Adequacy of PCF

Natalie Ravenhill
Supervisors: Neelakantan Krishnaswami and Martín Escardó

Submitted in conformity with the requirements
for the degree of MSc Advanced Computer Science
School of Computer Science
University of Birmingham

Abstract

Adequacy of PCF

Natalie Ravenhill

We give a proof of Adequacy, a theorem relating the Denotational Semantics of PCF, a typed language with recursion, to its Operational Semantics, using the Logical Relations technique. To do this, we first develop Operational and Denotational Semantics of PCF, proving Type Safety and some necessary lemmas for this, and Soundness of the semantics of PCF. Finally we consider whether all possible elements in our denotational model are denotations of syntactically correct PCF terms.

Keywords: *Computational Adequacy, Denotational Semantics, Domain Theory, Logical Relations, Operational Semantics, PCF*

Acknowledgements

I must thank my first supervisor, Dr. Neelakantan Krishnaswami for his guidance and encouragement, particularly in helping me to understand Domain Theory and Type Safety and for enabling me to make a lot of progress with my project before he left the university.

I would also like to thank Dr. Martín Escardó for taking over the supervision of my project, helping me to finish my proof of Adequacy correctly and for enabling me to extend my project past my proposal and consider other aspects of PCF. Also, for taking so much time to help me write my report!

In addition, I would like to thank Dr. Paul Levy for organising and enabling me to attend the Midlands Graduate School in the Foundations of Computer Science and Prof. Achim Jung for delivering the Denotational Semantics course there, which inspired this project as I enjoyed it so much and wanted to understand the content more.

Finally I would like to thank Dr. Paul Levy and Dr. Rowanne Fleck for attending my demonstration and offering helpful advice to complete my report.

Contents

Chapter 1

Introduction

Semantics is the study of the meaning of computer programs. We require semantics because we can easily compile syntactically correct programs in some language into machine code, but this does not tell us how they actually behave. There are many different approaches to semantics, but the main three are Operational [Plotkin, 1981], Denotational [Scott, 1993] and Axiomatic Semantics [Hoare, 1969].

In this report, we are interested in the first two approaches. Operational Semantics describes how a syntactic program is reduced to some other syntactic value either by using a transition relation, or ”running” the program on an abstract machine. In Denotational Semantics, we construct a mathematical model of the language and analyse that instead of the syntax of the language.

In general, Denotational Semantics is considered to be more abstract, enabling us to remove detail that is unnecessary for analysing behaviour and just makes proofs less readable, whereas Operational Semantics can be more easily linked to an implementation of a language. Also, the Operational Semantics of a language are usually proved to be sound with relation to the Denotational Semantics, in a theorem called Soundness (see Section 6.3). In the other direction the theorem is usually referred to as Completeness. We will see that a result this strong cannot be proved for all languages, so instead we relate the denotational model of PCF to its Operational Semantics using a theorem called Adequacy (see Chapter 7). Therefore having both of these approaches to semantics is beneficial to analyse a the language in different ways. For example, later in the report we prove a property called Type Safety (see Chapter 5), for which we use just the Operational Semantics of the language. In contrast, to prove that the Operational Semantics is sound we require both approaches.

1.1 History

Early examples of creating a mathematical model for a programming language include Landin's model for ALGOL 60 using the λ calculus [Landin, 1964, 1965] and Gordon's denotational model for LISP [Gordon, 1973].

The language we model is PCF (Programming Computable Functions). It is a programming language that is analysed by the research community more often than it is actually compiled and used to write programs. However, it forms a subset of popular functional languages like ML [Milner, 1997] and Haskell [Marlow et al., 2010].

It was first discussed as a programming language and given semantics by Plotkin, [Plotkin, 1977], but is inspired by a "Logic of Computable Functions" defined in [Scott, 1993] (a very famous article that was widely known and written in 1969, but not published until 1993 at Scott's request) and described by [Milner, 1973]. In [Scott, 1993], a logical system is defined and an explanation of how it can modelled by specific mathematical structures is given. The structures used are partially ordered domains of continuous functions (which we define in section 2.1).

A theorem prover was created for LCF in the work of [Milner, 1972a], with an explanation of its working given by [Milner, 1972b], in which the syntax and semantics of a simple language are described in the logic. It was applied to describe and prove the correctness of a compiler in [Milner and Weyhrauch, 1972].

LCF was developed into the modern proof assistants HOL and Isabelle [Nipkow et al., 2002] and influenced many other systems. Also, it resulted in the creation of ML, which started as an aid for LCF, as discussed in [Gordon, 2000].

Therefore although PCF is a research language, it has many applications, so studying it is worthwhile. [Plotkin, 1977] defines the theorem of Adequacy, which is what we ultimately prove, by using a technique called logical relations, which is a different approach to Plotkin's proof, but also documented in modern textbooks such as [Streicher, 2006].

1.2 Structure

In Chapter 2, we explain Domain Theory, a area of mathematics studying the structures that we use to specify our Denotational Semantics and logical relations, a proof technique required for proving Adequacy.

In chapter 3, we define the syntax of PCF and give the typing rules, which all syntactically correct PCF programs must satisfy. In Chapter 4, we specify the Operational Semantics of PCF and in Chapter 5, we prove Type Safety, which states that all correctly typed PCF programs must have a valid evaluation in the Operational Semantics if they are expected to.

In Chapter 6, we define our Denotational Semantics of PCF and prove that the Operational Semantics are sound with respect to this. In Chapter 7, we prove the Adequacy theorem and in Chapter 7.4 we discuss other approaches to the proof of Adequacy. In Chapter 8, we discuss how there are functions in the denotational model that cannot be defined in PCF.

Finally our work is evaluated (in Chapter 9) and concluded (in Chapter 10).

Chapter 2

Background

There are some mathematical topics and proof techniques, which are very important in semantics, that are used in this report. To prove Adequacy, we need a model for our language in which we can work.

The most obvious choice is to model the language in sets, where we can use a set to model each type in PCF. However, for function types, if a function of some type never terminates, then its result will not be in the set of that type, and we cannot model it. Therefore we need some additional "undefined" element in the model to represent this.

We also want to compare different programs in the model, which we can do by using an information ordering, where more defined programs are higher in the ordering and the element representing non-termination is the lowest. The usefulness of the ordering will become apparent when we discuss recursion (see 2.4).

A good ordering to use is a partial order, which is defined as follows:

Definition 1. *A partial order on a set X is a binary relation $\sqsubseteq$ that it is reflexive, anti-symmetric and transitive, which are the following properties:*

- $\forall x \in X,\ x \sqsubseteq x$ *Reflexivity*
- $\forall x, y \in X.\ x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$ *Antisymmetry*
- $\forall x, y, z \in X.\ x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$ *Transitivity*

Therefore, we need a structure which includes an underlying set, an partial order on the set and an undefined element - this is a **domain**.

2.1 Domain Theory

There is an entire area of mathematics devoted to exploring these structures called Domain Theory, as described in [Gunter, 1992] and [Hutton, 2014].

Generally domains can be defined in two different ways, either by using chains or by using directed sets.

Definition 2. *A* ***chain*** *is a sequence $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$ where every element is larger than the preceding one.*

Formally we define a chain as a function $\mathbb{N} \rightarrow X$, such that if $i \leq j$ then $x_i \sqsubseteq x_j$

As an example, given the lifted set $\mathbb{N}_\perp$ (a set with $\perp$ added) then a chain can be formed in three ways:

- $\perp \sqsubseteq \cdots \sqsubseteq \perp$, for any number of $\perp$s
- $n \sqsubseteq \cdots \sqsubseteq n$, for any number of ns, where n is the same number each time
- $\perp \sqsubseteq \cdots \sqsubseteq \perp \sqsubseteq n \sqsubseteq \cdots \sqsubseteq n$, for any number of $\perp$s followed by any number of identical ns

This is because the ordering on the lifted set is only defined between numbers and themselves, so it only contains $\perp \sqsubseteq \perp$ and $\perp \sqsubseteq n$ for some $n \in \mathbb{N}$.

Definition 3. *A* ***directed set*** *X, is a non-empty set where*

$$\forall x, y \in X.\ \exists z \in X.\ x \sqsubseteq z\ \wedge y\ \sqsubseteq z$$

Domains can be defined using either of these structures. If we take all the elements in a chain as a set, then the least upper bound will be the value of z for any two elements of the set. Therefore a chain is an example of a directed set.

For the rest of the report, we use the definition of domains that uses chains, as opposed to directed sets.

Definition 4. *A domain $(X, \perp, \sqsubseteq)$ consists of a set X, an element $\perp$ and a relation $\sqsubseteq\ \subseteq X_\perp \times X_\perp$ such that:*

- $\forall x \in X.\ \perp \sqsubseteq X$
- *$\sqsubseteq$ is a partial order*

- *All chains of elements of $X_\perp$ have a limit (i.e. a least upper bound). To prove this there are two properties we must prove, for some $z \in X_\perp$*
  - $\forall i.x_i \sqsubseteq z$ (z is the upper bound)
  - $\forall y.(\forall i.x_i \sqsubseteq y) \Rightarrow z \sqsubseteq y$ (z is the least upper bound)

The limit of a chain is usually written as $\bigsqcup x_n$, where x_n is a chain of length n and x_i is the element at the ith position in the chain.

2.2 Examples of Domains

2.2.1 Single Element Domain

In a single element domain the underlying set is $\{x\}$, so the only element $\perp$ can be is x and $\sqsubseteq$ just contains the pair (x, x)

Lemma 1. *$(\{x\}, x, \sqsubseteq)$ is a domain.*

Proof. We prove the three conditions in the domain definition to show that $(\{x\}, x, \sqsubseteq)$ is a domain).

There is only one element, x, and $x \sqsubseteq x$ is in the ordering, so $\forall x \in \{x\}.\ x \sqsubseteq x$.

Next, we prove $\sqsubseteq$ is a partial order. As the only element in the underlying set is x and $x \sqsubseteq x$ is in the ordering, then it must be reflexive. We can only have $x \sqsubseteq x$ in the ordering and $x = x$, so it is antisymmetric. Any x, y and z must all be x and $x \sqsubseteq x$ is in the ordering, so $\sqsubseteq$ is transitive. Therefore $\sqsubseteq$ is a partial order.

Finally, we prove that all chains must have a least upper bound. As x is the only possible element, all chains will be of the form

$$x \sqsubseteq x \sqsubseteq x \sqsubseteq \dots$$

Let $z = \bigsqcup x_n = x$. Then the only possible x_i is x, so we must have $x \sqsubseteq x$. This is in the ordering. Therefore $\forall i.x_i \sqsubseteq x$.

Then we prove $\forall y.(\forall i.x_i \sqsubseteq y) \Rightarrow x \sqsubseteq y$. The only possible value of y is x. Therefore we must have $x \sqsubseteq x$, which is in the ordering.

Now we have proved all the conditions, so the single element domain is a domain. □

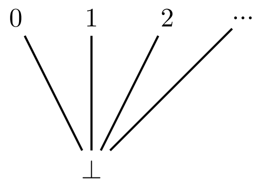
2.2.2 Flat Natural Numbers

A domain is a **flat** domain if the only ordering is between $\perp$ and the elements of the underlying set. In this example the underlying set is $\mathbb{N}$, the set of natural numbers, so the only orderings we have are $\perp \sqsubseteq 0$, $\perp \sqsubseteq 1$, $\perp \sqsubseteq 2$, etc...

Therefore the relation is defined as:

$$\sqsubseteq \; = \{(\perp, \perp)\} \cup \{(\perp, n), (n, n) \mid n \in \mathbb{N}\}$$

which gives us the following picture:



Now we must prove that this is a domain:

Lemma 2. $(\mathbb{N}, \perp, \sqsubseteq)$ *is a domain.*

Proof. We prove the three conditions in the domain definition:

In the definition of our relation we have $\{(\perp, n) \mid n \in \mathbb{N}\}$, so $\forall x \in \mathbb{N}.\ \perp \sqsubseteq x$.

Next, we prove $\sqsubseteq$ is a partial order. From the definition of $\sqsubseteq$, we have $\{(x, x) \mid x \in \mathbb{N}_\perp\}$, which is a subset of $\sqsubseteq$, so $\sqsubseteq$ must be reflexive.

