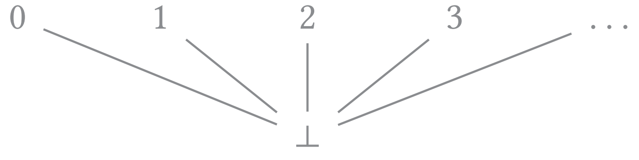
Domain theory and denotational semantics

by

Tom de Jong

Lecture notes and exercises for the Midlands Graduate School (MGS)

2–6 April 2023, Birmingham, UK



School of Computer Science
University of Nottingham

February–March 2023

Abstract

Denotational semantics aims to understand computer programs by assigning mathematical meaning to the syntax of a programming language. In this course we will study a simple functional programming language called PCF. Notably, this language has general recursion through a fixed point operator. This means a simple denotational semantics based on sets is not suitable. Instead, we interpret the types of PCF as certain partially ordered sets leading to domain theory and Scott's model of PCF in particular. The central theorems of soundness and computational adequacy, formulated and proved by Plotkin, then tell us that a PCF program computes to a value if and only if their interpretations in the model are equal.

Acknowledgements

It is my pleasure to thank Martín Escardó who introduced me to the beautiful subjects of denotational semantics and domain theory. I am also grateful for his comments on earlier drafts of these notes.

Additionally, I wish to express my sincere thanks to the teaching assistants Ayberk Tosun and Josh Chen for proof-reading and suggesting several revisions.

Finally, I am grateful to the *MGS 2023* participants Florian Frank, Jonas Forster, Matt Russell and Paul Levy whose comments have led to improvements in these notes.

Contents

Chapter 1

Introduction

Denotational semantics, as pioneered by Scott and Strachey [Sco70; SS71], aims to understand and reason about computer programs by assigning mathematical meaning to the syntax of a programming language.

While other choices for denotational semantics are possible, e.g. using games [Abr97; Hyl97] or realizability [Lon95], these notes employ a denotational semantics based on *domain theory* to study the functional programming language *PCF* [Plo77; Sco93].

The syntax (the types and terms) of PCF will be interpreted as certain kinds of partially ordered sets and so-called continuous maps between them. But a programming language is more than just its syntax: it should compute. The *operational semantics* of PCF specify its computational behaviour by determining a reduction strategy for terms.

Following [Esc07a], we might summarise as:

> Operational semantics is about *how* we compute.
>
> Denotational semantics is about *what* we compute.

The central theorems of *soundness* and *computational adequacy*, formulated and proved by Plotkin [Plo77], then tell us that a PCF program computes to a value if and only if the denotational semantics of the program and the value are equal. Put differently, we might say that the operational and denotational semantics of PCF are "in sync".

1.1 Aims

We hope that these notes provide a self-contained and accessible introduction to domain-theoretic denotational semantics for graduate students in theoretical computer science.

Domain theory [AJ94; GHK+03] is a fruitful mathematical subject with applications outside the semantics of programming languages, e.g. in algebra, higher-type computability [LN15] and topology [GHK+03].

For some, domain theory may be fairly abstract however, and for this reason we have chosen to motivate the domain-theoretic definitions and constructions by appealing to its application of modelling the programming language PCF.

In our investigations we are not only introduced to basic domain theory, but, when proving computational adequacy, also to the fundamental technique of logical relations.

Moreover, although no knowledge of category theory is required for these notes, there are several exercises touching on category theory which hopefully provide an illustration of abstract categorical concepts for those already familiar with category theory, or, alternatively, might encourage those unfamiliar to study it.

More generally, we hope that this course will be taken up as an invitation to explore the wonderful interplay between mathematics and computer science. A striking example of this is Escardó's collection of "seemingly impossible" programs that perform fast exhaustive search on infinite datatypes (see [Esc07b] and the references therein). While the programs can be understood without knowledge of domain theory, their conception and proofs of correctness do rely on domain theory and topology.

1.2 Exercises

The exercises are interspersed in the text, but each chapter ends with a list of its exercises for reference. There are 25 exercises in total.

1.3 References

Our treatment is largely based on Streicher's account [Str06], although we have included several examples and proofs in an attempt to make our notes more accessible. Moreover, at times, we deviate from Streicher's treatment and follow Hart's Agda formalisation [Har20] instead, e.g. we only consider well-typed terms and no "raw" terms and include Lemma 5.10.

Hart's Agda formalisation is the result of a final-year *MSci* project building on our constructive and predicative account of domain theory in the topical univalent foundations (also known as homotopy type theory) [dJon22]. For accessibility reasons, these notes use a classical set-theoretic foundation however.

1.4 Further reading

A natural resource for further reading is the aforementioned textbook [Str06]. Additionally, one may wish to consult the textbooks [Win93; Gun92] or the lecture notes [Plo83; PWF12] for more on domain-theoretic denotational semantics. For more on general domain theory, we recommend [AJ94; GHK+03].

Chapter 2

PCF and its operational semantics

PCF (Programming Computable Functions) [Plo77] is a call-by-name [Gun92; Rie93] typed functional programming language with general recursion. We can think of PCF as a simpler, well-behaved fragment of a modern functional programming language such as Haskell [Mar+10]. The point of PCF is not to be a particularly convenient or rich programming language. Instead, it's meant to be a simple and principled language enabling us to study it from a mathematical viewpoint without having to deal with complex features that you might find in modern, real-world programming languages. The techniques that we will employ could, with some effort, be extended to more complex programming languages however, see e.g. [Plo83].

At the same time, it should be mentioned that PCF is not quite a toy language and has a rather rich theory. For example, it captures Kleene–Kreisel higher-type computability [LN15] and an extension of PCF—with parallel-or and $\exists$ (see [Str06] for details)—can simulate recursively defined datatypes (such as trees and lists) [Str94].

2.1 PCF

We describe the syntax of PCF and its small-step operational semantics which describe how to compute in PCF.

Definition 2.1 (Types of PCF, **nat**, $\sigma \Rightarrow \tau$)**.** The *types of PCF* are inductively defined as:

(i) **nat** is the *base type* of PCF, and
(ii) if σ and τ are types of PCF, then we have the *function type* $\sigma \Rightarrow \tau$.

Moreover, as usual, we will write $\sigma \Rightarrow \tau \Rightarrow \rho$ for $\sigma \Rightarrow (\tau \Rightarrow \rho)$.

From now on, such inductive definitions will be presented in the following style:

$$\frac{}{\textbf{nat} \text{ is a type of PCF}} \qquad \frac{\sigma \text{ is a type of PCF} \qquad \tau \text{ is a type of PCF}}{(\sigma \Rightarrow \tau) \text{ is a type of PCF}}$$

For each type σ, we assume to have a countably infinite set of typed variables, typically denoted by $\mathtt{x} : \sigma$, $\mathtt{y} : \sigma$, or $\mathtt{x}_1 : \sigma$, $\mathtt{x}_2 : \sigma$, etc.

Definition 2.2 (Context, Γ)**.** A *context* Γ is a list of variables:

$$\Gamma = [\mathtt{x}_0 : \sigma_0, \mathtt{x}_1 : \sigma_1, \ldots, \mathtt{x}_{n-1} : \sigma_{n-1}].$$

For $n = 0$, we get the *empty context*: an empty list with no variables.

We are now ready to define the (well-typed) terms of PCF.

Definition 2.3 (Terms of PCF, $\Gamma \vdash M : \sigma$)**.** We inductively define the *terms M of type σ in context* Γ, written $\Gamma \vdash M : \sigma$, by the following clauses:

$$\frac{}{\Gamma, \mathtt{x} : \sigma, \Delta \ \vdash\ \mathtt{x} : \sigma} \qquad \frac{\Gamma, \mathtt{x} : \sigma \ \vdash\ M : \tau}{\Gamma \ \vdash\ (\lambda \mathtt{x} : \sigma \,.\, M) : \sigma \Rightarrow \tau}$$

$$\frac{\Gamma \ \vdash\ M : \sigma \Rightarrow \tau \qquad \Gamma \ \vdash\ N : \sigma}{\Gamma \ \vdash\ M(N) : \tau} \qquad \frac{\Gamma \ \vdash\ M : \sigma \Rightarrow \sigma}{\Gamma \ \vdash\ \mathtt{fix}_\sigma(M) : \sigma}$$

$$\frac{}{\Gamma \ \vdash\ \mathtt{zero} : \mathbf{nat}} \qquad \frac{\Gamma \ \vdash\ M : \mathbf{nat}}{\Gamma \ \vdash\ \mathtt{succ}(M) : \mathbf{nat}} \qquad \frac{\Gamma \ \vdash\ M : \mathbf{nat}}{\Gamma \ \vdash\ \mathtt{pred}(M) : \mathbf{nat}}$$

$$\frac{\Gamma \ \vdash\ M : \mathbf{nat} \qquad \Gamma \ \vdash\ N_1 : \mathbf{nat} \qquad \Gamma \ \vdash\ N_2 : \mathbf{nat}}{\Gamma \ \vdash\ \mathtt{ifzero}(M, N_1, N_2) : \mathbf{nat}}$$

When Γ is the empty context, we simply write $\vdash M : \sigma$ and we call M a *closed* term or a *program*. Note that programs do not contain any free variables.
We will often write $M\,N$ instead of $M(N)$ to ease readability.

The first three rules above will look familiar to someone who has seen the typed λ-calculus before and basically say that we form functions that we can apply. The three rules on the third row give us natural numbers with a predecessor constructor. We illustrate the remaining rules, those for `fix` and `ifzero`, in Examples 2.7 and 2.8 below.

So far, we only have some terms, but we have not specified any computational behaviour of those terms yet. Definition 2.5 defines a reduction strategy that specifies the computational behaviour of PCF and should help us understand the intended meaning of the terms.

Definition 2.4 (Numeral, $\underline{n}$)**.** For a natural number $n \in \mathbb{N}$, we define the *numeral* $\underline{n}$ in PCF inductively: $\underline{0} \coloneqq \mathtt{zero}$ and $\underline{m+1} \coloneqq \mathtt{succ}\ \underline{m}$.

Definition 2.5 (Small-step operational semantics of PCF, $M \rhd N$)**.** We inductively define when a term M *(small-step) reduces* to another term N (of the same type, in the same context), written $M \rhd N$, by the following inductive clauses:

$$\frac{}{(\lambda\,\mathtt{x} : \sigma\,.\,M)N \rhd M[N/\mathtt{x}]} \qquad \frac{}{\mathtt{fix}_\sigma\ M \rhd M(\mathtt{fix}_\sigma\ M)}$$

$$\frac{}{\mathtt{pred}\ \underline{0} \rhd \underline{0}} \qquad \frac{}{\mathtt{pred}\ \underline{n+1} \rhd \underline{n}}$$

$$\frac{}{\mathtt{ifzero}(\underline{0}, M, N) \rhd M} \qquad \frac{}{\mathtt{ifzero}(\underline{n+1}, M, N) \rhd N}$$

$$\frac{M_1 \rhd M_2}{M_1\,N \rhd M_2\,N} \qquad \frac{M_1 \rhd M_2}{\mathtt{succ}\ M_1 \rhd \mathtt{succ}\ M_2}$$

$$\frac{M_1 \rhd M_2}{\mathtt{pred}\ M_1 \rhd \mathtt{pred}\ M_2} \qquad \frac{M_1 \rhd M_2}{\mathtt{ifzero}(M_1, N_1, N_2) \rhd \mathtt{ifzero}(M_2, N_1, N_2)}$$

Here $M[N/x]$ denotes the result of substituting N for the variable $\mathtt{x}$ in M.

