent?

Quite remarkably, Loader [14] shows that these are recursively undecidable questions. Thus, while we can compute the denotational semantics of terms of $\mathrm{PCF}_{\texttt{bool}}$ in the domain model, and test their equality, there is no hope to get a fully abstract model, even for $\mathrm{PCF}_{\texttt{bool}}$, in which we can effectively compute denotations and test elements for equality. This puts a strong limitation as to how “concrete” fully abstract models can be.

However, if one’s goal is to develop tools to show contextual equivalence of programs, instead of looking for a one-size-fits all domain which exactly captures contextual equivalence with its denotational semantics, one can try and develop other tools. A successful approach to this is *applicative bisimilarity* [15], a relation which captures contextual equivalence but is much more amenable to proofs in particular cases, although it remains undecidable even in simple cases by Loader’s result.

## 8.3 Exercises

**Exercise 19** Suppose that a monotonic function $p\colon\colon \mathbb{B}_\perp \to \mathbb{B}_\perp \to \mathbb{B}_\perp$ satisfies

$$p(\mathrm{true})(\perp) = \mathrm{true} \qquad p(\perp)(\mathrm{true}) = \mathrm{true} \qquad p(\mathrm{false})(\mathrm{false}) = \mathrm{false}.$$

Show that $p = \mathrm{por}$, by showing that $p(d_1, d_2) = \mathrm{por}(d_1)(d_2)$, for all $d_1, d_2 \in \mathbb{B}_\perp$.

**Exercise 20** Show that even though there are overlapping rules in Fig. 5, nevertheless the evaluation relation for PCF+`por` is still deterministic (in the sense of Proposition 23).

**Exercise 21** Give the axioms and rules for an inductively defined transition relation for PCF+`por`. This should take the form of a binary relation $t \rightsquigarrow t'$ between closed PCF+`por` terms. It should satisfy

$$t \Downarrow v \Leftrightarrow t \rightsquigarrow^\star v$$

(where $\rightsquigarrow^\star$ is the reflexive-transitive closure of $\rightsquigarrow$).

# References

[1] Lawrence C. Paulson. *Logic and computation - interactive proof with Cambridge LCF*. Vol. 2. Cambridge tracts in theoretical computer science. Cambridge University Press, 1987. ISBN: 978-0-521-34632-0.

[2] Dana S. Scott. "A Type-Theoretical Alternative to ISWIM, CUCH, OWHY". In: *Theor. Comput. Sci.* 121.1&2 (1993), pp. 411–440. DOI: 10.1016/0304-3975(93)90095-B. URL: https://doi.org/10.1016/0304-3975(93)90095-B.

[3] Gordon D. Plotkin. "LCF Considered as a Programming Language". In: *Theor. Comput. Sci.* 5.3 (1977), pp. 223–255. DOI: 10.1016/0304-3975(77)90044-5. URL: https://doi.org/10.1016/0304-3975(77)90044-5.

[4] Ketan D Mulmuley. *Full abstraction and semantic equivalence*. 1986. MIT Press.

[5] Carl Gunter. *Semantics of programming languages: structures and techniques*. MIT press, 1992.

[6] Kurt Sieber. "Reasoning about sequential functions via logical relations". In: *Applications of categories in computer science* 177 (1992), pp. 258–269.

[7] Pierre-Louis Curien. *Categorical combinators, sequential algorithms, and functional programming*. Springer Science & Business Media, 2012.

[8] J. M. E. Hyland and C.-H. Luke Ong. "On Full Abstraction for PCF: I, II, and III". In: *Inf. Comput.* 163.2 (2000), pp. 285–408. DOI: 10.1006/INCO.2000.2917.

[9] Gérard Berry. "Stable Models of Typed lambda-Calculi". In: *Automata, Languages and Programming, Fifth Colloquium, Udine, Italy, July 17-21, 1978, Proceedings*. Ed. by Giorgio Ausiello and Corrado Böhm. Vol. 62. Lecture Notes in Computer Science. Springer, 1978, pp. 72–89. DOI: 10.1007/3-540-08860-1_7. URL: https://doi.org/10.1007/3-540-08860-1%5C_7.

[10] Peter W. O'Hearn and Jon G. Riecke. "Kripke Logical Relations and PCF". In: *Inf. Comput.* 120.1 (1995), pp. 107–116. DOI: 10.1006/INCO.1995.1103.

[11] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. "Full Abstraction for PCF". In: *Inf. Comput.* 163.2 (2000), pp. 409–470. DOI: 10.1006/INCO.2000.2930.

[12] Samson Abramsky et al. "Semantics of interaction: an introduction to game semantics". In: *Semantics and Logics of Computation* 14.1 (1997).

[13] Martin Hyland. "Game semantics". In: *Semantics and logics of computation* 14 (1997), p. 131.

[14] Ralph Loader. "Finitary PCF is not decidable". In: *Theor. Comput. Sci.* 266.1-2 (2001), pp. 341–364. DOI: 10.1016/S0304-3975(00)00194-8.

[15] S. Abramsky. “The Lazy λ-Calculus”. In: *Research Topics in Functional Programming*. Ed. by D. Turner. Addison Wesley, 1990, pp. 65–117.