We prove antisymmetry by case analysis. When $x = \perp$, the only possible y we can have such that $y \sqsubseteq x$ is $y = \perp$, as $n \sqsubseteq \perp$ is not defined in the relation for any n. Therefore $x = y = \perp$. When $x = n$, the only possible value of y is n, so $x = y = n$. Therefore every element of the underlying set satisfies antisymmetry.

We also prove transitivity by case analysis. If $x = \perp$ and $y = n$, then we must have $z = n$ for (y, z) to be in $\sqsubseteq$. Then we need $\perp \sqsubseteq n$, which we have, as $(\perp, n)$, for any $n \in \mathbb{N}$ is defined in the relation. If $x = \perp = y$, then we have two options for z. When $z = n$, we should have $\perp \sqsubseteq n$, which we have, as $(\perp, n)$ for any $n \in \mathbb{N}$ is defined in the relation. When $z = \perp$, we want $\perp \sqsubseteq \perp$, which is also in the definition of $\sqsubseteq$. If $x = n$, then both y and z must also be equal to n for $x \sqsubseteq y$ and $y \sqsubseteq z$ to be defined. Therefore we should have $n \sqsubseteq n$. This is in the definition of $\sqsubseteq$.

Finally we prove that all chains must have a least upper bound. We prove this by case analysis on the different chains. For chains of the form $\bot \sqsubseteq \cdots \sqsubseteq \bot$, let $z = \bot$. The last element in the chain will always be $\bot$, so for every i we have $\bot \sqsubseteq \bot$. Therefore $\forall i.x_i \sqsubseteq \bot$. For the second statement, as every element is $\bot$, $x_i = \bot$ and $y = \bot$, we have $\bot \sqsubseteq \bot$ for $z \sqsubseteq y$. Therefore $\forall y.(\forall i.x_i \sqsubseteq y) \Rightarrow \bot \sqsubseteq y$ holds.

For chains of the form $n \sqsubseteq \cdots \sqsubseteq n$, let $z = n$. The last element in the chain will always be n, so for every n we have $n \sqsubseteq n$. Therefore $\forall i.x_i \sqsubseteq n$. For the second part, every element is n, so $x_i = n$ and $y = n$. Then we have $n \sqsubseteq n$ for $z \sqsubseteq y$. Therefore $\forall y.(\forall i.x_i \sqsubseteq y) \Rightarrow n \sqsubseteq y$ holds.

For chains of the form $\bot \sqsubseteq \cdots \sqsubseteq \bot \sqsubseteq n \sqsubseteq \cdots \sqsubseteq n$, let $z = n$. The last element will be n. We have both $\bot \sqsubseteq n$ and $n \sqsubseteq n$ in the relation, so for any x, we have $x \sqsubseteq n$. Therefore $\forall i.x_i \sqsubseteq n$. For the second part, $(\forall i.x_i \sqsubseteq y)$ is only true when $y = n$, so we only have to consider this case. Then we have $n \sqsubseteq n$ for $z \sqsubseteq y$. Therefore $\forall y.(\forall i.x_i \sqsubseteq y) \Rightarrow n \sqsubseteq y$ holds. □

2.2.3 Product of domains

Given two domains $\mathbb{X} = (X, \bot_X, \sqsubseteq_X)$ and $\mathbb{Y} = (Y, \bot_Y, \leq_Y)$, we have a new domain, where $X \times Y$ is the underlying set, the bottom element is $(\bot_X, \bot_Y)$ and $(x, y) \sqsubseteq (x', y')$ is defined when $x \sqsubseteq_X x'$ and $y \leq_Y y'$ are defined.

Now we must prove that this is a domain.

Lemma 3. $(X \times Y, (\bot_X, \bot_Y), \sqsubseteq)$ *is a domain*

Proof. We prove the three conditions in the domain definition:

As $\mathbb{X}$ is a domain, we know $\forall x \in X.\ \bot_X \sqsubseteq_X x$ and because $\mathbb{Y}$ is a domain, we know $\forall y \in Y.\ \bot_Y \leq_Y y$. Therefore we have $\forall x, y.\ \bot_X \sqsubseteq_X x \wedge \bot_Y \leq_Y y$. This is the same as $\forall (x, y) \in X \times Y.(\bot_X, \bot_Y) \sqsubseteq (x, y)$.

Next, we prove $\sqsubseteq$ is a partial order. For an element $(x, y) \in X \times Y$, we have $x \sqsubseteq_X x$ and $y \leq_Y y$ because $\mathbb{X}$ and $\mathbb{Y}$ are domains, so their orderings are reflexive. This means we have $(x, y) \sqsubseteq (x, y)$, so $\sqsubseteq$ is also reflexive. For elements (x, y) and (x', y') we can assume $(x, y) \sqsubseteq (x', y')$ and $(x', y') \sqsubseteq (x, y)$. Expanding these definitions we have $x \sqsubseteq_X x' \wedge y \leq_Y y' \wedge x' \sqsubseteq_X x \wedge y' \leq_Y y$. If we reorder this we have:

$$x \sqsubseteq_X x' \wedge x' \sqsubseteq_X x \wedge y \leq_Y y' \wedge y' \leq_Y y$$

As the orderings on $\mathbb{X}$ and $\mathbb{Y}$ are antisymmetric, we can rewrite this as $x = x'$ and $y = y'$. Therefore we have $(x, y) = (x', y')$, so $\sqsubseteq$ is antisymmetric. For elements $(x, y), (x', y')$ and (x'', y'') we can assume $(x, y) \sqsubseteq (x', y')$ and $(x', y') \sqsubseteq (x'', y'')$. Expanding these definitions gives us $x \sqsubseteq_X x' \ \wedge \ y \leq_Y y' \wedge \ x' \sqsubseteq_X x'' \wedge y' \leq_Y y''$. If we reorder this we have:

$$x \sqsubseteq_X x' \wedge x' \sqsubseteq_X x'' \wedge y \leq_Y y' \wedge y' \leq_Y y''$$

As the orderings on $\mathbb{X}$ and $\mathbb{Y}$ are transitive, we can rewrite this as $x \sqsubseteq_X x''$ and $y \leq_Y y''$. Therefore we can now define $(x, y) \sqsubseteq (x'', y'')$, so $\sqsubseteq$ is transitive.

Finally we prove that all chains have a least upper bound. Chains of $\mathbb{X} \times \mathbb{Y}$ will be of the form:

$$(x, y) \sqsubseteq (x', y') \sqsubseteq (x'', y'') \sqsubseteq \dots$$

where $x \sqsubseteq_X x' \sqsubseteq_X x'' \dots$ and $y \leq_Y y' \leq_Y y'' \dots$.

Let $z = \bigsqcup (x, y)_n = (\bigsqcup x_n, \bigsqcup y_n)$. Then as $\mathbb{X}$ and $\mathbb{Y}$ are domains, we have $\forall i.\ x_i \sqsubseteq_X \bigsqcup x_n$ and $\forall i.\ y_i \leq_Y \bigsqcup y_n$. Therefore, for any (x, y) we have $\forall i. (x_i, y_i) \sqsubseteq (\bigsqcup x_n, \bigsqcup y_n)$.

As $\mathbb{X}$ and $\mathbb{Y}$ are domains, we have $\forall x'.\ (\forall i.\ x_i \sqsubseteq_X x') \Rightarrow \bigsqcup x_n \sqsubseteq_X x'$ and $\forall y'.\ (\forall i.\ y_i \leq_Y y') \Rightarrow \bigsqcup y_n \leq_Y y'$.

Therefore if we assume $\forall (x', y'). (\forall i. (x_i, y_i) \sqsubseteq (x', y'))$, then we know $\bigsqcup x_n \sqsubseteq_X x'$ and $\bigsqcup y_n \leq_Y y'$. This is the definition of $(\bigsqcup x_n, \bigsqcup y_n) \sqsubseteq (x', y')$.

Now we have proved all the conditions, so the product of two domains is also a domain. □

2.3 Monotone and Continuous Functions

There are two different types of functions that we will use when modelling PCF functions:

Definition 5. *A* ***monotone*** *function, f, is a function that preserves the order of a partially ordered set, X, so:*

$$\forall x, y \in X.\ x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

Given a chain $x_0 \sqsubseteq x_1 \sqsubseteq \dots$, we can form the chain $f(x_0) \sqsubseteq f(x_1) \sqsubseteq \dots$ using a monotone function.

Definition 6. *A* ***continuous*** *function f is a function which when applied to the limit of a chain gives the same result as the limit of the chain formed by applying f to every element of another chain. Formally:*

$$f(\bigsqcup x_n) = \bigsqcup(f(x_n))$$

Therefore continuous functions must also be monotone:

Theorem 1. *Continuous functions are monotone*

Proof. Given a continuous function f, on a partially ordered set X, we need to show that $\forall x, y \in X.\ x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. Assume we have a chain x_n, where $x_0 = x$ and $x_{n+1} = y$ for all n:

$$x \sqsubseteq y \sqsubseteq y \ldots$$

.

The limit of this chain will be y, so $f(\bigsqcup x_n) = f(y) = \bigsqcup(f(x_n))$.

We also know that because there are only two different elements in the chain, $\bigsqcup(f(x_n)) = f(x) \sqcup f(y) = f(y)$, so therefore $f(x) \sqsubseteq f(y)$.

□

2.3.1 Domain of Continuous Functions

We can form a domain of continuous functions between two other domains:

Given two domains $\mathbb{X} = (X, \perp_X, \sqsubseteq_X)$ and $\mathbb{Y} = (Y, \perp_Y, \leq_y)$, we can form the set $\text{Cont}(X, Y) = \{f : X \to Y\}$, of continuous functions between the underlying sets, where:

- $\forall x, x' \in X.\ x \sqsubseteq_X x' \Rightarrow f(x) \leq_Y f(x')$ f preserves the ordering of chains in $\mathbb{X}$
- $x_n \in \text{Chain}(X) \Rightarrow f(\bigsqcup x_n) = \bigsqcup f(x_n)$ f is continuous

where $\text{Chain}(X)$ is the set of all possible chains we can form from $\mathbb{X}$.

$\perp_{X \to Y}$ is defined as the function $\perp = \lambda x.\perp(x)$, the function that loops on all inputs. The output of this function will always be $\perp$, because it does not terminate.

The relation $\sqsubseteq_C$ is defined as

$$\sqsubseteq_C = \{(f, g) \mid f, g \in \text{Cont}(X, Y) \ \wedge\ \forall x \in X.\ f(x) \leq_Y g(x)\}$$

Therefore our domain will be $(\mathrm{Cont}(X, Y), \bot_{X \to Y}, \sqsubseteq_C)$. Now we must prove that this is a domain.

Lemma 4. $(\mathrm{Cont}(X, Y), \bot_{X \to Y}, \sqsubseteq_C)$ *is a domain.*

Proof. We must prove the three conditions in the domain definition:

For all $x \in X$ we have $\bot \leq_Y f(x)$. As $\mathbb{Y}$ is a domain we know this holds for every element of Y and as the codomain of f is Y, every $f(x)$ is in Y. Therefore $\forall f \in \mathrm{Cont}(X, Y).\ \bot_{X \to Y} \sqsubseteq_C f$.