The final three rules are like congruence rules saying that you can reduce a term $\mathtt{succ}\,M$ by reducing the inner term M. The reduction $\mathtt{fix}_\sigma\ M \rhd M(\mathtt{fix}_\sigma\ M)$ corresponds to the idea that $\mathtt{fix}_\sigma\ M$ is a *fixed point* of $M : \sigma \Rightarrow \sigma$. Alternatively, we can think of it as a single unfolding of a recursive definition, as illustrated in Exercise 2.9 and Example 2.11.

Example 2.7. We can translate the program `if (x + 1 == 0) then 5 else 3`, written in pseudocode, to the PCF program $\lambda\,\mathtt{x} : \mathbf{nat}\,.\,\mathtt{ifzero}(\mathtt{succ}\ \mathtt{x}, \underline{5}, \underline{3})$.

The point of `fix` is that it gives us general recursion, as we will explain with an example now.

Example 2.8 (Addition by n in PCF)**.** For a natural number $n \in \mathbb{N}$, consider addition by n as a recursively defined function:

$$\begin{aligned}
&\mathrm{add}_n : \mathbb{N} \to \mathbb{N} \\
&\mathrm{add}_n(0) \quad\ \ \coloneqq n, \\
&\mathrm{add}_n(k+1) \coloneqq \mathrm{add}_n(k) + 1.
\end{aligned}$$

We show how to write this function as a PCF program. We start by slightly rewriting add_n as:

$$\mathrm{add}_n(m) \coloneqq \begin{cases} n & \text{if } m = 0, \\ \mathrm{succ}(\mathrm{add}_n(\mathrm{pred}(m))) & \text{else.} \end{cases} \tag{$*$}$$

Looking at the above, we see that we have most of the analogues readily available in PCF: `ifzero`, `succ` and `pred`, as well as the numeral $\underline{n}$.
We next define the program $F : (\mathbf{nat} \Rightarrow \mathbf{nat}) \Rightarrow (\mathbf{nat} \Rightarrow \mathbf{nat})$ as follows:

$$F \coloneqq \lambda\,\mathtt{f} : (\mathbf{nat} \Rightarrow \mathbf{nat})\,.\,\lambda\,\mathtt{y} : \mathbf{nat}\,.\,\mathtt{ifzero}\big(\mathtt{y}, \underline{n}, \mathtt{succ}(\mathtt{f}(\mathtt{pred}\ \mathtt{y}))\big).$$

In the program F, the variable $\mathtt{y}$ plays the role of m in $(*)$ and $\mathtt{f}$ is like a placeholder for the recursive call.

Mirroring (∗) further, what we want is a program $f : \mathbf{nat} \Rightarrow \mathbf{nat}$ such that f is "equal" to $F\,f$. That is, we want a *fixed point* of F. Hence, we finally define $\mathtt{add}_n$ as:

$$\mathtt{add}_n \coloneqq \mathtt{fix}_{\mathbf{nat}\Rightarrow\mathbf{nat}}\ F.$$

Exercise 2.9. Give sequences of small-step reductions showing that $\mathtt{add}_n\ \underline{0}$ and $\mathtt{add}_n\ \underline{1}$ respectively compute to the numerals $\underline{n}$ and $\underline{n+1}$.

Exercise 2.10. Construct PCF programs $\mathtt{add}, \mathtt{mult} : \mathbf{nat} \Rightarrow \mathbf{nat} \Rightarrow \mathbf{nat}$ implementing addition and multiplication, respectively.

Having general recursion also means that we have non-terminating programs, such as the following one:

Example 2.11. Define $S \coloneqq (\lambda\,\mathrm{x} : \mathbf{nat}\,.\,\mathtt{succ}\ \mathrm{x}) : \mathbf{nat} \Rightarrow \mathbf{nat}$ and consider the PCF program $M \coloneqq \mathtt{fix}_{\mathbf{nat}}(S) : \mathbf{nat}$. Repeatedly applying the small-step operational semantics, we get:

$$\begin{aligned} M &= \mathtt{fix}_{\mathbf{nat}}\ S \\ &\rhd S(\mathtt{fix}_{\mathbf{nat}}\ S) \\ &\rhd \mathtt{succ}(\mathtt{fix}_{\mathbf{nat}}\ S) && = \mathtt{succ}\ M \\ &\rhd \mathtt{succ}(S(\mathtt{fix}_{\mathbf{nat}}\ S)) \\ &\rhd \mathtt{succ}(\mathtt{succ}(\mathtt{fix}_{\mathbf{nat}}\ S)) && = \mathtt{succ}(\mathtt{succ}\ M) \\ &\rhd \ldots \end{aligned}$$

Thus, the term $M : \mathbf{nat}$ never computes to a numeral.

2.2 Big-step operational semantics

In our investigations into the denotational semantics of PCF, we will typically not be interested in intermediate reduction steps, e.g. we don't really care about the whole sequence $\mathtt{ifzero}(\mathtt{pred}\ \underline{3}, \underline{5}, \mathtt{pred}\ \underline{7}) \rhd \mathtt{ifzero}(\underline{2}, \underline{5}, \mathtt{pred}\ \underline{7}) \rhd \mathtt{pred}\ \underline{7} \rhd \underline{6}$; we only care that $\mathtt{ifzero}(\mathtt{pred}\ \underline{3}, \underline{5}, \mathtt{pred}\ \underline{7})$ computes to the numeral $\underline{6}$. The big-step operational semantics of PCF is a way of formalising this desideratum.

Definition 2.12 (Big-step operational semantics of PCF, $M \Downarrow V$)**.** We inductively define when a term M *(big-step) reduces* to another term V (of the same type, in the same context), written $M \Downarrow V$, by the following inductive clauses:

$$\frac{}{\mathrm{x} \Downarrow \mathrm{x}} \qquad\qquad \frac{}{\lambda\,\mathrm{x} : \sigma\,.\,M \Downarrow \lambda\,\mathrm{x} : \sigma\,.\,M}$$

$$\frac{M \Downarrow \lambda x : \sigma \,.\, E \qquad E[N/x] \Downarrow V}{M\,N \Downarrow V} \qquad \frac{M(\mathtt{fix}_\sigma\ M) \Downarrow V}{\mathtt{fix}_\sigma\ M \Downarrow V}$$

$$\frac{}{\underline{0} \Downarrow \underline{0}} \qquad \frac{M \Downarrow \underline{n}}{\mathtt{succ}\ M \Downarrow \underline{n+1}}$$

$$\frac{M \Downarrow \underline{0}}{\mathtt{pred}\ M \Downarrow \underline{0}} \qquad \frac{M \Downarrow \underline{n+1}}{\mathtt{pred}\ M \Downarrow \underline{n}}$$

$$\frac{M \Downarrow \underline{0} \qquad N_1 \Downarrow V}{\mathtt{ifzero}(M, N_1, N_2) \Downarrow V} \qquad \frac{M \Downarrow \underline{n+1} \qquad N_2 \Downarrow V}{\mathtt{ifzero}(M, N_1, N_2) \Downarrow V}$$

Definition 2.14 (Value)**.** A term is a *value* if it is either a variable, a numeral or a λ-abstraction, i.e. it is of the form $\lambda x : \sigma \,.\, N$ for some term N.
Note that the values of type **nat** in the empty context are precisely the numerals.

The reason that we use V in the big-step semantics is the following:

Lemma 2.15. *If $M \Downarrow V$, then V is a value.*

Moreover, values do not reduce any further:

Lemma 2.16. *If V is a value, then*
(i) there is no term N such that $V \rhd N$, and
(ii) whenever $V \Downarrow N$, we have $V = N$.

Thus, the big-step operational semantics reduces a term all the way to a value, forgetting about the intermediary reductions.
Furthermore, the values computed by the big-step operational semantics are unique:

Lemma 2.17 (Big-step is deterministic)**.** *If $M \Downarrow V$ and $M \Downarrow W$, then $V = W$.*

The lemmas are proved by induction on the structure of derivations of $M \Downarrow V$ and inspection of the small-step operational semantics.
Finally, we relate the small-step and big-step operational semantics: We write $\rhd^*$ for the reflexive transitive closure of $\rhd$. That is, we have $M \rhd^* N$ exactly if there is a sequence $M = M_0 \rhd M_1 \rhd \ldots \rhd M_{n-1} = N$. For $n = 0$, this reads $M = N$, and for $n = 1$, it reads $M \rhd N$.

Exercise 2.18. Show that:
(i) if $M \Downarrow V$, then $M \rhd^* V$;
(ii) if $M \rhd N$, then $N \Downarrow V$ implies $M \Downarrow V$ for all values V;
(iii) if $M \rhd^* N$, then $N \Downarrow V$ implies $M \Downarrow V$ for all values V.

For (i) and (ii), use induction on the structure of the derivations; for (iii) you can repeatedly apply (ii).
Conclude that $M \Downarrow V$ if and only if $M \rhd^* V$ for all terms M and values V.

2.3 List of exercises

1. Exercise 2.9: On computing additions using the small-step operational semantics.
2. Exercise 2.10: On defining addition and multiplication in PCF.
3. Exercise 2.18: On relating the small-step and big-step operational semantics.

Chapter 3

Denotational semantics and domain theory

Previously we introduced the operational semantics of PCF. Next, we wish to introduce a *denotational semantics* to PCF. The basic idea is to assign to each type σ of PCF some kind of *mathematical* object $[\![\sigma]\!]$. A program M (i.e. closed term) of type σ is then interpreted as an element $[\![M]\!]$ of $[\![\sigma]\!]$. More generally, we extend the interpretation of types to contexts and $\Gamma \vdash M : \sigma$ will then be interpreted as a certain (ω-continuous) function from $[\![\Gamma]\!]$ to $[\![\sigma]\!]$.

For the denotational semantics we have three desiderata:

- *Compositionality*: This can be summarised as follows:

  The interpretation of a composite is the composite of the interpretations.

  For example, the interpretation of a function type $\sigma \Rightarrow \tau$ will be a certain set of functions from $[\![\sigma]\!]$ to $[\![\tau]\!]$, and if we have programs $M : \sigma \Rightarrow \tau$ and $N : \sigma$, then $[\![M\,N]\!] = [\![M]\!]([\![N]\!])$.
- *Soundness*: We want our interpretation to respect the operational semantics, i.e. if we have terms M and N with $M \Downarrow N$, then their interpretations should be equal.
- *Computational adequacy*: We should be able to use our interpretation to compute, i.e. if M is a program of type **nat** and $[\![M]\!] = n$ for some natural number n, then $M \Downarrow \underline{n}$.

Soundness and computational adequacy will be proved in Section 4.2 and Chapter 5, while compositionality will be a direct consequence of our definitions.

3.1 Towards domain theory

All this leaves the question what kind of mathematical objects we have in mind for interpreting the types of PCF. A first, naive attempt may be to interpret each type σ as a set $[\![\sigma]\!]$ with:

$$\begin{aligned} [\![\mathbf{nat}]\!] &\coloneqq \text{the set } \mathbb{N} \text{ of natural numbers, and} \\ [\![\sigma \Rightarrow \tau]\!] &\coloneqq \text{the set } \{f \colon [\![\sigma]\!] \to [\![\tau]\!]\} \text{ of all functions from } [\![\sigma]\!] \text{ to } [\![\tau]\!]. \end{aligned}$$

There are two problems with this naive approach. The first problem is that $[\![\mathbf{nat}]\!]$ should not only offer natural numbers, but it should also offer an interpretation of programs of type **nat** that do not terminate such as the one in Example 2.11. Right now, this program M would have to be interpreted as some natural number n, forcing us to make some arbitrary choice. Moreover, it is not true that $M \Downarrow \underline{n}$, whatever choice we make, so we would violate computational adequacy.