Next we prove $\sqsubseteq_C$ is a partial order. As $\mathbb{Y}$ is a domain, we know that $\leq_Y$ is a partial order. For reflexivity, we need to prove that $\forall f \in \mathrm{Cont}(X, Y).\ f \sqsubseteq_C f$. We can rewrite this using the definition of $\sqsubseteq_C$ to get

$$\forall f \in \mathrm{Cont}(X, Y).\ (\forall x \in X.\ f(x) \leq_Y f(x))$$

Functions are single valued, so we know $\forall f.\ \forall x.\ f(x) = f(x)$ and as $\leq_Y$ is reflexive we know $\forall f. \forall x \in X.\ f(x) \leq_Y f(x)$. Therefore we have $f \sqsubseteq_C f$, for any $f \in \mathrm{Cont}(X, Y)$. For antisymmetry, we need to prove that $\forall f, g \in \mathrm{Cont}(X, Y).\ ((f \sqsubseteq_C g) \ \wedge\ (g \sqsubseteq_C f)) \Rightarrow f = g$. Rewriting this using the definition of $\sqsubseteq_C$ gives us

$$\forall f, g \in \mathrm{Cont}(X, Y).\ (\forall x \in X.\ ((f(x) \leq_Y g(x)) \ \wedge\ (g(x) \leq_Y f(x))) \Rightarrow f(x) = g(x))$$

$\leq_Y$ is antisymmetric, so we have $\forall x \in X.\ f(x) = g(x)$, for any values of f and g. Therefore $\sqsubseteq_C$ is also antisymmetric. For transitivity, we need to prove that $\forall f, g, h \in \mathrm{Cont}(X, Y).\ ((f \sqsubseteq_C g) \ \wedge\ (g \sqsubseteq_C h)) \Rightarrow f \sqsubseteq_C h$. Rewriting this using the definition of $\sqsubseteq_C$ gives us

$$\forall f, g, h \in \mathrm{Cont}(X, Y).\ (\forall x \in X.\ ((f(x) \leq_Y g(x)) \ \wedge\ (g(x) \leq_Y h(x))) \Rightarrow f(x) \sqsubseteq_C h(x))$$

As $\leq_Y$ is transitive, we have $\forall x \in X. f(x) \leq_Y h(x)$, for all f, g and h. Therefore $\sqsubseteq_C$ is also transitive.

Finally we prove that all chains have a least upper bound. Let $z = \lambda x. \bigsqcup^Y f_n(x)$, where $\bigsqcup^Y f_n(x)$ is the limit of the chain (in $\mathbb{Y}$) obtained by applying the functions in some chain of elements of $\mathrm{Cont}(X, Y)$ to a certain element $x \in X$.

An example of a chain of such functions is:

$$f_1 \sqsubseteq_C f_2 \sqsubseteq_C \dots \sqsubseteq_C \bigsqcup f_n$$

If we expand this using the definition of $\sqsubseteq_C$ we have

$$\forall x \in X.\ (f_1(x) \leq_y f_2(x) \leq_Y \cdots \leq_Y \bigsqcup f_n(x))$$

This is a set of chains in $\text{Chain}(Y)$ where every chain contains the result of each function on a certain $x \in X_\perp$. As $\mathbb{Y}$ is a domain, the least upper bound is defined for any chain using the elements of $Y_\perp$. Therefore we know that the least upper bound $\bigsqcup f_n(x)$ is defined. Now we can see that this is the same as our definition of z, which was $\lambda x. \bigsqcup^Y f_n(x)$.

For the second part of the proof, we can rewrite it using the definition of $\sqsubseteq_C$ as

$$\forall x \in X.\ (\forall g.(\forall i. f_i(x) \leq_Y g(x)) \Rightarrow z(x) \leq_Y g(x))$$

As $\mathbb{Y}$ is a domain, $(\forall i. f_i(x) \leq_Y g(x)) \Rightarrow z(x) \leq_Y g(x))$ holds for each of our individual chains for each $x \in X$. Therefore we have $\forall g.(\forall i. f_i \sqsubseteq_C g) \Rightarrow z \sqsubseteq_C g$ □

2.4 Fixpoint Theorem

Now we have a domain of continuous functions, we can use it to state and prove the fixpoint theorem. The following theorem is an important result in recursion theory, that we will use to model recursion in PCF. The chain given in the theorem is the chain obtained by repeatedly iterating a recursive function on its previous result, starting with $\perp$. If an input of the function is computed using $f^n(\perp)$ and it needs more than n iterations then it will not terminate. As the chain can be infinitely long, it can model infinite (general) recursion.

Theorem 2. *For every domain X, every continuous function $f : X \to X$ has a least fixpoint, which is the limit of the chain $\perp \sqsubseteq f(\perp) \sqsubseteq f^2(\perp) \sqsubseteq \dots$*

Proof. Assume we have a domain X and a continuous function $f : X \to X$. First we must show that we can define the chain in the theorem using f. To do this we must show that $f^n(\perp) \sqsubseteq f^{n+1}(\perp)$, for any n. We prove this by induction on n. $f^0(\perp) = \perp$ and $\perp \sqsubseteq f(\perp)$, as $\perp$ is the least element. For the inductive case, we know that $f^n \sqsubseteq f^{n+1}$, and that continuous functions are monotone (see Lemma 1), so we have $f(f^n(\perp)) \sqsubseteq f(f^{n+1}(\perp)) = f^{n+1}(\perp) \sqsubseteq f^{n+2}(\perp)$. Therefore we have proved the above statement for all cases, so we know we can form the chain. Define the fixpoint function $\mathsf{fix}(f) \equiv \bigsqcup f^n(\perp)$. This is the limit of the chain in the theorem. We know this limit exists because $\mathbb{X}$ is a domain, so all chains in $\mathbb{X}$ have a limit.

$\bigsqcup f^n(\perp)$ is a fixpoint For the limit to be a fixpoint we must have $f(\bigsqcup f^n(\perp)) = \bigsqcup f^n(\perp)$. As f is continuous, we have $f(\bigsqcup f^n(\perp)) = \bigsqcup f(f^n(\perp)) = \bigsqcup f^{n+1}(\perp)$. As we have applied f

to the whole chain to get the new limit, we now have a chain $f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$. This is the same as our original chain, but without $\bot$ at the start. Because $\mathbb{X}$ is a domain, we know that $\forall x \in X.\ \bot \sqsubseteq x$. Therefore $\bot$ has no effect on the limit because every element is higher than it, so removing $\bot$ will not change the limit. This means that $\bigsqcup f^{n+1}(\bot) = \bigsqcup f^n(\bot)$.

$\bigsqcup f^n(\bot)$ is the least fixpoint Let x be an element of our chain such that $f(x) = x$. Then for $\bigsqcup f^n(\bot)$ to be the least fixpoint, we must have $\bigsqcup f^n(\bot) \sqsubseteq x$ (i.e. so x is an upper bound that is higher than $\bigsqcup f^n(\bot)$). First we prove x is an upper bound, so we must show $\forall n.\ f^n(\bot) \sqsubseteq x$. We prove by this by induction on n. If $n = 0$, then $f^0(\bot) \sqsubseteq x$, This is the same as $\bot \sqsubseteq x$, which is true because $\bot$ is the least element of the chain. Our inductive hypothesis is $f^n(\bot) \sqsubseteq x$. As f is continuous, f is monotone, so $f(f^n(\bot)) \sqsubseteq f(x) = f^{n+1}(\bot) \sqsubseteq x$. Therefore we know that for any element $f^n(\bot)$ in the chain, $f^n(\bot) \sqsubseteq x$.

As $\bigsqcup f^n(\bot)$ is a least upper bound, we know that $\forall x \in X.\ \forall n.\ (f^n(\bot) \sqsubseteq x) \Rightarrow \bigsqcup f^n(\bot) \sqsubseteq x$. We have just proved the left hand side of this, so we now have $\bigsqcup f^n(\bot) \sqsubseteq x$. Therefore we have proved that $\bigsqcup f^n(\bot)$ is the least fixpoint of f. $\square$

Now that we have proved the above theorem, we know enough Domain Theory to model recursive PCF programs of any type.

2.5 Logical Relations

Logical Relations, developed in [Tait, 1967],[Plotkin, 1973],[Statman, 1985], is a proof technique used for establishing properties that cannot be proved by structural induction on program terms alone, due to higher order constructions being present in the structure we are proving a property of.

Instead we define a family of relations containing a relation for each type, and the property we want to prove is contained in the definition of the relation at base type. We then prove a Main Lemma on the logical relation. The proof of our original property is obtained as a corollary of this Main Lemma.

Logical Relations have been used to prove Strong Normalisation (i.e. that every expression terminates) of systems such as the Simply Typed λ Calculus. They have also been used to prove Type Safety and Program Equivalence.

We construct two logical relations in our report:

- A relation between program terms and their denotations (in Chapter 7)
- A relation between program denotations (in Chapter 8)

Chapter 3

Definition of PCF and Syntax

Programming Computable Functions (PCF) is a programming language that is based on the Simply Typed λ Calculus, with the addition of a "fix" operator, that allows us to write recursive functions using fixpoint recursion.

3.1 Definition of PCF

3.1.1 Types

There are different versions of PCF in certain papers and books (such as [Plotkin, 1977], [Gunter, 1992]) where some use Booleans and Natural Numbers for the base types. Here we just use Natural numbers, where 0 represents False and any non zero number represents True. We define our types using the following grammar:

$$A ::= \text{Nat} \mid A \to B$$

3.1.2 Expressions

The allowable expressions include variables (represented by x), a constant z representing the number 0, and a successor function $s(e)$, which takes any natural number expression as input and returns its successor.

case takes an expression e, which we assume is a numerical value, as input. If e is zero, we return an expression e_0, otherwise we have the successor of some value x and we return the expression e_S.

Then we have function application, in which a function e is applied to an expression e', and λ-abstraction, which denotes a function e that takes an input x of type A.

Our last expression is the fixpoint expression, which takes a value x as input to a larger function e.

The grammar for expressions is:

$$e ::= \; x \mid z \mid s(e) \mid \text{case } (e, z \mapsto e_0, s(x) \mapsto e_S) \mid e\ e' \mid \lambda x : A.e \mid \text{ fix } \; x : A.\ e$$

3.2 Type System for PCF

Γ is an example of a typing context, which is a function that maps variables to their types. For example, if we have an expression $x : A$ in the context, then $\Gamma(x) = A$. We write $\Gamma \vdash e : A$ for the context associated with an expression e of type A.

Therefore we need a typing rule for each expression in PCF, which if satisfied means that we are allowed to write that expression. The typing rules are given as inference rules, where our assumptions are above the line and the conclusion underneath:

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\,\text{VARIABLES} \qquad \frac{}{\Gamma \vdash z : \text{Nat}}\,\text{ZERO} \qquad \frac{\Gamma \vdash e : Nat}{\Gamma \vdash s(e) : \text{Nat}}\,\text{SUCC}$$

For variables, if x is in the domain of Γ, then we can conclude that we have a variable of that type.

For zero, z is a constant, so it needs no assumptions.

For successor, we must have an expression e of type Nat that we can apply the successor function to. We then know we have $s(e)$ of type Nat.

$$\frac{\Gamma \vdash e : \text{Nat} \qquad \Gamma \vdash e_0 : A \qquad \Gamma, x : \text{Nat} \vdash e_S : A}{\Gamma \vdash \text{ case } (e, z \mapsto e_0, s(x) \mapsto e_S) \ : A}\,\text{CASE}$$

For case, we must have an expression e of type Nat to evaluate. Then we must have some other expressions e_0 and e_S to return, which can be of any type, as long as they are the same type. As the condition of e_S contains a specific x value, we must also know that this is well typed. Therefore we add x : Nat to the context of e_S.

$$\frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e\ e' : B}\ \textsc{application} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.e : A \to B}\ \textsc{abstraction}$$

For function application, e must have a function type and e' must have the same type as the domain of e. Then we can apply e to e' to get the expression $e\ e'$ of the type of the codomain of e.

For λ-abstraction, we need an expression e of some type B and we must know that the parameter x is in the context of this expression, to be able to use it in the λ abstraction. Then in the conclusion, as x is now bound to the expression, we remove it from the context of B.

$$\frac{\Gamma, x : A \vdash e : A}{\Gamma \vdash \ \text{fix}\ \ x : A.\ e : A}\ \textsc{fix}$$

The fixpoint case is the same as λ abstraction, except that the x is bound to the fixpoint expression and is of the same type as the entire expression.

For any well typed PCF expression, we can obtain a derivation tree, where the root is the whole expression and the branches go up until we end up with the variables and constants of the expression at the leaves.

We also note that given a typing context Γ, an expression e and a type A such that we have the typing judgement $\Gamma \vdash e : A$, there is only one possible derivation of this judgement.

Chapter 4

Operational Semantics of PCF

Now we have defined the syntax and typing rules of PCF, we can use this to define its Operational Semantics.

We use the **call by name** evaluation strategy, which means that function arguments are placed into the body of the function and evaluated within the entire function's evaluation, instead of before.

The semantics we define are **small step** semantics, which means that in $e \mapsto e'$, the transition relation $\mapsto$ must take an expression e to another expression e' in only one step.