This problem is easily solved by introducing a new, special element $\perp_\sigma \in [\![\sigma]\!]$ that interprets non-terminating programs of type σ. And indeed, this will be a part of our eventual interpretation.

However, it does not solve the second, more serious problem, namely that, in order to interpret PCF's $\texttt{fix}_\sigma$ we would need every function $f \colon [\![\sigma]\!] \to [\![\sigma]\!]$ to have a fixed point. But this is easily seen to *false*, e.g. consider

$$\begin{aligned} f \colon \mathbb{N} \cup \{\perp\} &\to \mathbb{N} \cup \{\perp\} \\ \perp &\mapsto 0, \\ n &\mapsto n+1. \end{aligned}$$

This is where domain theory comes to the rescue: Instead of using plain sets, we will interpret types as so-called *ω-cppos*: these are *partially ordered* sets with a *least* element $\perp$ and *least upper bounds* of increasing sequences. A function between (the underlying sets of) two ω-cppos is *ω-continuous* if it preserves the order and these least upper bounds. The upshot of restricting to such ω-continuous functions is that these can be shown to have (least) fixed points, thus solving our second problem.

3.2 Basic definitions and the least fixed point theorem

We define ω-cppos (short for: pointed, ω-chain complete posets) and ω-continuous functions between them.

Definition 3.1 (Poset)**.** A *partially ordered set*, or *poset*, is a set X together with a binary relation $\sqsubseteq$ satisfying:

(i) *reflexivity*: for every $x \in X$, we have $x \sqsubseteq x$;
(ii) *transitivity*: for every $x, y, z \in X$, if $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$;
(iii) *antisymmetry*: for every $x, y \in X$, if $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$.

Example 3.2. The natural numbers, rational numbers and real numbers with their usual orderings are all examples of posets.

Definition 3.3 (ω-chain, least upper bound, ω-cpo)**.** Let $(X, \sqsubseteq)$ be a poset.

(i) An ω-chain in X is a subset $\{x_0, x_1, x_2, \ldots\} \subseteq X$ of increasing elements, i.e. we have $x_0 \sqsubseteq x_1$, $x_1 \sqsubseteq x_2$, $x_2 \sqsubseteq x_3$, and so on.
(ii) An *upper bound* of a subset $S \subseteq X$ is an element $x \in X$ such that $s \sqsubseteq x$ for every $s \in S$.
(iii) A *least upper bound* of a subset $S \subseteq X$ is an upper bound $x \in X$ of S such that for every upper bound $y \in X$ of S we have $x \sqsubseteq y$.
(iv) The poset X is *ω-chain complete* if every ω-chain in X has a least upper bound.

An ω-chain complete poset will be called an *ω-cpo*.

Exercise 3.4. Show that least upper bounds are unique. That is, if x and y are least upper bounds of some subset S of a poset X, then $x = y$.
Hint: Use antisymmetry.

Because least upper bounds are unique, we will speak of *the* least upper bound (of a given subset) and in the case of ω-chain $x_0 \sqsubseteq x_1 \ldots$, we will denote the least upper bound by $\bigsqcup_{n\in\mathbb{N}} x_n$.

Definition 3.5 (Least element, ω-cppo)**.** Let $(X, \sqsubseteq)$ be a poset.

(i) A *least element* of X is an element $x \in X$ such that $x \sqsubseteq y$ for every $y \in X$.
(ii) An ω-cpo is called *pointed* if it has a least element.

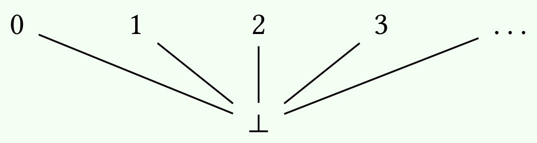
A pointed ω-cpo will be called an *ω-cppo*.

Exercise 3.6. Let X be a poset with a least element x. Show that x is the least upper bound of the empty subset and hence conclude that x must be unique.

Because least elements are unique, we will speak of *the* least element and will typically denote it by $\perp$.

Example 3.7. The set of natural numbers $\mathbb{N}$ with the usual order is *not* an ω-cpo, because the ω-chain $\{0, 1, \ldots\}$ does not have a (least) upper bound.

Example 3.8 ($\mathbb{N}_\perp$)**.** A fundamental example of an ω-cppo is the *flat* ω-cppo $\mathbb{N}_\perp$, which is given by the set $\mathbb{N} \cup \{\perp\}$ ordered as depicted in the following diagram:



Here we interpret a line going up from an element x to an element y as saying that $x \sqsubseteq y$. Thus, the element $\perp$ is the least element and all other elements are unrelated. In particular, we have do *not* have $0 \sqsubseteq 1$ as in the usual ordering of the natural

numbers.
The intuition here is that the partial order $\sqsubseteq$ does not reflect the numerical value, but rather how "defined" an element is with $\bot$ representing an "undefined". In our interpretation of PCF, we will interpret **nat** as $\mathbb{N}_\bot$ and $\bot$ will serve as an interpretation of non-terminating programs.

Exercise 3.9. Show that the ω-chains in $\mathbb{N}_\bot$ are precisely $\{\bot\}$, $\{n\}$ and $\{\bot, n\}$ for a natural number $n \in \mathbb{N}$. Use this to check that $\mathbb{N}_\bot$ is indeed an ω-cpo.

Remark 3.10. It is perhaps a bit arbitrary that to exhibit $\{\bot, n\} \subseteq \mathbb{N}_\bot$ as an ω-chain we have to repeat elements and order them linearly. One way to address this is to replace the notion of ω-chain by that of a *directed* (see Exercise 3.11) subset, and to consider *dcpos*: posets with least upper bounds for all directed subsets.
For general domain theory the notion of dcpo is arguably the better notion (see also [AJ94, Section 2.2.4], but for the denotational semantics of PCF and in particular, the existence of least fixed points (Theorem 3.17), the notion of ω-cpo suffices.

Exercise 3.11. Let $(X, \sqsubseteq)$ be a poset. A subset $S \subseteq X$ is *directed* if it is non-empty and for every two elements $x, y \in S$, there exists an element $z \in S$ with $x \sqsubseteq z$ and $y \sqsubseteq z$.

(i) Show that every ω-chain is a directed subset. Conclude that every poset that has least upper bounds of directed subsets (a so-called *dcpo*) is an ω-cpo.
(ii) In the other direction, use the axiom of choice to show that every ω-cpo has least upper bounds of countable, directed subsets.

We proceed by defining the ω-continuous maps between (pointed) ω-cpos.

Definition 3.12 (ω-continuity)**.** A function $f : A \to B$ between the underlying sets of two ω-cpos A and B is *ω-continuous* if

(i) it is *monotone* (or *order preserving*), i.e. if $x \sqsubseteq y$ in A, then $f(x) \sqsubseteq f(y)$ in B;
(ii) it *preserves least upper bounds of ω-chains*, i.e. if $\{x_0, x_1, \ldots\}$ is an ω-chain in A with least upper bound a, then $f(a)$ is the least upper bound in B of the subset $\{f(x_0), f(x_1), \ldots\}$.

Remark 3.13. It follows from monotonicity that $\{f(x_0), f(x_1), \ldots\}$ is an ω-chain in B and hence we could rewrite the second condition as:

$$f(\textstyle\bigsqcup_{n\in\mathbb{N}} x_n) = \bigsqcup_{n\in\mathbb{N}} f(x_n).$$

It is worthwhile to reflect on the computational intuitions underlying monotonicity and preservation of least upper bounds of ω-chains. If we think of $f : A \to B$ as some computational procedure, then monotonicity says:

more (or better) input leads to more (or better) output,

while the second condition of ω-continuity says:

every output can be patched together from the outputs of approximations.

The idea here is that a computational procedure can only inspect a finite amount of input before returning an output, so that the approximations suffice to determine the output. These intuitions and ideas are nicely illustrated in the following example:

Exercise 3.14. Let C be the poset of finite and infinite binary sequences ordered by prefix. For an infinite sequence α we write $\alpha \upharpoonright n$ for its finite prefix of length n.

(i) Show that C is an ω-cppo.

(ii) Show that a function $f\colon C \to C$ is ω-continuous if and only if for every infinite sequence α we have $f(\alpha) = \bigsqcup_{n\in\mathbb{N}} f(\alpha \upharpoonright n)$.

Actually, monotonicity, item (i) in Definition 3.12, is redundant as the following exercise ask you to check:

Exercise 3.15. Show that if a function f between ω-cpos satisfies item (ii) of Definition 3.12, then f must be monotone.
Hint: If $x \sqsubseteq y$, then $\{x, y\}$ is an ω-chain with least upper bound y.

Example 3.16. The function $f\colon \mathbb{N}_\perp \to \mathbb{N}_\perp$ given by $\perp \mapsto 0$ and $n \mapsto n+1$ is *not* ω-continuous, because it is not monotone: $\perp \sqsubseteq 1$, but $f(\perp) = 0$ is not below $f(1) = 2$ in the ordering of $\mathbb{N}_\perp$.
On the other hand, the function $g\colon \mathbb{N}_\perp \to \mathbb{N}_\perp$ given by $\perp \mapsto \perp$ and $n \mapsto n+1$ is ω-continuous. In fact, it will serve as the interpretation of the PCF program $\lambda\, \mathtt{x} : \mathbf{nat}\ .\ \mathtt{succ}\ \mathtt{x}$ of type $\mathbf{nat} \Rightarrow \mathbf{nat}$.

The point of introducing ω-continuity is the following fundamental result:

Theorem 3.17 (Kleene fixed point theorem). *Every ω-continuous function $f\colon X \to X$ on a pointed ω-cpo X has a least fixed point given by the least upper bound of the ω-chain*

$$\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \ldots.$$

Proof sketch. We write $f^n(\perp)$ for the n-fold application of f to the least element $\perp$. We first check that $\{f^n(\perp) \mid n \in \mathbb{N}\}$ is indeed an ω-chain. Since $\perp$ is the least element, we certainly have $\perp \sqsubseteq f(\perp)$. But now we also have $f(\perp) \sqsubseteq f^2(\perp)$ by monotonicity of f. It follows by induction on n that $f^n(\perp) \sqsubseteq f^{n+1}(\perp)$, as desired. Now we calculate:

$$\begin{aligned}
f\Big(\textstyle\bigsqcup_{n\in\mathbb{N}} f^n(\perp)\Big) &= \bigsqcup_{n\in\mathbb{N}} f(f^n(\perp)) && \text{(since f preserves least upper bounds of ω-chains)} \\
&= \bigsqcup_{n\in\mathbb{N}} f^{n+1}(\perp) && \text{(by definition)} \\
&= \bigsqcup_{n\in\mathbb{N}} f^n(\perp) && \text{(by antisymmetry and calculation)}
\end{aligned}$$

so we have a fixed point, as claimed. That it is the least follows from Exercise 3.18. □

Exercise 3.18 (Park induction)**.** Show that for every $y \in X$ with $f(y) \sqsubseteq y$, we have $\bigsqcup_{n\in\mathbb{N}} f^n(\bot) \sqsubseteq y$. Conclude that $\bigsqcup_{n\in\mathbb{N}} f^n(\bot)$ is indeed the *least* fixed point.

Exercise 3.19. Show that ω-cpos and ω-continuous functions form a *category*, i.e.

1. if X is an ω-cpo, then the identity function $x \mapsto x$ on X is ω-continuous, and
2. if we have ω-continuous functions $f: A \to B$ and $g: B \to C$, then so is their composition (as functions) $g \circ f: A \to C$.

Exercise 3.20 (For those familiar with category theory)**.** Write ω-CPO for the category constructed above and ω-CPPO$_\bot$ for the category of *pointed* ω-cpos and those morphisms of ω-cpos that preserve the least element.
Construct adjunctions

$$\begin{tikzcd}
\mathsf{Set} \arrow[r, bend left] \arrow[r, phantom, "\perp"] & \omega\text{-CPO} \arrow[l, bend left] \arrow[r, bend left] \arrow[r, phantom, "\perp"] & \omega\text{-CPPO}_\bot \arrow[l, bend left]
\end{tikzcd}$$

Hint: The composite functor $\mathsf{Set} \to \omega$-CPPO$_\bot$ takes the set $\mathbb{N}$ to the ω-cppo $\mathbb{N}_\bot$ from Example 3.8.

Remark 3.21. For the interpretation of PCF, we do *not* use the category ω-CPPO$_\bot$, but rather the full subcategory of ω-CPO of pointed ω-cpos. That is, we do *not* restrict to those ω-continuous functions that preserve the least element. For instance, the interpretation of `ifzero` will not necessarily preserve the least element in its second and third arguments. This makes sense, because $\texttt{ifzero}(\underline{0}, M, N) \Downarrow M$ holds even if the program N does not terminate.

3.3 Products: interpreting contexts

As explained in the introduction to this chapter our goal is to interpret the types of PCF as ω-cppos. Thus, we will have an ω-cppo $[\![\sigma]\!]$ for each PCF type σ. Moreover, we wish to extend this interpretation to contexts, which are lists of typed variables. In a context $\Gamma = [\mathrm{x}_0 : \sigma_0, \mathrm{x}_1 : \sigma_1, \dots, \mathrm{x}_n : \sigma_{n-1}]$ the variables themselves are not really important; it's the types that matter. Accordingly, we will interpret such a context as the *product* $[\![\sigma_0]\!] \times [\![\sigma_1]\!] \times \cdots \times [\![\sigma_{n-1}]\!]$ of the interpretations of the types.

The empty context will be interpreted as follows:

Example 3.22 (The one point ω-cppo $\mathbf{1}$)**.** The one point ω-cppo $\mathbf{1}$ is the unique ω-cppo with the singleton $\{\star\}$ as its underlying set.

Definition 3.23 (Binary product of ω-cpos, $A \times B$)**.** Given two ω-cpos A and B, their *binary product* $A \times B$ is defined by taking the cartesian product of their underlying sets with pairwise partial order:

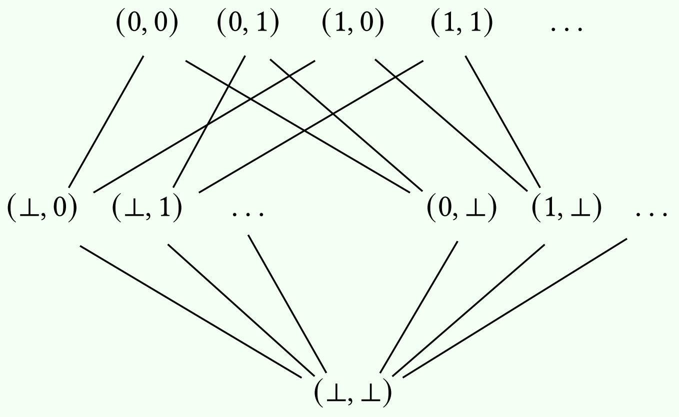
$$(x_1, y_1) \sqsubseteq_{A\times B} (x_2, y_2) :\equiv (x_1 \sqsubseteq_A x_2) \text{ and } (y_1 \sqsubseteq_B y_2).$$

Given an ω-chain $\{(x_0, y_0), (x_1, y_1), \ldots\}$ in $A \times B$, one verifies that $\{x_0, x_1, \ldots\}$ and $\{y_0, y_1, \ldots\}$ are ω-chains in A and B, respectively. Hence, we can consider their least upper bounds $\bigsqcup_{n\in\mathbb{N}} x_n$ and $\bigsqcup_{n\in\mathbb{N}} y_n$ in A and B. One then checks that the least upper bound of the original ω-chain in $A \times B$ is given by the pair of least upper bounds $(\bigsqcup_{n\in\mathbb{N}} x_n, \bigsqcup_{n\in\mathbb{N}} y_n)$.
Finally, if A and B are pointed, then so is $A \times B$ with least element $(\bot_A, \bot_B)$.

Example 3.24. The product of the ω-cppos $\begin{tikzcd}[row sep=small] x \arrow[d, dash] \\ \bot \end{tikzcd}$ and $\begin{tikzcd}[row sep=small] y \arrow[d, dash] \\ \bot \end{tikzcd}$ looks like this:

$$\begin{tikzcd}
 & (x, y) & \\
(x, \bot) \arrow[ur, dash] & & (\bot, y) \arrow[ul, dash] \\
 & (\bot, \bot) \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}$$

Example 3.25 (An illustration of the ω-cppo $\mathbb{N}_\bot \times \mathbb{N}_\bot$).



The construction of least upper bounds of ω-chains in the product ensures:

Lemma 3.26. *Given two ω-cpos A and B, the* projections

$$\begin{aligned} \pi_1 \colon A \times B &\to A & \qquad \pi_2 \colon A \times B &\to B \\ (x, y) &\mapsto x & (x, y) &\mapsto y \end{aligned}$$

are ω-continuous.

Exercise 3.27. Show that a function $f \colon A \times B \to C$ between ω-cpos is ω-continuous if and only if the functions $f(x, -) \colon B \to C$ and $f(-, y) \colon A \to C$ are ω-continuous for every $x \in A$ and $y \in B$.
Thus, ω-continuity of f can be checked in each argument separately.

Exercise 3.28. Prove that if $f\colon C \to A$ and $g\colon C \to B$ are ω-continuous functions between ω-cpos, then so is the induced map

$$\begin{aligned}\langle f, g\rangle\colon C &\to A \times B\\ x &\mapsto (f(x), g(x)).\end{aligned}$$

For those familiar with category theory: conclude that the category ω-CPO has finite products.

3.4 Exponentials: interpreting function types

To interpret the function type $\sigma \Rightarrow \tau$ of PCF using ω-cpos, we are going to construct an ω-cpo of ω-continuous functions: the *exponential*.

Definition 3.29 (Exponential of ω-cpos, B^A)**.** Given two ω-cpos A and B, their *exponential* B^A is defined by equipping the set of ω-continuous functions from A to B with the *pointwise* order:

$$f \sqsubseteq g \coloneqq \forall_{x \in A}\,(f(x) \sqsubseteq_B g(x)).$$

Given an ω-chain $\{f_0, f_1, \ldots\}$ in B^A, one verifies that $\{f_0(x), f_1(x), \ldots\}$ is an ω-chain in B for every $x \in A$. Hence, for every $x \in A$, we can consider the least upper bound $\bigsqcup_{n\in\mathbb{N}} f_n(x)$ in B. One then checks that the function $x \mapsto \bigsqcup_{n\in\mathbb{N}} f_n(x)$ is ω-continuous and moreover, that it is the least upper bound of the original ω-chain in B^A.
Finally, if B is pointed, then so is B^A with least element $x \mapsto \bot$.

We illustrate the exponential with the following example and exercise before proceeding to develop the required machinery for interpreting PCF using ω-cppos and ω-continuous maps.

Example 3.30. Consider the ω-cpo $\mathbf{2} \coloneqq \begin{array}{c}\top\\ |\\ \bot\end{array}$. Writing $[x, y]$ for the function $f\colon \mathbf{2} \to \mathbf{2}$ given by $f(\bot) \coloneqq x$ and $f(\top) \coloneqq y$, we picture the exponential $\mathbf{2}^{\mathbf{2}}$ as:

$$\begin{array}{c}[\top, \top]\\ |\\ [\bot, \top]\\ |\\ [\bot, \bot]\end{array}$$

Note that $[\top, \bot]$ is *not* an element of $\mathbf{2}^{\mathbf{2}}$, because it isn't monotone.

Exercise 3.31. For each natural number $k \in \mathbb{N}$, define the function $s_k : \mathbb{N}_\perp \to \mathbb{N}_\perp$ by

$$s_k(\perp) \coloneqq \perp \quad \text{and} \quad s_k(n) \coloneqq \begin{cases} n+1 & \text{if } n \leq k, \\ \perp & \text{else.} \end{cases}$$

(i) Show that each s_k is ω-continuous.
(ii) Show that $\{s_k \mid k \in \mathbb{N}\}$ is an ω-chain in $\mathbb{N}_\perp^{\mathbb{N}_\perp}$ and its least upper bound is $s : \mathbb{N}_\perp \to \mathbb{N}_\perp$ with $s(\perp) \coloneqq \perp$ and $s(n) \coloneqq n+1$.

Thus, the functions $(s_k)_{k\in\mathbb{N}}$ are increasingly better approximations of the successor map on $\mathbb{N}_\perp$.

Lemma 3.32. *Given two ω-cpos A and B, the* evaluation *function*

$$\begin{aligned} B^A \times A &\to B \\ (f, x) &\mapsto f(x) \end{aligned}$$

is ω-continuous.

Exercise 3.33. Prove Lemma 3.32.
Hint: Use Exercise 3.27.

Exercise 3.34. Show that if $f : C \times A \to B$ is an ω-continuous function between ω-cpos, then so is the *curried* function

$$\begin{aligned} \hat{f} &: C \to B^A \\ \hat{f}(c) &\coloneqq a \mapsto f(c, a). \end{aligned}$$

For those familiar with category theory: conclude that the category ω-CPO is cartesian closed.

The following theorem says that the construction of the Kleene least fixed point (Theorem 3.17) is continuous and will be used to interpret PCF's fixed point operator $\texttt{fix}_\sigma : (\sigma \Rightarrow \sigma) \Rightarrow \sigma$.

Theorem 3.35. *For every ω-cppo A, the assignment*

$$\begin{aligned} \mu : A^A &\to A \\ f &\mapsto \textstyle\bigsqcup_{n\in\mathbb{N}} f^n(\perp) \end{aligned}$$

of an ω-continuous endofunction on A to its least fixed point is ω-continuous.