The first rules we have are **congruence rules**, which use the assumption $e \mapsto e'$ to replace e with e' in the whole expression. We can define these rules for any PCF expression that has an expression as a parameter, so this will be function application, successor and case:

$$\frac{e_0 \mapsto e_0'}{e_0\ e_1 \mapsto e_0'\ e_1} \qquad \frac{e \mapsto e'}{s(e) \mapsto s(e')}$$

$$\frac{e \mapsto e'}{\text{case } (e, z \mapsto e_0, s(x) \mapsto e_S) \mapsto \text{case } (e', z \mapsto e_0, s(x) \mapsto e_S)}$$

(Note that we could have given the typing contexts before each expression in the evaluation rules, but as they do not change, we can omit them. The same is also true for the rest of the rules we define below.)

Then we define rules on individual expressions. Note that the case rule above was only defined on expressions that reduce. The one defined below is only defined for **values**, which are expressions that have no applicable evaluation rule (including zero, successors of values, and lambda abstractions). This ensures that there is only one possible rule to apply to any expression.

$$\frac{}{(\lambda x : A.\ e)\ e' \mapsto [e'/x]e}$$

The above rule states that given a function application, where the function is defined by a λ-abstraction, we substitute e' for x in the expression e. This means that we replace every occurrence of x in e with the expression e'.

$$\frac{}{\text{case } (z, z \mapsto e_0, s(x) \mapsto e_S) \mapsto e_0}$$

$$\frac{}{\text{case } (s(v), z \mapsto e_0, s(x) \mapsto e_S) \mapsto [v/x]e_S}$$

The above rules give the evaluation of case, when we have a value as the condition expression. The first rule states that if $e = z$, then our result will be e_0. If $e = s(v)$, our result is e_S, but we must also substitute v for x in e_S (as e_S is defined for a bound variable x, which we now know is equal to v).

$$\frac{}{\text{fix } \ x : A.e \mapsto [\text{fix } \ x : A.e/x]e}$$

The above rule states that we replace the bound variable x in the expression e with the entire fixpoint expression. This replaces a parameter in e, which should be the recursive call, with the contents of the function, so that we can evaluate it all again. Then when we get to that point in the new evaluation, we will replace x with the whole expression. We can keep doing this infinitely, and keep expanding the evaluation in the following way:

$$\text{fix } \ x : A.e \mapsto [\text{fix } \ x : A.e/x]e \mapsto [[\text{fix } \ x : A.e/x]e/x]e \mapsto [[[\text{fix } \ x : A.e/x]e/x]e/x]e \ldots$$

We give an example of a function that uses the fixpoint operator in the following section.

4.1 Example of a program in PCF

4.1.1 Addition

In functional languages, we usually define mathematical operators on numbers as recursive functions. For example addition is the following function:

$$\mathrm{add}\,0\;y = y$$

$$\mathrm{add}\,s(n)\;y = s(\mathrm{add}\,n\;y)$$

We can write this as a λ abstraction:

$$\mathrm{add} = \lambda x, y : \mathrm{Nat}\,.\;\mathrm{case}\;(x, z \mapsto y, s(v) \mapsto s(\mathrm{add}\;\;v\;y))$$

This is a recursive function, so to define it in PCF, it must be the fixpoint of some other function A:

$$A = \lambda f : Nat \to Nat \to Nat.\;\lambda x, y : \mathrm{Nat}\,.\;\mathrm{case}\;(x, z \mapsto y, s(v) \mapsto s(f\;\;v\;y))$$

Therefore we can define this in PCF as:

$$\mathrm{add}\,x\;y = (\mathrm{fix}\,f : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}\,.\;A : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat})\;x\;y$$

.

When we try to evaluate this term, we get the following, by the evaluation rule for fix:

$\mathrm{fix}\,f : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}\,.\;A : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat} = [\mathrm{fix}\,f : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}\,.\;A : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}\,/f]A$. This expands to:

$$\lambda x, y : \mathrm{Nat}\,.\;\mathrm{case}\;(x, z \mapsto y, s(v) \mapsto$$

$$s((\mathrm{fix}\,f : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}\,.\;A : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat})\;v\;y))$$

Therefore expanding this infinitely gives all possible executions of the addition function.

Chapter 5

Type Safety

Type Safety is an important property of a programming language, as it proves that well typed programs do not go wrong. It is usually expressed as a property of the Operational Semantics, which we have defined in the previous chapter (see Chapter 4), and follows from the conclusion of two other lemmas we will prove; Type Preservation (5.2.1) and Type Progress (5.2.2)

5.1 Lemmas for Type Safety

There are two simple lemmas we must prove which will aid us in proving the lemmas of Type Safety:

5.1.1 Weakening

Weakening is the following theorem, which says that for an expression of type A in a context Γ, adding another variable x (of any type) to the context will not change the type of the expression:

Theorem 3. *If* $\Gamma \vdash e : A$ *then* $\Gamma, x : C \vdash e : A$

Proof. We prove this theorem by induction on derivation trees, so if we have a derivation tree for our assumption, then there is a derivation tree for the conclusion.

As there is only one derivation for a given judgement $\Gamma \vdash e : A$, we can use induction on the possible expressions:

Variables We rename x to y using α equivalence. We assume $\Gamma \vdash y : A$, giving us the following derivation tree:

$$\frac{\Gamma(y) = A}{\Gamma \vdash y : A}$$

of which $\Gamma(y) = A$ is a subtree. Γ is a function, so can also be represented by a set of (variable, type) pairs. Therefore $\Gamma, x : C$ is the set $\Gamma\ \cup \{(x, C)\}$, so we define the function $(\Gamma, x : C)$, where $(\Gamma, x : C)(y) = A$ and for any other variable z, $(\Gamma, x : C)(z) = \Gamma(z)$. Then we just use the typing rule for variables to get the following derivation tree:

$$\frac{(\Gamma, x : C)(y) = A}{\Gamma, x : C \vdash y : A}$$

Now we have the required derivation tree, so weakening holds for variables.

Zero We assume $\Gamma \vdash z : \text{Nat}$, giving us the following derivation tree:

$$\frac{}{\Gamma \vdash z : \text{Nat}}$$

. The typing rule for zero says that no matter what Γ is, we always have zero, because there are no assumptions. Therefore we can have $\Gamma, x : C$ as the context and get the following derivation tree:

$$\frac{}{\Gamma, x : C \vdash z : \text{Nat}}$$

Now we have the required derivation tree, so weakening holds for zero.

Successor We assume $\Gamma \vdash s(e) : \text{Nat}$, giving us the following derivation tree from the typing rule:

$$\frac{\Gamma \vdash e : \text{Nat}}{\Gamma \vdash s(e) : \text{Nat}}$$

.

where $\Gamma \vdash e : \text{Nat}$ is a subtree. We can use the inductive hypothesis of weakening on this subtree to get $\Gamma, x : C \vdash e : \text{Nat}$. Then we use the typing rule for successor to get the following derivation tree:

$$\frac{\Gamma, x : C \vdash e : \text{Nat}}{\Gamma, x : C \vdash s(e) : \text{Nat}}$$

.

Now we have the required derivation tree, so weakening holds for the successor function.

Case We assume $\Gamma \vdash$ case $(e, z \mapsto e_0, s(y) \mapsto e_S) : A$, (renaming x to y using alpha equivalence) giving us the following derivation tree from the typing rule:

$$\frac{\Gamma \vdash e : \text{Nat} \qquad \Gamma \vdash e_0 : A \qquad \Gamma, y : \text{Nat} \vdash e_S : A}{\Gamma \vdash \text{ case } (e, z \mapsto e_0, s(y) \mapsto e_S) \ : A}$$

giving us the subtrees $\Gamma \vdash e : \text{Nat}$, $\Gamma \vdash e_0 : A$ and $\Gamma, y : \text{Nat} \vdash e_S : A$ from the assumption of the typing rule. Using the inductive hypothesis on each of these, we get $\Gamma, x : C \vdash e : \text{Nat}$, $\Gamma, x : C \vdash e_0 : A$ and $\Gamma, y : \text{Nat}, x : C \vdash e_S : A$, so we can use the typing rule again with these assumptions:

$$\frac{\Gamma, x : C \vdash e : Nat \qquad \Gamma, x : C \vdash e_0 : A \qquad \Gamma, x : C, y : \text{Nat} \vdash e_S : A}{\Gamma, x : C \vdash \text{ case } (e, z \mapsto e_0, s(y) \mapsto e_S) \ : A}$$

Now we have the required derivation tree, so weakening holds for the case expression.

Application We assume $\Gamma \vdash e_0 \ e_1 : B$, giving us the following derivation tree:

$$\frac{\Gamma \vdash e_0 : A \to B \qquad \Gamma \vdash e_1 : A}{\Gamma \vdash e_0 \ e_1 : B}$$

which gives us the subtrees $\Gamma \vdash e_0 : A \to B$ and $\Gamma \vdash e_1 : A$. Using the inductive hypothesis on these trees gives us $\Gamma, x : C \vdash e_0 : A \to B$ and $\Gamma, x : C \vdash e_1 : A$, so we can just use the typing rule for application again to get the following derivation tree:

$$\frac{\Gamma, x : C \vdash e_0 : A \to B \qquad \Gamma, x : C \vdash e_1 : A}{\Gamma, x : C \vdash e_0 \ e_1 : B}$$

Therefore weakening holds for function application.

λ-Abstraction We rename x to y using α equivalence. We assume $\Gamma \vdash \lambda y : A.\ e : B$, giving us the following derivation tree:

$$\frac{\Gamma, y : A \vdash e : B}{\Gamma \vdash \lambda y : A.\ e : A \to B}$$

which gives us the subtree $\Gamma, y : A \vdash e : B$. Using the inductive hypothesis, we get $\Gamma, y : A, x : C \vdash e : B$. Then we use the typing rule for λ abstraction to get the following tree:

$$\frac{\Gamma, y : A, x : C \vdash e : B}{\Gamma, x : C \vdash \lambda y : A.\ e : A \to B}$$

Now we have the required derivation tree, so weakening holds for λ abstraction.

Fixpoint We assume $\Gamma \vdash \text{fix}\ \ y : A.\ e : A$, renaming x to y using α equivalence. This gives us the following derivation tree:

$$\frac{\Gamma, y : A \vdash e : A}{\Gamma, y : A \vdash \text{fix}\ \ y : A.\ e : A}$$

As we have the subtree $\Gamma, y : A \vdash e : A$, we use the inductive hypothesis on this to get $\Gamma, x : C, y : A. \vdash e : A$. Then we use the typing rule for fix to get the following tree:

$$\frac{\Gamma, x : C, y : A \vdash e : A}{\Gamma, x : C, y : A \vdash \text{fix}\ \ y : A.\ e : A}$$

Therefore weakening holds for the fixpoint operator. Now we have proved weakening for derivation trees of any expression so weakening always holds.

□

5.1.2 Substitution Rules

Next we want to prove a lemma that shows that substitutions preserve the intended type of a given PCF expression.

But before we prove this, we must actually define rules for substitution on the level of each possible expression that can be formed in PCF, which are all instances of $[e/x]e'$. This notation says that an expression e replaces a variable x in another expression e'.

When defining the substitution rules, we must be careful that we avoid **variable capture** by bound variables. For example, given the following substitution:

$$[s(x)/y](\lambda x.x + y)$$

if we naively substitute $s(x)$ for y in $\lambda x.x + y$, we get $\lambda x.\ x + s(x)$ and the value of x is now the value assigned to the bound x. Therefore we can use **renaming** to avoid this.

Variables There are two cases for substitution in variables. The first is for when e' is the same as the variable being replaced:

$$[e/x]x = e$$

The second one is when e' is a completely different variable, for which nothing happens:

$$[e/x]y = y$$

Zero For zero, any substitution will have no effect, as zero is a constant:

$$[e/x]z = z$$

Successor The successor function cannot be changed, so we substitute e in its argument:

$$[e/x]s(e') = s([e/x]e')$$

Case As we cannot change the case statement, we could substitute e in the expressions given as arguments to the case statement, in the following way:

$$[e/x]\ (\text{case}\ (e', z \mapsto e_0, s(x) \mapsto e_S)) = \text{case}\ ([e/x]e', z \mapsto [e/x]e_0, s(x) \mapsto [e/x]e_S)$$

But we must be careful with variable capture, as the derivation of e_S is $\Gamma, x : \text{Nat} \vdash e_S$. If we have $[s(x)/y]e_S$ and $e_S = x + y$, then we have $x + s(x)$ and the value of $s(x)$ is bound by $\Gamma, x : \text{Nat}$. Therefore we should rename $s(x)$ in the case statement to something that is not free in e_S.

Therefore our rule for case will be:

(case $(e', z \mapsto e_0, s(x) \mapsto e_S)) =$

$$\begin{cases} (\text{case } ([e/x]e', z \mapsto [e/x]e_0, s(x) \mapsto [e/x]e_S)) & \text{if } x \notin FV(e') \\ (\text{case } ([e/x]e', z \mapsto [e/x]e_0, s(y) \mapsto [e/x]e_S)) & \text{if } x \in FV(e') \end{cases}$$

where $y \notin FV(e')$.

For some expression e, $FV(e)$ is the set of free variables it contains.

Application We substitute e in the function and its argument, then apply the new function to the new argument:

$$[e/x](e_0\ e_1) = [e/x]e_0\ ([e/x]e_1)$$

λ-Abstraction There are two cases, the first when the bound variable is x. This does nothing, as we are just rewriting the function using alpha equivalence in this case:

$$[e/x](\lambda x : A.\ e') = \lambda x : A.\ e'$$

The second case is when the bound variable is not equal to x, where we substitute e in the expression. If y is a free variable in e' then we must rename the bound variable to something else:

$[e/x](\lambda y : A.e') =$

$$\begin{cases} \lambda y : A.[e/x]e' & \text{if } y \notin FV(e') \\ \lambda z : A.[e/x]([z/y]e') & \text{if } y \in FV(e') \end{cases}$$

where $z \notin FV(e')$.

Fixpoint Fixpoint is similar to λ abstraction, so we have two cases. The first case is when the bound variable is x, for which we just rewrite the function using α equivalence. Therefore this rule does nothing:

$$[e/x]\,\text{fix}\ \ x : A.e' = \text{fix}\ \ e : A.e'$$

When the bound variable is not equal to x, we substitute e in the expression e'. If y is a free variable in e' then we must rename the bound variable to something else:

$[e/x](\text{fix } y : A.e') =$

$$\begin{cases} \text{fix } y : A.[e/x]e' & \text{if } y \notin FV(e') \\ \text{fix } z : A.[e/x]([z/y]e') & \text{if } y \in FV(e') \end{cases}$$

where $z \notin FV(e')$.

5.1.3 Substitution

Now we have all the rules, we can prove Substitution, which is the following theorem. It says that if we have an expression e that is well typed in the context Γ and an expression e' that is well typed in the context $\Gamma, x : A$, then the expression obtained by substituting e for x in e' will be derivable in the context Γ:

Theorem 4. *If* $\Gamma \vdash e : A$ *and* $\Gamma, x : A \vdash e' : C$ *then* $\Gamma \vdash [e/x]e' : C$

Proof. We can also prove this by induction on derivation trees.

As there is only one derivation for a given judgement $\Gamma \vdash e : A$, again we can use induction on the possible values of e':

Variables We assume derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash y : C$. As y has a different type to x, it cannot be equal to it, so there is only one case, as we can only use one of the substitution rules. The tree for y that is given is:

$$\frac{(\Gamma, x : A)(y) = C}{\Gamma, x : A \vdash y : C}$$

The function Γ is the set of pairs $(\Gamma, x : A)\backslash\{(x, A)\}$. As $x : A$ does not affect the value of y, we will still have $\Gamma(y) = C$. Using the typing rule for variables, we get the tree:

$$\frac{\Gamma(y) = C}{\Gamma \vdash y : C}$$

$\Gamma \vdash y : C$ will be the same as $\Gamma \vdash [e/x]y : C$ using the substitution rule for variables, so we have the derivation tree needed. Therefore variables satisfy substitution.

Zero We assume derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash z : \text{Nat}$. As z is a constant, it exists in any context Γ, so we have a tree for $\Gamma \vdash z : \text{Nat}$. This is equal to $\Gamma \vdash [e/x]z : \text{Nat}$, as this is always zero no matter what e and x are. Therefore as we already have the tree for $\Gamma \vdash z : \text{Nat}$, we use it as the derivation tree for $\Gamma \vdash [e/x]z : \text{Nat}$.

Successor We assume derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash s(e') : \text{Nat}$. The second tree is the following:

$$\frac{\Gamma, x : A \vdash e' : \text{Nat}}{\Gamma, x : A \vdash s(e') : \text{Nat}}$$

Therefore we have a subtree $\Gamma, x : A \vdash e' : \text{Nat}$. Using the induction hypothesis on this and $\Gamma \vdash e : A$, we have a derivation tree for $\Gamma \vdash [e/x]e' : \text{Nat}$. Using the typing rule for successor on this gives us the following tree:

$$\frac{\Gamma \vdash [e/x]e' : \text{Nat}}{\Gamma \vdash s([e/x]e') : \text{Nat}}$$

Using the substitution rule for successor, we know the bottom half is equal to $[e/x]s(e')$, so we get the following derivation tree:

$$\frac{\Gamma \vdash [e/x]e' : \text{Nat}}{\Gamma \vdash [e/x]s(e') : \text{Nat}}$$

which is a derivation tree for $\Gamma \vdash [e/x]s(e') : \text{Nat}$. Therefore substitution holds for successor function.

Case We rename x to y using alpha equivalence, then assume derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash \text{case } (e', z \mapsto e_0, s(y) \mapsto e_S)\ : C$. The second derivation tree gives us the subtrees $\Gamma, x : A \vdash e' : \text{Nat}$, $\Gamma, x : A \vdash e_0 : C$ and $\Gamma, y : \text{Nat}, x : A \vdash e_S : C$. Using the induction hypothesis and the tree for $\Gamma \vdash e : A$, we get $\Gamma \vdash [e/x]e' : \text{Nat}$ and $\Gamma \vdash [e/x]e_0 : C$.

For $\Gamma, y : \text{Nat}, x : A \vdash e_S : C$, we need to change the context of e before we can apply the inductive hypothesis. We do this using our Weakening Lemma we just proved (in Section 5.1.1), which gives us $\Gamma, y : \text{Nat} \vdash e : A$. Now we apply the inductive hypothesis with this to get $\Gamma, y : \text{Nat} \vdash [e/x]e_S : C$.

Now we can apply the typing rule to these trees to get the following derivation tree:

$$\frac{\Gamma \vdash [e/x]e' : \text{Nat} \qquad \Gamma \vdash [e/x]e_0 : C \qquad \Gamma, y : \text{Nat} \vdash [e/x]e_S : C}{\Gamma \vdash \text{ case } ([e/x]e', z \mapsto [e/x]e_0, s(y) \mapsto [e/x]e_S) \ : C}$$

By the substitution rule for case, we can replace the bottom half of the tree with $\Gamma \vdash [e/x] \text{ case } (e', z \mapsto e_0, s(y) \mapsto e_S) \ : C$. Therefore substitution holds for the case statement.

Application We assume derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash e_0\ e_1 : C$. The second tree is the following:

$$\frac{\Gamma, x : A \vdash e_0 : B \to C \qquad \Gamma, x : A \vdash e_1 : B}{\Gamma, x : A \vdash e_0\ e_1 : C}$$

which contains the subtrees $\Gamma, x : A \vdash e_0 : B \to C$ and $\Gamma, x : A \vdash e_1 : B$. Combining each of these trees with $\Gamma \vdash e : A$ and the inductive hypothesis gives us $\Gamma \vdash [e/x]e_0 : B \to C$ and $\Gamma \vdash [e/x]e_1 : B$. Using the typing rule for function application with these trees gives us a derivation tree for $\Gamma \vdash [e/x]e_0([e/x]e_1) : C$. This is equal to $\Gamma \vdash [e/x](e_0\ e_1) : C$, so we have the derivation tree for this judgement.

Therefore substitution holds for function application.

λ-Abstraction We rename x to y using α equivalence. Then derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash \lambda y : B.\ e' : B \to C$. For the second tree we have the following:

$$\frac{\Gamma, x : A, y : B \vdash e' : C}{\Gamma, x : A \vdash \lambda y : B.\ e' : B \to C}$$

which gives us the subtree $\Gamma, x : A, y : B \vdash e' : C$. Then we use weakening on $\Gamma \vdash e : A$ to get $\Gamma, y : B \vdash e : A$, which when used with the inductive hypothesis gives us $\Gamma, y : B \vdash [e/x]e' : C$.

Applying the typing rule for λ abstraction to this gives us the following tree:

$$\frac{\Gamma, y : B \vdash [e/x]e' : C}{\Gamma \vdash \lambda y : B.\ [e/x]e' : B \to C}$$

.

Now there are two cases:

1. When $y \notin FV(e')$, the substitution rule gives us $\lambda y : B.\ [e/x]e' = [e/x]\lambda y : B.\ e'$, so we have a derivation tree for $\Gamma \vdash [e/x]\lambda y : B.\ e' : B \to C$

2. When $y \in FV(e')$, rewrite $\lambda y : B.\ [e/x]e' : B \to C$ as $\lambda z : B.[e/x]([z/y]e')$. Then this is the same as $[e/x]\lambda y : B.\ e'$, so we have a derivation tree for $\Gamma \vdash [e/x]\lambda y : B.\ e' : B \to C$

Fixpoint We rename x to y using alpha equivalence, then assume derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash \text{fix}\ \ y : C.\ e' : C$. The second tree is the following:

$$\frac{\Gamma, y : C, x : A \vdash e' : C}{\Gamma, x : A \vdash \text{fix}\ \ y : C.e' : C}$$

giving us the subtree $\Gamma, y : C, x : A \vdash e' : C$. Then we use weakening on $\Gamma \vdash e : A$ to get $\Gamma, y : C \vdash e : A$, which when used with the inductive hypothesis gives us $\Gamma, y : C \vdash [e/x]e' : C$.

Applying the typing rule for fixpoint to this gives us the following derivation tree:

$$\frac{\Gamma, y : C \vdash [e/x]e' : C}{\Gamma \vdash \text{fix}\ \ y : C.[e/x]e' : C}$$

Now there are two cases:

1. When $y \notin FV(e')$, the substitution rule for fixpoint gives us $\Gamma \vdash \text{fix}\ \ y : C.[e/x]e' : C = \Gamma \vdash [e/x](\text{fix}\ \ y : C.e') : C$, so we have the required derivation tree and substitution holds for fixpoint.

2. When $y \in FV(e')$, rewrite $\text{fix}\ \ y : C.\ [e/x]e' : C$ as $\text{fix}\ \ z : C.[e/x]([z/y]e')$. Then this is the same as $[e/x]\,\text{fix}\ \ y : C.\ e'$, so we have a derivation tree for $\Gamma \vdash [e/x]\,\text{fix}\ \ y : C.\ e' : C$

Now we have proved substitution holds for derivation trees of any expression e'. □

5.2 Type Safety

Now we can prove the two lemmas that form the property of Type Safety.

5.2.1 Type Preservation

Type Preservation says that if an expression e of type A is well typed in a context Γ, and it evaluates in one step to e', then e' will also have type A in Γ (i.e. we get the same type at the end of the evaluation as we had at the start):

Theorem 5. *If $\Gamma \vdash e : A$ and $e \mapsto e'$, then $\Gamma \vdash e' : A$*

Proof. We can prove this by induction on derivation trees.

There will be a derivation tree for each rule in the Operational Semantics, so we can check this statement for every evaluation rule on every possible expression:

Variables There are no rules in the Operational Semantics for when e is just a variable so we do nothing here.

Zero Same as for variables.

Successor We assume $\Gamma \vdash s(e) : \text{Nat}$, so we have the following tree (by the typing rule of successor):

$$\frac{\Gamma \vdash e : \text{Nat}}{\Gamma \vdash s(e) : \text{Nat}}$$

Then we assume $s(e) \mapsto s(e')$, so we have the following tree, using the congruence rule for successor:

$$\frac{e \mapsto e'}{s(e) \mapsto s(e')}$$

From these two trees, we get the subtrees $\Gamma \vdash e : \text{Nat}$ and $e \mapsto e'$. Using the inductive hypothesis of type preservation we get a tree for $\Gamma \vdash e' : \text{Nat}$. Then using the typing rule for successor with this we get a tree for $\Gamma \vdash s(e') : \text{Nat}$.