Proof sketch. For each natural number $n \in \mathbb{N}$, define

$$\begin{aligned} \mathrm{iter}_n : A^A &\to A \\ f &\mapsto f^n(\perp). \end{aligned}$$

By induction on n, we see that each iter_n is ω-continuous. Thus, they are elements of the exponential $A^{(A^A)}$. Moreover, $\{\mathrm{iter}_n \mid n \in \mathbb{N}\}$ is an ω-chain in this ω-cpo. Hence, we can take its least upper bound $F \in A^{(A^A)}$ which is an ω-continuous function by construction. Finally, from the construction of least upper bounds in the exponential, we calculate that $F(f) = \mu(f)$ for every $f \in A^A$, so that μ must be ω-continuous. □

3.5 List of exercises

1. Exercise 3.4: On the uniqueness of least upper bounds.
2. Exercise 3.6: On the uniqueness of least elements.
3. Exercise 3.9: On ω-chains in $\mathbb{N}_\perp$.
4. Exercise 3.11: On directed subsets and ω-chains.
5. Exercise 3.14: On the ω-cppo of finite and infinite binary sequences.
6. Exercise 3.15: On deriving monotonicity from preservation of least upper bounds of ω-chains.
7. Exercise 3.18: On showing that the Knaster–Tarski fixed point is the least.
8. Exercise 3.19: On the category of ω-cpos.
9. Exercise 3.20: On adjunctions between the categories of sets and (pointed) ω-cpos.
10. Exercise 3.27: On checking ω-continuity in each argument of the product.
11. Exercise 3.28: On ω-continuity of the induced map to the product.
12. Exercise 3.31: On approximating the successor map on $\mathbb{N}_\perp$.
13. Exercise 3.33: On ω-continuity of the evaluation map.
14. Exercise 3.34: On ω-continuity of the curried map.

Chapter 4

The Scott model of PCF

We started these notes by introducing PCF and its operational semantics in Chapter 2. We then developed sufficient domain theory in Chapter 3 to now give a denotational semantics of PCF, known as the Scott model of PCF, where we interpret the types of PCF as ω-cppos and terms as ω-continuous maps.

Definition 4.1 (Interpretation of PCF types, $[\![\sigma]\!]$)**.** We inductively assign an ω-cppo $[\![\sigma]\!]$ to each type σ of PCF:

$$[\![\mathbf{nat}]\!] \coloneqq \mathbb{N}_\perp \quad \text{and} \quad [\![\sigma \Rightarrow \tau]\!] \coloneqq [\![\tau]\!]^{[\![\sigma]\!]},$$

where we recall $\mathbb{N}_\perp$ from Example 3.8 and the exponential from Definition 3.29.

This interpretation extends to contexts by using iterated binary products:

Definition 4.2 (Interpretation of contexts, $[\![\Gamma]\!]$)**.** The interpretation of a context $\Gamma = [\mathrm{x}_0 : \sigma_0, \ldots, \mathrm{x}_{n-1} : \sigma_{n-1}]$ is

$$[\![\Gamma]\!] \coloneqq [\![\sigma_0]\!] \times \cdots \times [\![\sigma_{n-1}]\!]$$

with the convention that the empty context is interpreted as the empty product: the one point ω-cppo $\mathbf{1}$ from Example 3.22.

The interpretation of the terms of PCF proceeds by yet another inductive definition:

Definition 4.3 (Interpretation of PCF terms, $[\![M]\!]$)**.** A term $\Gamma \vdash M : \sigma$ of type σ in context Γ will be interpreted as an *ω-continuous* function

$$[\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![\sigma]\!]$$

according to the following clauses which mirror Definition 2.3:

(i) The interpretation of

$$\frac{}{\Gamma, \mathrm{x} : \sigma, \Delta \vdash \mathrm{x} : \sigma}$$

is

$$[\![\Gamma]\!] \times [\![\sigma]\!] \times [\![\Delta]\!] \xrightarrow{\pi} [\![\sigma]\!],$$

where π is a suitable composition of projections (recall Lemma 3.26), which is ω-continuous, because ω-continuous functions are closed under composition (recall Exercise 3.19).

(ii) To interpret

$$\frac{\Gamma, \mathrm{x} : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda\, \mathrm{x} : \sigma \,.\, M) : \sigma \Rightarrow \tau}$$

we first remark that we have

$$[\![\Gamma]\!] \times [\![\sigma]\!] \xrightarrow{[\![M]\!]} [\![\tau]\!]$$

by induction hypothesis, so that we can define

$$[\![\Gamma]\!] \xrightarrow{[\![\lambda\, \mathrm{x}:\sigma.M]\!]} [\![\tau]\!]^{[\![\sigma]\!]}$$

as the curried (recall Exercise 3.34) version of $[\![M]\!]$, i.e. for $\gamma \in [\![\Gamma]\!]$, we have $[\![\lambda\, \mathrm{x} : \sigma \,.\, M]\!](\gamma) \coloneqq x \mapsto [\![M]\!](\gamma, x)$.

(iii) To interpret

$$\frac{\Gamma \vdash M : \sigma \Rightarrow \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}$$

we first remark that we have

$$[\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![\tau]\!]^{[\![\sigma]\!]} \quad \text{and} \quad [\![\Gamma]\!] \xrightarrow{[\![N]\!]} [\![\sigma]\!]$$

by induction hypothesis, so that we can define

$$[\![\Gamma]\!] \xrightarrow{[\![M\,N]\!]} [\![\tau]\!]$$

by application: for $\gamma \in [\![\Gamma]\!]$, we have $[\![M\,N]\!](\gamma) \coloneqq [\![M]\!](\gamma)\big([\![N]\!](\gamma)\big)$.
More abstractly, it is the composition of

$$[\![\Gamma]\!] \xrightarrow{\langle [\![M]\!], [\![N]\!] \rangle} [\![\tau]\!]^{[\![\sigma]\!]} \times [\![\sigma]\!] \xrightarrow{\text{evaluation}} [\![\tau]\!],$$

where we recall Exercise 3.28 and Lemma 3.32.

(iv) To interpret

$$\frac{\Gamma \vdash M : \sigma \Rightarrow \sigma}{\Gamma \vdash \mathtt{fix}_\sigma(M) : \sigma}$$

we first remark that we have

$$[\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![\sigma]\!]^{[\![\sigma]\!]}$$

by induction hypothesis, so that we can define

$$[\![\Gamma]\!] \xrightarrow{[\![\mathtt{fix}_\sigma\, M]\!]} [\![\sigma]\!]$$

as the composition $[\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![\sigma]\!]^{[\![\sigma]\!]} \xrightarrow{\mu} [\![\sigma]\!]$, where we recall μ, which assigns least fixed points, from Theorem 3.35.

(v) The interpretation of

$$\frac{}{\Gamma \vdash \texttt{zero} : \textbf{nat}}$$

is

$$[\![\Gamma]\!] \xrightarrow{[\![\texttt{zero}]\!]} \mathbb{N}_\perp$$

which is given by $\gamma \in [\![\Gamma]\!] \mapsto 0 \in \mathbb{N}_\perp$.

(vi) To interpret

$$\frac{\Gamma \vdash M : \textbf{nat}}{\Gamma \vdash \texttt{succ}(M) : \textbf{nat}}$$

we first remark that we have

$$[\![\Gamma]\!] \xrightarrow{[\![M]\!]} \mathbb{N}_\perp$$

by induction hypothesis, so that we can define

$$[\![\Gamma]\!] \xrightarrow{[\![\texttt{succ}\ M]\!]} \mathbb{N}_\perp$$

as the composition $[\![\Gamma]\!] \xrightarrow{[\![M]\!]} \mathbb{N}_\perp \xrightarrow{s} \mathbb{N}_\perp$, where $s : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is the successor function on $\mathbb{N}_\perp$, i.e. $s(\perp) \coloneqq \perp$ and $s(n) \coloneqq n + 1$.

(vii) To interpret

$$\frac{\Gamma \vdash M : \textbf{nat}}{\Gamma \vdash \texttt{pred}(M) : \textbf{nat}}$$

we first remark that we have

$$[\![\Gamma]\!] \xrightarrow{[\![M]\!]} \mathbb{N}_\perp$$

by induction hypothesis, so that we can define

$$[\![\Gamma]\!] \xrightarrow{[\![\texttt{pred}\ M]\!]} \mathbb{N}_\perp$$

as the composition $[\![\Gamma]\!] \xrightarrow{[\![M]\!]} \mathbb{N}_\perp \xrightarrow{p} \mathbb{N}_\perp$, where $p : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is the predecessor function on $\mathbb{N}_\perp$, i.e. $p(\perp) \coloneqq \perp$, $p(0) \coloneqq 0$ and $p(n + 1) \coloneqq n$.

(viii) Finally, to interpret

$$\frac{\Gamma \vdash M : \textbf{nat} \qquad \Gamma \vdash N_1 : \textbf{nat} \qquad \Gamma \vdash N_2 : \textbf{nat}}{\Gamma \vdash \texttt{ifzero}(M, N_1, N_2) : \textbf{nat}}$$

we first remark that we have

$$[\![\Gamma]\!] \xrightarrow{[\![M]\!]} \mathbb{N}_\perp,\ [\![\Gamma]\!] \xrightarrow{[\![N_1]\!]} \mathbb{N}_\perp \text{ and } [\![\Gamma]\!] \xrightarrow{[\![N_2]\!]} \mathbb{N}_\perp$$

by induction hypothesis, so that we can define

$$[\![\Gamma]\!] \xrightarrow{[\![\texttt{ifzero}(M,N_1,N_2)]\!]} \mathbb{N}_\perp$$

as the composition $[\![\Gamma]\!] \xrightarrow{\langle [\![M]\!], [\![N_1]\!], [\![N_2]\!] \rangle} \mathbb{N}_\perp \times \mathbb{N}_\perp \times \mathbb{N}_\perp \xrightarrow{c} \mathbb{N}_\perp$, where

$$c : \mathbb{N}_\perp \times \mathbb{N}_\perp \times \mathbb{N}_\perp \to \mathbb{N}_\perp$$

is defined as the ω-continuous function $c(\perp, y, z) \coloneqq \perp$, $c(0, y, z) \coloneqq y$ and $c(n + 1, y, z) \coloneqq z$.

Remark 4.4 (Programs of type σ are interpreted as elements of $[\![\sigma]\!]$). Note that a program, i.e. a closed term, $\vdash M : \sigma$ is interpreted as a function $[\![M]\!] : \mathbf{1} \longrightarrow [\![\sigma]\!]$. Since the underlying set of $\mathbf{1}$ is the singleton $\{\star\}$, this function simply picks out an element. With this in mind, we see that programs of type σ are interpreted as elements of $[\![\sigma]\!]$.

Exercise 4.5. Show that the numerals of PCF are interpreted as the natural numbers in $\mathbb{N}_\perp$, i.e. $[\![\underline{n}]\!] = n \in \mathbb{N}_\perp$ for every natural number $n \in \mathbb{N}$.

4.1 Towards soundness

The next section will be devoted to proving the *soundness* theorem, which says that if $M \Downarrow N$, then $[\![M]\!] = [\![N]\!]$. In other words, the interpretation respects the operational semantics. The proof of soundness relies on the following lemma:

Lemma 4.6 (β-equality). *The Scott model of PCF validates β-equality. That is, if $\Gamma, \mathrm{x} : \sigma \vdash M : \tau$ and $\Gamma \vdash N : \sigma$, then*

$$[\![(\lambda\, \mathrm{x} : \sigma \,.\, M)\, N]\!] = [\![M[N/\mathrm{x}]]\!]$$

Exercise 4.7. Prove Lemma 4.6 from the substitution lemma given below.

Suppose that we have a term $M : \tau$ in context $\Gamma = [\mathrm{x}_0 : \sigma_0, \dots, \mathrm{x}_{k-1} : \sigma_{k-1}]$, and assume that we have another context Δ with terms $\Delta \vdash N_i : \sigma_i$ for every $0 \leq i \leq k-1$. This yields (by a recursive definition on the structure of derivations of $\Gamma \vdash M : \tau$) a term

$$\Delta \vdash M[N_0/\mathrm{x}_0, \dots, N_{k-1}/\mathrm{x}_{k-1}] : \tau$$

in context Δ.

Thus, in our interpretation, we get

$$[\![M[N_0/\mathrm{x}_0, \dots, N_{k-1}/\mathrm{x}_{k-1}]]\!](\delta) \in [\![\tau]\!] \tag{†}$$

for every $\delta \in [\![\Delta]\!]$.

Alternatively, we can note that $[\![N_i]\!](\delta) \in [\![\sigma_i]\!]$ for every $0 \leq i \leq k-1$, so that we can feed them as inputs to $[\![M]\!] : [\![\sigma_0]\!] \times \cdots \times [\![\sigma_{k-1}]\!] \longrightarrow [\![\tau]\!]$ which yields:

$$[\![M]\!]\big([\![N_0]\!](\delta), \dots, [\![N_{k-1}]\!](\delta)\big) \in [\![\tau]\!]. \tag{‡}$$

The substitution lemma says that (†) and (‡) agree.

Lemma 4.8 (Substitution lemma). *If $[\mathrm{x}_0 : \sigma_0, \dots, \mathrm{x}_{k-1} : \sigma_{k-1}] = \Gamma \vdash M : \tau$, then for every context Δ and terms $\Delta \vdash N_i : \sigma_i$ with $0 \leq i \leq k-1$, we have*

$$[\![M[N_0/\mathrm{x}_0, \dots, N_{k-1}/\mathrm{x}_{k-1}]]\!](\delta) = [\![M]\!]\big([\![N_0]\!](\delta), \dots, [\![N_{k-1}]\!](\delta)\big)$$

for every $\delta \in [\![\Delta]\!]$.

Proof. By induction on the structure of derivations of $\Gamma \vdash M : \tau$.
For example, for the case of λ-abstraction, we consider the term $\Gamma, y : \tau \vdash M : \rho$ with $[\mathrm{x}_0 : \sigma_0, \ldots, \mathrm{x}_{k-1} : \sigma_{k-1}] = \Gamma$, and we assume to have a context Δ with terms $\Delta \vdash N_i : \sigma_i$ for $0 \leq i \leq k-1$. We have to prove that

$$[\![(\lambda\, \mathrm{y} : \tau \,.\, M)[N_0/\mathrm{x}_0, \ldots, N_{k-1}/\mathrm{x}_{k-1}]]\!](\delta) = [\![\lambda\, \mathrm{y} : \tau \,.\, M]\!]\big([\![N_0]\!](\delta), \ldots, [\![N_{k-1}]\!](\delta)\big)$$

for every $\delta \in [\![\Delta]\!]$.
Note that this is an equality of elements of $[\![\rho]\!]^{[\![\tau]\!]}$, i.e. an equality of ($\omega$-continuous) functions, so let $t \in [\![\tau]\!]$ be arbitrary and note that

$$\begin{aligned}
&([\![(\lambda\, \mathrm{y} : \tau \,.\, M)[N_0/\mathrm{x}_0, \ldots, N_{k-1}/\mathrm{x}_{k-1}]]\!](\delta))(t) \\
&= [\![M[N_0/\mathrm{x}_0, \ldots, N_{k-1}/\mathrm{x}_{k-1}, \mathrm{y}/\mathrm{y}]]\!](\delta, t) \\
&= [\![M]\!]\big([\![N_0]\!](\delta, t), \ldots, [\![N_{k-1}]\!](\delta, t), [\![\mathrm{y}]\!](\delta, t)\big) && \text{(by IH applied to } \Gamma, \mathrm{y} : \tau \vdash M : \rho) \\
&= [\![M]\!]\big([\![N_0]\!](\delta, t), \ldots, [\![N_{k-1}]\!](\delta, t), t\big) \\
&= \big([\![\lambda\, \mathrm{y} : \tau \,.\, M]\!]\big([\![N_0]\!](\delta), \ldots, [\![N_{k-1}]\!](\delta)\big)\big)(t),
\end{aligned}$$

as desired. Note that the N_i in lines 2–4 refer to the terms N_i in the extended context $\Delta, \mathrm{y} : \tau$. □

4.2 Soundness

The soundness theorem expresses that the denotational semantics, in the form of the Scott model of PCF, respects the operational semantics.

Theorem 4.9 (Soundness)**.** *For terms M and N of the same type in the same context, we have: if $M \Downarrow N$, then $[\![M]\!] = [\![N]\!]$.*

Proof. By induction on the structure of the derivations of $M \Downarrow N$. We work out two cases: the fixed point operator and application.

- Recall that the former is the rule

$$\frac{M(\mathtt{fix}_\sigma\ M) \Downarrow V}{\mathtt{fix}_\sigma\ M \Downarrow V}$$

so that we may assume $[\![M(\mathtt{fix}_\sigma\ M)]\!] = [\![V]\!]$ and we have to prove:

$$[\![\mathtt{fix}_\sigma\ M]\!] = [\![V]\!].$$

But this holds: if M is a term in context Γ, then for every $\gamma \in [\![\Gamma]\!]$, we have

$$\begin{aligned}
[\![\mathtt{fix}_\sigma\ M]\!](\gamma) &= \mu\big([\![M]\!](\gamma)\big) && \text{(by definition)} \\
&= \big([\![M]\!](\gamma)\big)\big(\mu\big([\![M]\!](\gamma)\big)\big) && \text{(because } \mu \text{ gives a fixed point)} \\
&= [\![M(\mathtt{fix}_\sigma\ M)]\!](\gamma) && \text{(by definition)} \\
&= [\![V]\!](\gamma) && \text{(by assumption).}
\end{aligned}$$

- The big-step rule for application is

$$\frac{M \Downarrow \lambda\, \mathrm{x} : \sigma \, .\, E \qquad E[N/x] \Downarrow V}{M\, N \Downarrow V}$$

so that we may assume $[\![M]\!] = [\![\lambda\, \mathrm{x} : \sigma \, .\, E]\!]$ and $[\![E[N/x]]\!] = [\![V]\!]$, and we have to prove:

$$[\![M\, N]\!] = [\![V]\!].$$

If M and N are terms in a context Γ, then for every $\gamma \in [\![\Gamma]\!]$, we calculate:

$$\begin{aligned}
[\![M\, N]\!](\gamma) &= [\![M]\!](\gamma)\big([\![N]\!](\gamma)\big) && \text{(by definition)} \\
&= [\![\lambda\, \mathrm{x} : \sigma \, .\, E]\!](\gamma)\big([\![N]\!](\gamma)\big) && \text{(by assumption on } [\![M]\!]) \\
&= [\![(\lambda\, \mathrm{x} : \sigma \, .\, E)N]\!](\gamma) && \text{(by definition)} \\
&= [\![E[N/\mathrm{x}]]\!](\gamma) && \text{(by Lemma 4.6)} \\
&= [\![V]\!](\gamma) && \text{(by assumption on } [\![N]\!]),
\end{aligned}$$

completing the proof. □

Exercise 4.10. Consider the program $M :\equiv \mathtt{fix}_{\mathbf{nat}}(\lambda\, \mathrm{x} : \mathbf{nat} \, .\, \mathrm{x})$.
Prove that $[\![M]\!] = \bot$ and use the soundness theorem to conclude that M does not reduce to a value.
Hint: For a particularly quick proof that $[\![M]\!] = \bot$, use Exercise 3.18.

4.3 List of exercises

1. Exercise 4.5: On the interpretation of the numerals.
2. Exercise 4.7: On proving that the interpretation validates β-equality.
3. Exercise 4.10: On using the soundness theorem to prove non-termination.

Chapter 5

Computational adequacy

Soundness is a nice and fundamental result, but a converse would be more interesting: Can we use the model to compute in PCF? More precisely, if two terms M and N are equal in the model, does M reduce to N (or vice versa) in the operational semantics?

This is actually not the case, because the model is more extensional than PCF, e.g. the programs $\lambda\, \mathtt{x} : \mathbf{nat}\ .\ \mathtt{x}$ and $\lambda\, \mathtt{x} : \mathbf{nat}\ .\ \mathtt{pred}(\mathtt{succ}\ \mathtt{x})$ are different values in PCF, but their interpretations as ω-continuous functions are equal. Even if we did allow for reductions in the bodies of the λ-abstractions, we could easily get another counterexample such as the pair of programs $\lambda\, \mathtt{x} : \mathbf{nat}\ .\ \lambda\, \mathtt{y} : \mathbf{nat}\ .\ \mathtt{add}\, x\, y$ and $\lambda\, \mathtt{x} : \mathbf{nat}\ .\ \lambda\, \mathtt{y} : \mathbf{nat}\ .\ \mathtt{add}\, y\, x$ where $\mathtt{add}$ is a program for addition.

However, it *is* the case that if M is a program of type **nat** and $[\![M]\!] = n$, then $M \Downarrow \underline{n}$. This is known as the *computational adequacy* of the Scott model of PCF and allows us to compute in PCF using the domain-theoretic denotational semantics.

Theorem (Computational adequacy)**.** *For every program M of type* **nat** *and natural number $n \in \mathbb{N}$, if $[\![M]\!] = n$, then $M \Downarrow \underline{n}$.*

Unlike soundness, computational adequacy cannot be proved by a straightforward induction on types, or structure of the derivation of the program M, since the statement refers to closed terms of type **nat** only.

Therefore, instead of proving computational adequacy directly, we will derive it as a corollary of a more general result that does allow for a proof by induction on types.

More specifically, we will introduce a *logical relation* [Plo73]. It is an example of a fundamental and often used technique in the theory of programming languages, going back to *Tait's method of computability* [Tai67] and also known as the method of *reducibility candidates* [GLT89].

5.1 The logical relation

We introduce the logical relation R_σ, a binary relation between semantics and syntax of PCF, and use it to prove computational adequacy.

Definition 5.1 (Logical relation, R_σ). We define a binary relation R_σ between elements of $[\![\sigma]\!]$ and programs of type σ by induction on types:

(i) $x\ R_{\mathbf{nat}}\ M$ holds if $x = n$ with $n \in \mathbb{N}$ implies $M \Downarrow \underline{n}$,

(ii) $f\ R_{\sigma \Rightarrow \tau}\ M$ holds if $x\ R_\sigma\ N$ implies $f(x)\ R_\tau\ M\,N$ for every element $x \in [\![\sigma]\!]$ and program N of type σ.

Observe that computational adequacy is equivalent to the statement that for every program $M : \mathbf{nat}$ we have $[\![M]\!]\ R_{\mathbf{nat}}\ M$. Hence, we will have proven computational adequacy if we can show:

Lemma. *For every program $M : \sigma$, it holds that $[\![M]\!]\ R_\sigma\ M$.*

This, in turn, will follow from the fundamental theorem of the logical relation:

Lemma (Fundamental theorem of the logical relation). *If $[\mathrm{x}_0 : \sigma_0, \dots, \mathrm{x}_{k-1} : \sigma_{k-1}] = \Gamma \vdash M : \tau$, then whenever we have programs $N_i : \sigma_i$ and $s_i \in [\![\sigma_i]\!]$ such that $s_i\ R_{\sigma_i}\ N_i$ for all $0 \leq i \leq k-1$, it holds that*

$$\big([\![M]\!](s_0, \dots, s_{k-1})\big)\ R_\tau\ M[N_0/\mathrm{x}_0, \dots, N_{k-1}/\mathrm{x}_{k-1}].$$

The second clause, item (ii), of Definition 5.1 says that related inputs must go to related outputs and is designed to make proofs by induction on types possible.

Before we can prove the fundamental theorem of the logical relation, we need to establish several properties of the relation R_σ.