There are no other rules when the expression is $s(e)$, so type preservation holds for successor expressions.

Case There are three evaluation rules, which depend on the expression e being checked:

1. If e can be reduced, then we have the following tree from the typing rule for case:

$$\frac{\Gamma \vdash e : \text{Nat} \qquad \Gamma \vdash e_0 : A \qquad \Gamma, x : \text{Nat} \vdash e_S : A}{\Gamma \vdash \text{ case } (e, z \mapsto e_0, s(x) \mapsto e_S) \ : A}$$

   and our second assumption uses the congruence evaluation rule for case as:

$$\frac{e \mapsto e'}{\text{case } (e, z \mapsto e_0, s(x) \mapsto e_S) \mapsto \text{case } (e', z \mapsto e_0, s(x) \mapsto e_S)}$$

Then we have subtrees for $\Gamma \vdash e : \text{Nat}$, $\Gamma \vdash e_0 : A$, $\Gamma, x : \text{Nat} \vdash e_S : A$ and $e \mapsto e'$.

Using the inductive hypothesis of type preservation, with the trees for $\Gamma \vdash e : \text{Nat}$ and $e \mapsto e'$ we get a tree for $\Gamma \vdash e' : \text{Nat}$. Then we apply the typing rule for case with this, $\Gamma \vdash e_0 : A$ and $\Gamma, x : \text{Nat} \vdash e_S : A$ to get a tree for $\Gamma \vdash \text{case}\ (e', z \mapsto e_0, s(x) \mapsto e_S) : A$

2. If $e = \text{case}\ (z, z \mapsto e_0, s(x) \mapsto e_S)$, then we get the following tree from the typing rule:

$$\frac{\Gamma \vdash z : \text{Nat} \qquad \Gamma \vdash e_0 : A \qquad \Gamma, x : \text{Nat} \vdash e_S : A}{\Gamma \vdash \ \text{case}\ (z, z \mapsto e_0, s(x) \mapsto e_S)\ : A}$$

and we also have the tree for the evaluation rule $\text{case}\ (z, z \mapsto e_0, s(x) \to e_S) \mapsto e_0$. Therefore we need a tree for $\Gamma \vdash e_0 : A$, which we already have, as a subtree of the first assumption.

3. If $e = \text{case}\ (s(v), z \mapsto e_0, s(x) \mapsto e_S)$, then the tree formed from its typing rule is:

$$\frac{\dfrac{\Gamma \vdash v : \text{Nat}}{\Gamma \vdash s(v) : \text{Nat}} \qquad \Gamma \vdash e_0 : A \qquad \Gamma, x : \text{Nat} \vdash e_S : A}{\Gamma \vdash \ \text{case}\ (s(v), z \mapsto e_0, s(x) \mapsto e_S)\ : A}$$

and we also have is the tree for the evaluation rule: $\text{case}\ (s(v), z \mapsto e_0, s(x) \mapsto e_S) \mapsto [v/x]e_S$.

We get the tree for $\Gamma \vdash [v/x]e_S : A$ by using the substitution lemma, with the subtrees for $\Gamma \vdash v : \text{Nat}$ and $\Gamma, x : \text{Nat} \vdash e_S : A$ as parameters.

Application There are two evaluation rules for function application:

1. When e is a function that can be reduced further, we have the following tree from its typing rule:

$$\frac{\Gamma \vdash e_0 : A \to B \qquad \Gamma \vdash e_1 : A}{\Gamma \vdash e_0\ e_1 : B}$$

and the following tree obtained from its evaluation rule:

$$\frac{e_0 \mapsto e_0'}{e_0\ e_1 \mapsto e_0'\ e_1}$$

We use the induction hypothesis with the subtrees for $\Gamma \vdash e_0 : A \to B$ and $e_0 \mapsto e_0'$ to get a subtree for $\Gamma \vdash e_0' : A \to B$. Then using this and the subtree for $\Gamma \vdash e_1 : B$, in the typing rule for function application, we get a subtree for $\Gamma \vdash e_0'\ e_1 : B$.

2. When $e = (\lambda x : A.\ e)\ e'$, we have the following tree from its typing rule:

$$\dfrac{\dfrac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x : A.\ e) : A \to B} \qquad \Gamma \vdash e' : A}{\Gamma \vdash (\lambda x : A.\ e)\ e' : B}$$

   And the tree for $(\lambda x : A.\ e)\ e' \mapsto [e'/x]e$ from its evaluation rule. We have subtrees $\Gamma\ \vdash e' : A$ and $\Gamma, x : A \vdash e : B$, so we use these as parameters to the Substitution Lemma to get the tree for $\Gamma \vdash [e'/x]e : B$.

λ-Abstraction has no evaluation rules when taken as a single expression (because it is a value).

Fixpoint When $e = \text{fix}\ \ x : A.\ e$, we have the following tree from its typing rule:

$$\dfrac{\Gamma, x : A \vdash e : A}{\Gamma \vdash\ \text{fix}\ \ x : A.\ e : A}$$

and the tree for fix $x : A.\ e \mapsto [\text{fix}\ \ x : A.\ e/x]e$ from its evaluation rule. We already have the tree for $\Gamma \vdash\ \text{fix}\ \ x : A.\ e : A$ as an assumption and $\Gamma, x : A \vdash e : A$ is a subtree of it, so we can use the substitution lemma with these parameters to get a tree for $\Gamma \vdash [\text{fix}\ \ x : A.\ e/x]e : A$

Now we have proved type preservation for all the rules in the Operational Semantics on all possible expressions. $\square$

5.2.2 Type Progress

Type Progress says that if an expression e of type A is well typed in a context Γ, then it must evaluate to another expression e' in one step, or be a value (i.e. e cannot be evaluated further), where possible values are

$$v ::= z \mid s(v) \mid \lambda x : A.\ e$$

which are numbers or non-recursive functions (but note this is not saying the values terminate, or that they have normal form):

Theorem 6. *If $\vdash e : A$ then $e \mapsto e'$ or e is a value.*

Proof. We can prove this by induction on derivation trees of e. When we have a derivation tree for a closed term e, we either get a derivation tree for evaluating it in one step to another expression, or e is a value (so there is no new derivation tree).

Zero z is a value

Variables There are no closed terms that are variables, so this is vacuously true. This is because the context of a term is a set of $(variable, type)$ pairs, so when it is empty, there are no variables. Therefore we have no derivation tree for $\vdash x : A$, where x is a variable, so there is no case for variables.

Successor When we have an expression $s(e)$, we assume $\vdash s(e) : \text{Nat}$, giving us the following tree:

$$\frac{\vdash e : \text{Nat}}{\vdash s(e) : \text{Nat}}$$

so we have a subtree for $\vdash e : \text{Nat}$. We can use induction on this subtree to get $e \mapsto e'$ or e is a value. We then assume either side of this:

1. When $e \mapsto e'$ we use the congruence rule for successor to get a tree for $s(e) \mapsto s(e')$. Therefore $s(e) \mapsto s(e')$ or $s(e)$ is a value
2. When e is a value, v, we rewrite $s(e)$ to $s(v)$ This is a value, so $s(e) \mapsto s(e')$ or $s(e)$ is a value is true

Case When we have an expression case $(e, z \mapsto e_0, s(x) \mapsto e_S)$, we assume $\vdash$ case $(e, z \mapsto e_0, s(x) \mapsto e_S) : A$, giving us the following tree:

$$\frac{\vdash e : \text{Nat} \qquad \vdash e_0 : A \qquad x : \text{Nat} \vdash e_S : A}{\vdash \ \text{case } (e, z \mapsto e_0, s(x) \mapsto e_S) \ : A}$$

so we have subtrees for $\vdash e : \text{Nat}$, $\vdash e_0 : A$ and $x : \text{Nat} \vdash e_S : A$.

By induction on $\vdash e : \text{Nat}$, we know that $e \mapsto e'$ or e is a value. We can assume either side of this:

1. When $e \mapsto e'$, we use the congruence rule for case to get case $(e', z \mapsto e_0, s(x) \mapsto e_S) : A$. Therefore we know our original expression maps to some other expression.
2. When e is a value there are two cases.:

(a) $e = z$. The evaluation rule for this has no assumption, so we already have a tree that maps case $(z, z \mapsto e_0, s(x) \mapsto e_S)$ to another expression, which is e_0.

(b) $e = s(v)$. The evaluation rule for this also has no assumption, so we already have a tree that maps case $(s(v), z \mapsto e_0, s(x) \mapsto e_S)$ to another expression, which is $[v/x]e_S$

Therefore, no matter what e is in the case expression, it always maps to another expression, e', so case $(e, z \mapsto e_0, s(x) \to e_S) \mapsto e'$ or case $(e, z \to e_0, s(x) \to e_S)$ is a value is true.

Application When we have an expression $e_0\ e_1$, we assume $\vdash e_0\ e_1 : B$ giving us the following tree:

$$\frac{\vdash e_0 : A \to B \qquad \vdash e_1 : A}{\vdash e_0\ e_1 : B}$$

so we have a subtree for $\vdash e_0 : A \to B$. We can use the inductive hypothesis on this to get $e_0 \mapsto e_0'$ or e_0 is a value. We can assume either side of this:

1. When $e_0 \mapsto e_0'$, we use the congruence rule for application to get a tree for $e_0\ e_1 \mapsto e_0'\ e_1$
2. When e_0 is a value it must be $\lambda x : A.\ e_0$, as the other values are not function types. The evaluation rule for $(\lambda x : A.\ e_0)\ e_1$ has no assumption, so we already have a tree that maps $(\lambda x : A.\ e_0)\ e_1$ to another expression, which is $[e_1/x]e_0$

Therefore, for every possible value of $e_0\ e_1$, we can evaluate it to another expression in one step, so $e_0\ e_1 \mapsto e_0'\ e_1$ or $e_0\ e_1$ is a value is true.

λ-Abstraction $\lambda x : A.\ e$ is always a value, for any $x : A$ and $e : A$.

Fixpoint When we have an expression fix $x : A.\ e$, we assume $\vdash$ fix $x : A.\ e : A$, giving us the following tree:

$$\frac{x : A \vdash e : A}{\vdash \text{ fix } x : A.\ e : A}$$

There are no assumptions in the evaluation rule for fixpoint, so we already have the tree that maps fix $x : A.\ e$ to another expression, which is $[\text{fix } x : A.\ e/x]e$. Therefore we know that fix $x : A.\ e$ always maps to another expression.

Now we have proved Type Progress for all possible expressions. $\square$

Type Safety can be assumed from the the two lemmas we have just proved. This approach was first used in [Wright and Felleisen, 1994].

The theorem for Type Safety says that a closed term either evaluates to a value in a finite number of steps or loops forever:

Theorem 7. *If* $\vdash e : A$ *then* $e \mapsto^* v$ *(where* $\vdash v : A$*) or* $e \mapsto^\infty$

Chapter 6

Denotational Semantics

Denotational Semantics describe expressions in a programming language as functions in a mathematical model, as explained in Chapter 1. Now we use the Domain Theory we discussed in Section 2.1 to create that model for PCF:

6.1 Denotational Model of PCF

6.1.1 Denotation of Types

Our Denotational Semantics maps the types of PCF to a domain representing that type. We define a function:

$$[\![-]\!] : \mathit{Type} \to \mathit{Domain}$$

that maps a type to a domain. We defined our types by induction, so we must specify the different constructions of domains we use by induction on types:

1. The type of natural numbers is the base type, so they are modelled by a single domain. We use the flat domain of natural numbers, described in Section 2.2.2, where $\perp$ represents a term that does not terminate:

$$[\![\mathrm{Nat}]\!] = \mathbb{N}_\perp$$

2. Function types are formed of other types. We model them using the domain of continuous functions, described in Section 2.3.1.

$$[\![A \to B]\!] = [\![A]\!] \to [\![B]\!]$$

(Where $[\![A]\!] \to [\![B]\!]$ is the same as $\text{Cont}(A, B)$)

6.1.2 Denotation of Typing Contexts

We also need a domain for the typing contexts, which is given by the following function:

$$[\![-]\!]_{Ctx} : Context \to Domain$$

that maps a typing context to a domain. The domain will be a nested tuple, the size of which depends on the number of variables in Γ. Each variable's type is a domain, so the overall domain is a product of domains, described in Section 2.2.3. Therefore we define these domains by induction on the size of the given typing context:

The empty context is given by

$$[\![\cdot]\!]_{Ctx} = \mathbb{1}$$

the single element domain, which we defined in Section 2.2.1.

Adding a variable to a context Γ gives us the following domain:

$$[\![\Gamma, x : A]\!]_{Ctx} = [\![\Gamma]\!] \times [\![A]\!]$$

where $[\![\Gamma]\!]$ is a product of domains.

This gives us all combinations of all possible values of each variable in Γ. If we want a specific valuation of the variables, we can refer to $d^* \in [\![\Gamma]\!]_{Ctx}$. $d^* = (d_1, \ldots, d_n)$ will be a tuple in the underlying set of the product domain of size n, where n is the number of domains in the product domain and also the number or variables in Γ.

6.1.3 Denotation of well typed terms

We map well-typed PCF expressions to the domain that models their type. Given a well typed term $\Gamma \vdash e : A$ we have the continuous function:

$$[\![\Gamma \vdash e : A]\!] \in [\![\Gamma]\!]_{Ctx} \to [\![A]\!]$$

So $[\![\Gamma \vdash e : A]\!]d^*$ gives us an element of $[\![A]\!]$, the domain that models e's type. We define this function by induction on each possible value of e:

Variables Given a context $\Gamma = x_0 : A_0, \ \dots \ , x_n : A_n$, $[\![\Gamma]\!]_{Ctx}$ maps a tuple d^* in $[\![A_0]\!] \times \cdots \times [\![A_n]\!]$ to a value in $[\![A_i]\!]$:

$$[\![\Gamma \vdash x_i : A_i]\!] = \lambda d^* \in [\![\Gamma]\!].\ \pi_i(d^*)$$

We use the ith projection function to get the value of the ith variable in the context.

Zero z has the type Nat, the domain of which we have defined to be $\mathbb{N}_\perp$. As z is a constant, we always map it to the same value, which is 0, no matter what d^* is:

$$[\![\Gamma \vdash z : Nat]\!]d^* = 0$$

Successor When $\Gamma \vdash s(e) : \text{Nat}$ is a well typed term, $\Gamma \vdash e : \text{Nat}$ is too, so we can use $[\![\Gamma \vdash e : \text{Nat}]\!]$ in the definition of the denotational semantics for successor. As the domain of e is $\mathbb{N}_\perp$, we must consider the case where e maps to $\perp$, for which we would also have to map $s(e)$ to $\perp$:

$$[\![\Gamma \vdash s(e) : \text{Nat}]\!]d^* = \text{Let } v = [\![\Gamma \vdash e : \text{Nat}]\!]d^* \text{ in}$$
$$\begin{cases} v+1 & \text{if } v \neq \perp \\ \perp & \text{if } v = \perp \end{cases}$$

Case When $\Gamma \vdash \text{case } (e, z \mapsto e_0, s(y) \mapsto e_S) : C$ is a well typed term, $\Gamma \vdash e : \text{Nat}$ is too, so we can use $[\![\Gamma \vdash e : \text{Nat}]\!]$ in the definition of the denotational semantics for case:

$[\![\Gamma \vdash \text{case } (e, z \mapsto e_0, s(y) \mapsto e_S) : C]\!] d^* = \text{Let } v = [\![\Gamma \vdash e : \text{Nat}]\!] d^*$
in

$$\begin{cases} [\![\Gamma \vdash e_0 : C]\!] d^* & \text{if } v = 0 \\ [\![\Gamma, y : \text{Nat} \vdash e_S : C]\!] (d^*, n) & \text{if } v = n + 1 \\ \bot & \text{if } v = \bot \end{cases}$$

Application In this rule we already have a denotation for the function and for the element we are applying it to. The bottom element of our domain of functions is the function that loops on all inputs, $\bot = \lambda x \in X.\bot(x)$. Therefore the value of f will always be a function. Functions on domains can be applied to bottom elements, so we can still have $f(v)$ when $v = \bot$. Therefore there is only one case for function application:

$$[\![\Gamma \vdash e\ e' : B]\!] d^* = \text{Let } f = [\![\Gamma \vdash e : A \to B]\!] d^* \text{ in}$$
$$\text{Let } v = [\![\Gamma \vdash e' : A]\!] d^*$$
$$\text{in } f(v)$$

λ-Abstraction For λ abstraction, by its typing rule, we already have a denotation for $[\![\Gamma, x : A \vdash e : B]\!] d^*$. This is a function of type $[\![\Gamma]\!] \times [\![A]\!] \to [\![B]\!]$. The function we want to obtain is of type $[\![\Gamma]\!] \to ([\![A \to B]\!])$, so we must return a continuous function. We use currying, with our denotation of $\Gamma, x : A \vdash e : B$. As this is in a different context, we need our function to be in a context where the value of x is our $a \in [\![A]\!]$ that is in the argument to our function, which is (d^*, a) :

$$[\![\Gamma \vdash \lambda x : A.e : A \to B]\!] d^* = \lambda a \in [\![A]\!].[\![\Gamma, x : A \vdash e : B]\!] (d^*, a)$$

Fixpoint For fixpoint, by its typing rule we already have a denotation for $[\![\Gamma, x : A \vdash e : A]\!] d^*$ This is a function of type $[\![\Gamma]\!] \times [\![A]\!] \to [\![A]\!]$. The function we want to obtain is of type $[\![\Gamma]\!] \to [\![A]\!]$. To get an element of $[\![A]\!]$, we use the fixpoint function, $\mathsf{fix}_{[\![A]\!]}$, which is a continuous function of type $([\![A]\!] \to [\![A]\!]) \to [\![A]\!]$. The function we give to the fixpoint is the one that maps any given $a \in [\![A]\!]$ to the denotation of $\Gamma, x : A \vdash e : A$ in a context where a is the value of x:

$$[\![\Gamma \vdash \text{fix } x : A.e : A]\!]d^* = \mathsf{fix}_{[\![A]\!]}(\lambda a \in [\![A]\!].[\![\Gamma, x : A \vdash e : A]\!](d^*, a))$$

6.2 Substitution Theorem

The following theorem says that given a well typed expression $e : A$ and another expression $e' : C$, which is well typed in the context with $x : A$ added, then the denotation of e' with e substituted for x is the same as the denotation of the original expression in the context with $x : A$ added and valuation with the denotation of e as the value of x:

Theorem 8. *If* $\Gamma \vdash e : A$ *and* $\Gamma, x : A \vdash e' : C$ *and* $d^* \in [\![\Gamma]\!]$*, then* $[\![\Gamma \vdash [e/x]e' : C]\!]d^* = [\![\Gamma, x : A \vdash e' : C]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$

Proof. We prove this by induction on the value of e':

Variables There are two cases for variables:

1. For a variable $x : C$, C must be equal to A, so we get $[\![\Gamma \vdash [e/x]x : A]\!]d^* = [\![\Gamma \vdash e : A]\!]d^*$, from the substitution rule.

   On the right hand side, $[\![\Gamma, x : A \vdash x : A]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*) = \pi_i(d^*, [\![\Gamma \vdash e : A]\!]d^*)$. The value of this is the value of x, which is $[\![\Gamma \vdash e : A]\!]d^*$.

   Therefore $[\![\Gamma \vdash [e/x]x : A]\!]d^* = [\![\Gamma, x : A \vdash x : A]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*) = [\![\Gamma \vdash e : A]\!]d^*$

2. For a variable $y : C$, we have $[\![\Gamma \vdash [e/x]y : C]\!]d^* = [\![\Gamma \vdash y : C]\!]d^*$, by the substitution rule for variables. This is equal to $\pi_i(d^*)$, where $y : C$ is the ith element of Γ. If we extend the context Γ with $x : A$ and the valuation d^* with $[\![\Gamma \vdash e : A]\!]d^*$, then this does not affect $\pi_i(d^*)$, as each variable is independent. Therefore $[\![\Gamma \vdash [e/x]y : C]\!]d^* = [\![\Gamma, x : A \vdash y : C]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$.

Zero By the substitution rule for zero, $[\![\Gamma \vdash [e/x]z : \text{Nat}]\!]d^* = [\![\Gamma \vdash z : \text{Nat}]\!]d^*$. As z is a constant, its denotation will be the same for any Γ and d^*, so we always get 0. Therefore $[\![\Gamma \vdash [e/x]z : \text{Nat}]\!]d^* = [\![\Gamma, x : A \vdash z : \text{Nat}]\!](d^*, [\![\Gamma, \vdash e : A]\!]d^*) = 0$.

Successor Using the substitution rule, $[\![\Gamma \vdash [e/x]s(e') : \text{Nat}]\!]d^* = [\![\Gamma \vdash s([e/x]e') : \text{Nat}]\!]d^*$. The induction hypothesis is $[\![\Gamma \vdash [e/x]e' : C]\!]d^* = [\![\Gamma, x : A \vdash e' : C]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$, so we can use this to rewrite $[\![\Gamma \vdash s([e/x]e') : \text{Nat}]\!]d^*$ as the following function:

Let $v = [\![\Gamma, x : A \vdash e' : C]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$ in

$$\begin{cases} v+1 & \text{if } v \neq \bot \\ \bot & \text{if } v = \bot \end{cases}$$

This function is also the definition of $[\![\Gamma, x : A \vdash s(e') : C]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$.

Therefore $[\![\Gamma \vdash [e/x]s(e') : \text{Nat}]\!]d^* = [\![\Gamma, x : A \vdash s(e') : C]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$

Case Using the substitution rule for case, $[\![\Gamma \vdash [e/x](\text{case } (e', z \mapsto e_0, s(y) \mapsto e_S) : C]\!]d^* = [\![\Gamma \vdash (\text{case } ([e/x]e', z \mapsto [e/x]e_0, s(y) \mapsto [e/x]e_S) : C]\!]d^*$. We can use induction on all the expressions with substitutions to get the following definition of $[\![\Gamma \vdash (\text{case } ([e/x]e', z \mapsto [e/x]e_0, s(y) \mapsto [e/x]e_S) : C]\!]d^*$:

Let $v = [\![\Gamma, x : A \vdash e' : \text{Nat}]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$ in

$$\begin{cases} [\![\Gamma, x : A \vdash e_0 : C]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*) & \text{if } v = 0 \\ [\![\Gamma, y : \text{Nat}, x : A \vdash e_S : C]\!](d^*, n, [\![\Gamma \vdash e : A]\!]d^*) & \text{if } v = n+1 \\ \bot & \text{if } v = \bot \end{cases}$$

This function is also the definition of $[\![\Gamma, x : A \vdash [e/x](\text{case } (e', z \mapsto e_0, s(v) \mapsto e_S)) : C]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$

Therefore $[\![\Gamma \vdash [e/x](\text{case } (e', z \mapsto e_0, s(v) \mapsto e_S) : C]\!]d^* = [\![\Gamma, x : A \vdash \text{case } (e', z \mapsto e_0, s(v) \mapsto e_S) : C]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$

Application Using the substitution rule for application, $[\![\Gamma \vdash [e/x](e_0\ e_1) : B]\!]d^* = [\![\Gamma \vdash [e/x]e_0([e/x]e_1) : B]\!]d^*$. We can use induction on $[\![\Gamma \vdash [e/x]e_0 : A \to B]\!]$ and $[\![\Gamma \vdash [e/x]e_1 : A]\!]$ to rewrite the denotation as the following:

Let $f = [\![\Gamma, x : A \vdash e_0 : A \to B]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$ in

$$\text{Let } v = [\![\Gamma, x : A \vdash e_1 : A]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$$

$$\text{in } f(v)$$

This function is also the definition of $[\![\Gamma, x : A \vdash e_0\ e_1 : B]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$.