Lemma 5.2. *If $x \sqsubseteq y$ and $y\ R_\sigma\ M$, then $x\ R_\sigma\ M$.*

Proof. We proceed by induction on the type σ. For the base case, suppose we have $x \sqsubseteq y$ in $\mathbb{N}_\perp$ and $y\ R_{\mathbf{nat}}\ M$, and assume that $x = n$ for a natural number n. We have to prove that $M \Downarrow \underline{n}$. Since $n = x \sqsubseteq y$, we must have $y = n$ by definition of the partial order on $\mathbb{N}_\perp$, and hence $M \Downarrow \underline{n}$ because we assumed $y\ R_{\mathbf{nat}}\ M$.

For function types, we assume to have $f \sqsubseteq g$ in $[\![\tau]\!]^{[\![\sigma]\!]}$ with $g\ R_{\sigma\Rightarrow\tau}\ M$, and we have to prove $f\ R_{\sigma\Rightarrow\tau}\ M$. So suppose that we have $x\ R_\sigma\ N$. Then $g(x)\ R_\tau\ M\,N$ because we assumed $g\ R_{\sigma\Rightarrow\tau}\ M$. But $f \sqsubseteq g$, so $f(x) \sqsubseteq g(x)$ and hence, we get the desired $f(x)\ R_\tau\ M\,N$ by the induction hypothesis applied at the type τ. □

Lemma 5.3. *The least element is related to all programs. That is, for every program $M : \sigma$, we have $\perp\ R_\sigma\ M$, where we recall that $\perp$ denotes the least element of $[\![\sigma]\!]$.*

Lemma 5.4. *The logical relation is closed under least upper bounds of ω-chains. That is, for every program $M : \sigma$ and ω-chain $x_0 \sqsubseteq x_1 \dots$ in $[\![\sigma]\!]$, if $x_n \; R_\sigma \; M$ for every $n \in \mathbb{N}$, then $(\bigsqcup_{n \in \mathbb{N}} x_n) \; R_\sigma \; M$.*

Exercise 5.5. Prove Lemmas 5.3 and 5.4 by induction on PCF types.

Exercise 5.6. Check the following closure properties of $R_{\mathbf{nat}}$:

(i) $0 \; R_{\mathbf{nat}} \; \underline{0}$;

(ii) if $x \; R_{\mathbf{nat}} \; M$, then $s(x) \; R_{\mathbf{nat}} \; \mathtt{succ} \; M$, where s is the successor map on $\mathbb{N}_\bot$ as in Definition 4.3(vi);

(iii) if $x \; R_{\mathbf{nat}} \; M$, then $p(x) \; R_{\mathbf{nat}} \; \mathtt{pred} \; M$, where p is the predecessor map on $\mathbb{N}_\bot$ as in Definition 4.3(vii);

(iv) if $x \; R_{\mathbf{nat}} \; M$, $y \; R_{\mathbf{nat}} \; N_1$ and $z \; R_{\mathbf{nat}} \; N_2$, then $c(x,y,z) \; R_{\mathbf{nat}} \; \mathtt{ifzero}(M, N_1, N_2)$, where c is the if-zero map on $\mathbb{N}_\bot$ as in Definition 4.3(viii).

5.2 Applicative approximation

We will need one more ingredient for proving the fundamental theorem of the logical relation and (hence) computational adequacy: the *applicative approximation* relation.

In Lemma 5.2, we showed that if $x \sqsubseteq y$ and $y \; R_\sigma \; M$, then $x \; R_\sigma \; M$. The applicative approximation relation may be motivated by the desire to have a similar property of the logical relation, but for programs, rather than elements of the interpretation. That is, we introduce a binary relation $\precsim_\sigma$ on programs of type σ and show (Lemma 5.9) that if $x \; R_\sigma \; M$ and $M \precsim_\sigma N$, then $x \; R_\sigma \; N$.

Definition 5.7 (Applicative approximation)**.** We define a binary relation $\precsim_\sigma$ on programs of type σ by induction on types:

(i) $M \precsim_{\mathbf{nat}} N$ holds if $M \Downarrow \underline{n}$ implies $N \Downarrow \underline{n}$ for every natural number $n \in \mathbb{N}$.

(ii) $M \precsim_{\sigma \Rightarrow \tau} N$ holds if for every program K of type σ we have $M\,K \precsim_\tau N\,K$.

Note that the applicative approximation relation mirrors the partial orders on $\mathbb{N}_\bot$ and the exponentials. But the applicative approximation relation is only a *preorder*; it is reflexive and transitive, but not antisymmetric:

Exercise 5.8. Give examples of programs M and N such that $M \precsim_{\mathbf{nat} \Rightarrow \mathbf{nat}} N$ and $N \precsim_{\mathbf{nat} \Rightarrow \mathbf{nat}} M$, but $M \neq N$.

Lemma 5.9. *If $x \; R_\sigma \; M$ and $M \precsim_\sigma N$, then $x \; R_\sigma \; N$.*

Proof. We proceed by induction on types. For the base type, we assume $x \; R_{\mathbf{nat}} \; M$ and $M \precsim_{\mathbf{nat}} N$. Suppose that $x = n$ for $n \in \mathbb{N}$; we have to prove $N \Downarrow \underline{n}$. But $M \Downarrow \underline{n}$ since $x \; R_{\mathbf{nat}} \; M$ and hence, $N \Downarrow \underline{n}$ because $M \precsim_{\mathbf{nat}} N$.
For function types, we assume $f \; R_{\sigma \Rightarrow \tau} \; M$ and $M \precsim_{\sigma \Rightarrow \tau} N$. Suppose that we have $x \; R_\sigma \; K$; we have to prove $f(x) \; R_\tau \; N\,K$. By our assumption that $f \; R_{\sigma \Rightarrow \tau} \; M$, we get

$f(x)\ R_\tau\ M\,K$. Moreover, our assumption $M \precsim_{\sigma\Rightarrow\tau} N$ yields $M\,K \precsim_\tau N\,K$, so that $f(x)\ R_\tau\ N\,K$ by the induction hypothesis applied at τ, as desired. ☐

The following lemma has two useful corollaries for proving the fundamental theorem of the logical relation:

Lemma 5.10. *For all programs M and N of type σ, if $M \Downarrow V$ implies $N \Downarrow V$ for every program $V : \sigma$, then $M \precsim_\sigma N$.*

Exercise 5.11. Prove Lemma 5.10 by induction on types.

Corollary 5.12. *For every program $M : \sigma \Rightarrow \sigma$, it holds that $M(\mathtt{fix}_\sigma\ M) \precsim_\sigma \mathtt{fix}_\sigma\,M$.*

Proof. By Exercise 5.11 it is enough to establish that $M(\mathtt{fix}_\sigma\ M) \Downarrow V$ implies $\mathtt{fix}_\sigma\ M \Downarrow V$ for every program V. But this holds by definition of the big-step operational semantics. ☐

Corollary 5.13. *For every term $\mathrm{x} : \sigma \vdash M : \tau$ and program $N : \sigma$, it holds that $M[N/\mathrm{x}] \precsim_\tau (\lambda\,\mathrm{x} : \sigma\ .\ M)N$.*

Proof. By Lemma 5.10 and the definition of the big-step operational semantics. ☐

Corollary 5.12 allows us to prove that the logical relation is closed under the least fixed point operation μ (recall Theorem 3.17):

Lemma 5.14. *If $f\ R_{\sigma\Rightarrow\sigma}\ M$, then $\mu(f)\ R_\sigma\ \mathtt{fix}_\sigma\ M$.*

Proof. Suppose that $f\ R_{\sigma\Rightarrow\sigma}\ M$. Since $\mu(f) = \bigsqcup_{n\in\mathbb{N}} f^n(\bot)$, Lemma 5.4 tells us that it is enough to prove that $f^n(\bot)\ R_\sigma\ (\mathtt{fix}_\sigma\ M)$ for every $n \in \mathbb{N}$. We do so by induction on n. The case $n = 0$ is provided by Lemma 5.3. For the inductive case, suppose that $f^n(\bot)\ R_\sigma\ (\mathtt{fix}_\sigma\ M)$; we show that $f^{n+1}(\bot)\ R_\sigma\ (\mathtt{fix}_\sigma\ M)$. By induction hypothesis and the assumption that $f\ R_{\sigma\Rightarrow\sigma}\ M$, we see that $f(f^n(\bot))\ R_\sigma\ M\,(\mathtt{fix}_\sigma\ M)$. Hence, we obtain the desired $f^{n+1}(\bot)\ R_\sigma\ (\mathtt{fix}_\sigma\ M)$ by Lemma 5.9 and Corollary 5.12. ☐

5.3 Proving computational adequacy

Finally, we have established enough properties to prove:

Lemma 5.15 (Fundamental theorem of the logical relation). *If $M : \tau$ is a term in a context $\Gamma = [\mathrm{x}_0 : \sigma_0, \ldots, \mathrm{x}_{k-1} : \sigma_{k-1}]$, then whenever we have programs $N_i : \sigma_i$ and $s_i \in [\![\sigma_i]\!]$ such that $s_i\ R_{\sigma_i}\ N_i$ for all $0 \leq i \leq k-1$, it holds that*

$$\big([\![M]\!](s_0, \ldots, s_{k-1})\big)\ R_\tau\ M[N_0/\mathrm{x}_0, \ldots, N_{k-1}/\mathrm{x}_{k-1}].$$

Proof. We abbreviate $s_0, \ldots, s_{k-1}$ as $\vec{s}$ and $M[N_0/\mathrm{x}_0, \ldots, N_{k-1}/\mathrm{x}_{k-1}]$ as $M[\vec{N}/\vec{\mathrm{x}}]$. We proceed by induction on the structure of the derivations $\Gamma \vdash M : \tau$:

- For $[\mathrm{x}_0 : \sigma_0, \ldots, \mathrm{x}_{k-1} : \sigma_{k-1}] \vdash \mathrm{x}_i : \sigma_i$ and $s_i\ R_{\sigma_i}\ N_i$ for all $0 \leq i \leq k-1$, we observe that
$$[\![\mathrm{x}_i]\!](\vec{s}) = s_i\ R_{\sigma_i}\ N_i = \mathrm{x}_i[\vec{N}/\vec{\mathrm{x}}],$$
as wished.
- For application, we consider $\Gamma \vdash M : \tau \Rightarrow \rho$ and $\Gamma \vdash K : \tau$ and we have to prove
$$[\![M\,K]\!](\vec{s})\ R_\rho\ (M\,K)[\vec{N}/\vec{\mathrm{x}}].$$
Our induction hypothesis yields
$$[\![M]\!](\vec{s})\ R_{\tau\Rightarrow\rho}\ M[\vec{N}/\vec{\mathrm{x}}] \quad \text{and} \quad [\![K]\!](\vec{s})\ R_\tau\ K[\vec{N}/\vec{\mathrm{x}}].$$
And hence, by definition of $R_{\tau\Rightarrow\rho}$,
$$[\![M]\!](\vec{s})\big([\![K]\!](\vec{s})\big)\ R_\rho\ M[\vec{N}/\vec{\mathrm{x}}]\,K[\vec{N}/\vec{\mathrm{x}}],$$
but this is an unfolded version of what we wanted to prove.
- For λ-abstraction, we consider $\Gamma, \mathrm{y} : \tau \vdash M : \rho$ and we have to prove
$$[\![\lambda\,\mathrm{y} : \tau\,.\,M]\!](\vec{s})\ R_{\tau\Rightarrow\rho}\ (\lambda\,\mathrm{y} : \tau\,.\,M)[\vec{N}/\vec{\mathrm{x}}].$$
So suppose that we have $t\ R_\tau\ K$; we show that
$$\big([\![\lambda\,\mathrm{y} : \tau\,.\,M]\!](\vec{s})\big)(t)\ R_\rho\ \big((\lambda\,\mathrm{y} : \tau\,.\,M)[\vec{N}/\vec{\mathrm{x}}]\big)K.$$
The left-hand side unfolds to $[\![M]\!](\vec{s}, t)$, while the right-hand side unfolds to $\big(\lambda\,\mathrm{y} : \tau\,.\,M[\vec{N}/\vec{\mathrm{x}}]\big)K$. By Lemma 5.9 and Corollary 5.13, it thus suffices to prove
$$[\![M]\!](\vec{s}, t)\ R_\rho\ M[\vec{N}/\vec{\mathrm{x}}, K/\mathrm{y}],$$
which holds by induction hypothesis.
- For `fix`, we consider $\Gamma \vdash M : \tau \Rightarrow \tau$ and we have to prove
$$[\![\mathtt{fix}_\tau\ M]\!](\vec{s})\ R_\tau\ (\mathtt{fix}_\tau\ M)[\vec{N}/\vec{\mathrm{x}}].$$
By definition, $[\![\mathtt{fix}_\tau\ M]\!](\vec{s}) = \mu\big([\![M]\!](\vec{s})\big)$ and $(\mathtt{fix}_\tau\ M)[\vec{N}/\vec{\mathrm{x}}] = \mathtt{fix}_\tau(M[\vec{N}/\vec{\mathrm{x}}])$, so by Lemma 5.14, it is enough to prove that $[\![M]\!](\vec{s})\ R_{\tau\Rightarrow\tau}\ M[\vec{N}/\vec{\mathrm{x}}]$. But this holds by induction hypothesis.
- Finally, the cases for `zero`, `succ`, `zero` and `ifzero` follow from Exercise 5.6. □

For the special case of the empty context, we obtain:

Corollary 5.16. *For every program $M : \sigma$, it holds that $[\![M]\!]\ R_\sigma\ M$.*

By further specialising to the base type and the definition of $R_{\mathbf{nat}}$, we get:

Theorem 5.17 (Computational adequacy). *For every program M of type* **nat** *and natural number $n \in \mathbb{N}$, if $[\![M]\!] = n$, then $M \Downarrow \underline{n}$.*

We end these notes with an exercise illustrating two uses of computational adequacy: one on establishing the computational behaviour of a program, and one on using a domain-theoretic argument to show that a certain program cannot exist.

Exercise 5.18. Consider the following pseudocode:

```
root : (Nat -> Nat) -> Nat
root M = root' M 0

root' : (Nat -> Nat) -> Nat -> Nat
root' M m = if   M 0 == 0
            then m
            else root' (shift M) (m + 1)

shift : (Nat -> Nat) -> Nat -> Nat
shift M n = M (n + 1)
```

(i) Translate the above pseudocode to corresponding programs `root`, `root'` and `shift` in PCF.

(ii) Use computational adequacy to prove that for every program $M : \mathbf{nat} \Rightarrow \mathbf{nat}$ and natural number n, we have

$$\texttt{root}\ M \Downarrow \underline{n} \iff M\,\underline{n} \Downarrow \underline{0} \text{ and } \forall_{k<n}\big(M\,\underline{k} \text{ reduces to a non-zero numeral}\big).$$

That is, `root` M computes the first "root" (or zero) of M provided that M terminates on all values up to (and including) the root.

(iii) Give an informal argument as to why we can't drop the condition that $M\,\underline{k}$ reduces to a (non-zero) numeral for all $k \leq n$.

(iv) Show that there is *no* program $F : (\mathbf{nat} \Rightarrow \mathbf{nat}) \Rightarrow \mathbf{nat}$ such that for every program $M : \mathbf{nat} \Rightarrow \mathbf{nat}$ and natural number n, we have

$$F\,M \Downarrow \underline{n} \iff n \text{ is the least natural number with } M\,\underline{n} \Downarrow \underline{0}.$$

Hint: Consider the ω-continuous functions $f, g : \mathbb{N}_\perp \to \mathbb{N}_\perp$ given by $f(\perp) \coloneqq \perp$, $f(0) \coloneqq \perp$ and $f(n+1) \coloneqq 0$, and $g(x) = 0$ for all $x \in \mathbb{N}_\perp$. Use the observation that $f \sqsubseteq g$ to derive a contradiction from the assumption that a program F with the above specification exists.

5.4 List of exercises

1. Exercise 5.5: On proving that the logical relation contains the least element is and is closed under least upper bounds of ω-chains.

2. Exercise 5.6: On showing that the logical relation is suitably closed under the basic operations on the type of natural numbers.
3. Exercise 5.8: On a counterexample to antisymmetry of the applicative approximation relation.
4. Exercise 5.11: On a sufficient big-step condition for the applicative approximation relation to hold.
5. Exercise 5.18: On computational adequacy and programs finding roots.

Bibliography

[Abr97] Samson Abramsky. “Game Semantics”.
In: *Semantics and Logics of Computation.* Ed. by A. Pitts and P. Dybjer.
Cambridge University Press, 1997, pp. 1–32.
DOI: 10.1017/CBO9780511526619.002 (cit. on p. 1).

[AJ94] Samson Abramsky and Achim Jung. “Domain theory”.
In: *Handbook of Logic in Computer Science.*
Ed. by S. Abramsky, Dov M. Gabray, and T. S. E. Maibaum. Vol. 3.
Clarendon Press, 1994, pp. 1–168. Updated online version available at
https://www.cs.bham.ac.uk/~axj/pub/papers/handy1.pdf
(cit. on pp. 1–2, 12).

[dJon22] Tom de Jong.
“Domain Theory in Constructive and Predicative Univalent Foundations”.
PhD thesis. School of Computer Science, University of Birmingham, 2022.
arXiv: 2301.12405 [cs.LO].
URL: https://tdejong.com/writings/phd-thesis.pdf (cit. on p. 2).

[Esc07a] Martín Escardó.
“Domain theory and denotational semantics of functional programming”.
Slides for an advanced course at the *Midlands Graduate School (MGS) in the Foundations of Computing Science.* 2007. URL: https://www.cs.nott.ac.uk/~psznhn/MGS2007/LectureNotes/mgs2007-dom.pdf (cit. on p. 1).

[Esc07b] Martín Escardó. *Seemingly impossible functional programs.*
Guest post on Andrej Bauer’s blog. 2007.
URL: https://math.andrej.com/2007/09/28/seemingly-impossible-functional-programs/ (cit. on p. 2).

[GHK+03] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains.* Vol. 93.
Encyclopedia of Mathematics and its Applications.
Cambridge University Press, 2003. DOI: 10.1017/CBO9780511542725
(cit. on pp. 1–2).

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. Vol. 7. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989. Reprinted with minor corections in 1990; updated web version available at https://www.paultaylor.eu/stable/prot.pdf (cit. on p. 25).

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992 (cit. on pp. 2–3).

[Har20] Brendan Hart. "Investigating Properties of PCF in Agda". Final year MSci project. School of Computer Science, University of Birmingham, 2020. URL: https://raw.githubusercontent.com/BrendanHart/Investigating-Properties-of-PCF/master/InvestigatingPropertiesOfPCFInAgda.pdf. Agda code available at https://github.com/BrendanHart/Investigating-Properties-of-PCF (cit. on p. 2).

[Hyl97] Martin Hyland. "Game Semantics". In: *Semantics and Logics of Computation.* Ed. by A. Pitts and P. Dybjer. Cambridge University Press, 1997, pp. 131–184. DOI: 10.1017/CBO9780511526619.005 (cit. on p. 1).

[LN15] John Longley and Dag Normann. *Higher-Order Computability.* Theory and Applications of Computability. Springer, 2015. DOI: 10.1007/978-3-662-47992-6 (cit. on pp. 1, 3).

[Lon95] John Longley. "Realizability Toposes and Language Semantics". PhD thesis. Department of Computer Science, University of Edinburgh, 1995. URL: https://www.lfcs.inf.ed.ac.uk/reports/95/ECS-LFCS-95-332 (cit. on p. 1).

[Mar+10] Simon Marlow et al. *Haskell 2010 Language Report.* 2010. URL: https://haskell.org/definition/haskell2010.pdf (cit. on p. 3).

[Plo73] G. D. Plotkin. *Lambda-definability and logical relations.* Memorandum SAI-RM-4. School of Artificial Intelligence, University of Edinburgh, 1973. URL: https://homepages.inf.ed.ac.uk/gdp/publications/logical_relations_1973.pdf (cit. on p. 25).

[Plo77] G. D. Plotkin. "LCF considered as a programming language". In: *Theoretical Computer Science* 5.3 (1977), pp. 223–255. DOI: 10.1016/0304-3975(77)90044-5 (cit. on pp. 1, 3).

[Plo83] Gordon Plotkin. *Domains.* Course notes known as the *Pisa notes.* 1983. URL: https://homepages.inf.ed.ac.uk/gdp/publications/Domains_a4.ps (cit. on pp. 2–3).

[PWF12] Andrew M. Pitts, Glynn Winskel, and Marcelo Fiore. *Denotational Semantics: Lecture Notes for the Computer Science Tripos, Part II.* 2012. URL: `https://www.cl.cam.ac.uk/teaching/1112/DenotSem/dens-notes-bw.pdf` (cit. on p. 2).

[Rie93] Jon G. Rieke. "Fully abstract translations between functional languages". In: *Mathematical Structures in Computer Science* 3 (1993), pp. 387–415. DOI: `10.1017/S0960129500000293` (cit. on p. 3).

[Sco70] Dana Scott. *Outline of a Mathematical Theory of Computation.* Tech. rep. PRG02. Oxford University Computing Laboratory, Nov. 1970. URL: `https://www.cs.ox.ac.uk/publications/publication3720-abstract.html` (cit. on p. 1).

[Sco93] Dana S. Scott. "A type-theoretical alternative to ISWIM, CUCH, OWHY". In: *Theoretical Computer Science* 121.1–2 (1993), pp. 411–440. DOI: `10.1016/0304-3975(93)90095-B` (cit. on p. 1).

[SS71] Dana Scott and Christopher Strachey. *Towards a Mathematical Semantics for Computer Languages.* Tech. rep. PRG06. Oxford University Computing Laboratory, Aug. 1971. URL: `https://www.cs.ox.ac.uk/publications/publication3723-abstract.html` (cit. on p. 1).

[Str06] Thomas Streicher. *Domain-Theoretic Foundations of Functional Programming.* World Scientific, 2006. DOI: `10.1142/6284` (cit. on pp. 2–3).

[Str94] Thomas Streicher. "A universality theorem for PCF with recursive types, parellel-or and $\exists$". In: *Mathematical Structures in Computer Science* 4.1 (1994), pp. 111–115. DOI: `10.1017/S0960129500000384` (cit. on p. 3).

[Tai67] W. W. Tait. "Intensional interpretations of functionals of finite type I". In: *The Journal of Symbolic Logic* 32.2 (1967), pp. 198–212. DOI: `10.2307/2271658` (cit. on p. 25).

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages.* Ed. by Michael Garey and Albert Meyer. Foundations of Computing. MIT Press, 1993. DOI: `10.7551/mitpress/3054.001.0001` (cit. on p. 2).