Therefore, $[\![\Gamma \vdash [e/x](e_0\ e_1) : B]\!]d^* = [\![\Gamma, x : A \vdash e_0\ e_1 : B]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$

λ-Abstraction Using the substitution rule we have $[\![\Gamma \vdash [e/x](\lambda y : A.e') : A \to B]\!]d^* = [\![\Gamma \vdash \lambda y : A.([e/x]e') : A \to B]\!]d^*$. We can use induction on $[\![\Gamma, y : A \vdash [e/x]e' : B]\!]$, to rewrite the denotation as the following:

$$\lambda a \in [\![A]\!].[\![\Gamma, y : A, x : A \vdash e : B]\!](d^*, a, [\![\Gamma \vdash e : A]\!]d^*)$$

This is also the definition of $[\![\Gamma, x : A \vdash \lambda y : A.\ e' : A \to B]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$

Therefore $[\![\Gamma \vdash [e/x](\lambda y : A.e') : A \to B]\!]d^* = [\![\Gamma, x : A \vdash \lambda y : A.\ e' : A \to B]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*)$

Fixpoint Using the substitution rule for fixpoint, $[\![\Gamma \vdash [e/x](\text{fix}\ \ y : C.e' : C)]\!]d^* = [\![\Gamma \vdash \text{fix}\ \ y : C.[e/x]e' : C]\!]d^*$. The denotation of this is the following:

$$\mathsf{fix}_{[\![C]\!]}(\lambda c \in [\![C]\!].[\![\Gamma, y : C \vdash [e/x]e' : C]\!](d^*, c)$$

We can use induction on $[\![\Gamma, y : C \vdash [e/x]e' : C]\!]$ to rewrite the denotation as the following:

$$\mathsf{fix}_{[\![C]\!]}(\lambda c \in [\![C]\!].[\![\Gamma, y : C, x : A \vdash e' : C]\!](d^*, c, [\![\Gamma \vdash e : A]\!]d^*)$$

This is also the definition of $[\![\Gamma, x : A \vdash (\text{fix}\ \ y : C.e' : C)]\!](d^*, [\![\Gamma \vdash e : C]\!]d^*)$.

Therefore $[\![\Gamma \vdash [e/x](\text{fix}\ \ y : C.\ e' : C)]\!]d^* = [\![\Gamma, x : A \vdash (\text{fix}\ \ y : C.e' : C)]\!](d^*, [\![\Gamma \vdash e : A]\!]d^*/x)$.

Now we have proved the theorem for every case of e'. $\square$

6.3 Soundness

In general, Soundness says that if we have a mapping from one expression to another in the Operational Semantics, then these expressions must also have equal denotations in the Denotational Semantics (so the Operational Semantics are sound with relation to the Denotational Semantics).

Our Soundness theorem is the following theorem, which says that for a well typed expression e, if it maps to another expression e', then its denotation will be equal to that of the new expression in the same context:

Theorem 9. *If* $\Gamma \vdash e : A$ *and* $e \mapsto e'$ *and* $d^* \in [\![\Gamma]\!]$*, then* $[\![\Gamma \vdash e : A]\!]d^* = [\![\Gamma \vdash e' : A]\!]d^*$

Proof. By induction on $e \mapsto e'$, so there is a case for each evaluation rule:

Variables have no rules in the operational semantics, so there are no cases here.

Zero is a value, so it has no evaluation rules. Therefore there are no cases for zero.

Successor We use a congruence rule for successor, so when $s(e) \mapsto s(e')$ we also know that $e \mapsto e'$. From $\Gamma \vdash s(e) : \text{Nat}$, we know that $\Gamma \vdash e : \text{Nat}$. Therefore we can use induction on this to get $[\![\Gamma \vdash e : A]\!]d^* = [\![\Gamma \vdash e' : A]\!]d^*$.

We can use this to rewrite $[\![\Gamma \vdash s(e) : \text{Nat}]\!]d^*$ as:

Let $v = [\![\Gamma \vdash e' : \text{Nat}]\!]d^*$ in

$$\begin{cases} v+1 & \text{if } v \neq \bot \\ \bot & \text{if } v = \bot \end{cases}$$

Which is the same as $[\![\Gamma \vdash s(e') : \text{Nat}]\!]d^*$

Case There are three cases for case:

1. When e is an expression that can be reduced, we use a congruence rule for case, so when case $(e, z \mapsto e_0, s(y) \mapsto e_S) \mapsto$ case $(e', z \mapsto e_0, s(y) \mapsto e_S)$ we also know that $e \mapsto e'$. From $\Gamma \vdash$ case $(e, z \mapsto e_0, s(y) \mapsto e_S) : C$, we know that $\Gamma \vdash e : \text{Nat}$. Therefore we can use induction to get $[\![\Gamma \vdash e : \text{Nat}]\!]d^* = [\![\Gamma \vdash e' : \text{Nat}]\!]d^*$.

   We can use this to rewrite $[\![\Gamma \vdash \text{case } (e, z \mapsto e_0, s(y) \mapsto e_S) : C]\!]d^*$ as:

Let $v = [\![\Gamma \vdash e' : \text{Nat}]\!]d^*$ in

$$\begin{cases} [\![\Gamma \vdash e_0 : C]\!]d^* & \text{if } v = 0 \\ [\![\Gamma, y : \text{Nat} \vdash e_S : C]\!](d^*, n) & \text{if } v = n + 1 \\ \bot & \text{if } v = \bot \end{cases}$$

Which is the same as $[\![\Gamma \vdash \text{case } (e', z \mapsto e_0, s(y) \mapsto e_S) : C]\!]d^*$.

2. When $e = z$, we have $[\![\Gamma \vdash \text{case}(z, z \mapsto e_0, s(y) \mapsto e_S) : C]\!]d^*$ which is:

Let $v = [\![\Gamma \vdash z : \text{Nat}]\!]d^*$ in

$$\begin{cases} [\![\Gamma \vdash e_0 : C]\!]d^* & \text{if } v = 0 \\ [\![\Gamma, y : \text{Nat} \vdash e_S : C]\!](d^*, n) & \text{if } v = n + 1 \\ \bot & \text{if } v = \bot \end{cases}$$

As $[\![\Gamma \vdash z : \text{Nat}]\!]d^*$ is always 0, this can be simplified to $[\![\Gamma \vdash e_0 : C]\!]d^*$, which is the result of the evaluation rule.

3. When $e = s(v)$, we have $[\![\Gamma \vdash \text{case}(s(v), z \mapsto e_0, s(y) \mapsto e_S) : C]\!]d^*$ which is:

Let $v' = [\![\Gamma \vdash s(v) : \text{Nat}]\!]d^*$ in

$$\begin{cases} [\![\Gamma \vdash e_0 : C]\!]d^* & \text{if } v' = 0 \\ [\![\Gamma, y : \text{Nat} \vdash e_S : C]\!](d^*, n) & \text{if } v' = n + 1 \\ \bot & \text{if } v' = \bot \end{cases}$$

where $n = [\![\Gamma \vdash v : Nat]\!]d^*$.

There are two possibilities for the value of v'.

(a) If $v' = \bot$ then the function will return $\bot$

(b) Otherwise $v' = n + 1$, where $n = [\![\Gamma \vdash v : \text{Nat}]\!]d^*$. With this we can simplify the definition of the expression to $[\![\Gamma, y : \text{Nat} \vdash e_S : C]\!](d^*, n)$

This is the same as:

$$[\![\Gamma, y : Nat \vdash e_S : C]\!](d^*, [\![\Gamma \vdash v : Nat]\!]d^*)$$

Using the substitution lemma, we get $[\![\Gamma \vdash [v/y]e_S]\!]d^*$.

Application There are two cases for application:

1. We use a congruence rule for function application, so when $e_0\ e_1 \mapsto e_0'\ e_1$ we also know that $e \mapsto e'$. From $\Gamma \vdash e_0\ e_1 : B$, we know that $\Gamma \vdash e_0 : A \to B$. Therefore we can use induction on this to get $[\![\Gamma \vdash e_0\ e_1 : A]\!]d^* = [\![\Gamma \vdash e_0'\ e_1 : A]\!]d^*$.

   We can use this to rewrite $[\![\Gamma \vdash e_0\ e_1 : B]\!]d^*$ as:

   Let $f = [\![\Gamma \vdash e_0' : A \to B]\!]d^*$ in

   Let $v = [\![\Gamma \vdash e_1 : A]\!]d^*$

   in $f(v)$

   which is the same as $[\![\Gamma \vdash e_0'\ e_1 : B]\!]d^*$

2. When $e = \lambda x : A.\ e$, it is a value, so cannot be reduced further by the congruence rule. We use the following rule in the Operational Semantics:

$$\frac{}{(\lambda x : A.\ e)\ e' \mapsto [e'/x]e}$$

   so we need a denotation $[\![\Gamma \vdash [e'/x]e]\!]d^*$.

   As we have the denotation of $[\![\Gamma \vdash (\lambda x : A.\ e)\ e' : B]\!]d^*$, we have $f = [\![\Gamma \vdash \lambda x : A.e : A \to B]\!]d^*$ and $v = [\![\Gamma \vdash e' : A]\!]d^*$.

   $f = \lambda a \in [\![A]\!].[\![\Gamma, x : A \vdash e : A]\!](d^*, a)$, so $f\ v$ is $[\![\Gamma, x : A \vdash e : A]\!](d^*, [\![\Gamma \vdash e' : A]\!]d^*)$

   By the Substitution Theorem (see Theorem 8) , this is the same as $[\![\Gamma \vdash [e'/x]e]\!]d^*$

λ-Abstraction is a value, so has no evaluation rules. Therefore there are no cases.

Fixpoint As $[\![\Gamma \vdash \text{fix } x : A.\ e : A]\!]d^*$ is a fixpoint operator and we know that $f(\mathsf{fix}(f)) = \mathsf{fix}(f)$, we can rewrite it as:

$$(\lambda a \in [\![A]\!].[\![\Gamma, x : A \vdash e : A]\!](d^*, a))[\mathsf{fix}_{[\![A]\!]}(\lambda a \in [\![A]\!].[\![\Gamma, x : A \vdash e : A]\!](d^*, a))]$$

which is equal to:

$$[\![\Gamma, x : A \vdash e : A]\!](d^*, \mathsf{fix}_{[\![A]\!]}(\lambda a \in [\![A]\!].[\![\Gamma, x : A \vdash e : A]\!](d^*, a)))$$

which is equal to:

$$[\![\Gamma, x : A \vdash e : A]\!](d^*, [\![\Gamma \vdash \text{fix } x : A.e : A]\!]d^*)$$

Using the Substitution Theorem (see Theorem 8), this is the same as

$$[\![\Gamma \vdash [\text{fix } x : A.\ e/x]e : A]\!]d^*$$

The evaluation rule is

$$\frac{}{\text{fix } x : A.\ e \mapsto [\text{fix } x : A.e/x]e}$$

so this is the denotation we need. $\square$

Chapter 7

Adequacy

Now that we have defined our Operational Semantics (in Chapter 4) and our Denotational Semantics (in Chapter 6), we can relate them using the Adequacy Theorem:

Theorem 10. *If* $\vdash e : \text{Nat}$ *(i.e. e is a closed term of type* Nat*), then* $\forall n \in \mathbb{N}.\ [\![e]\!] = n \Leftrightarrow e \mapsto^* \underline{n}$

where $[\![e]\!] = [\![\vdash e : \text{Nat}]\!]$ and $\underline{n}$ represents the numeral n, (as opposed to the actual natural number n).

The prove this theorem we must prove it in two directions. We have mostly proved the backwards direction because of the Soundness proof we proved in the previous chapter (see Section 6.3). Therefore we have proved half of Adequacy.

Therefore, the right to left direction of Adequacy is a corollary of the Soundness proof:

Corollary 1. *If* $\vdash e : \text{Nat}$ *and* $[\![e]\!] = n$ *then* $e \mapsto^* \underline{n} \Rightarrow [\![e]\!] = n$

Proof. We rewrite $e \mapsto^* \underline{n}$ as $e \mapsto e_0 \mapsto \cdots \mapsto e_m \mapsto \underline{n}$, for any $m \geq 0$. Applying Soundness to these evaluations gives us $[\![e]\!] = \cdots = [\![e_m]\!] = [\![\underline{n}]\!]$. Therefore we have $[\![e]\!] = [\![\underline{n}]\!]$.

Now we need to prove $\forall n \in \mathbb{N}. [\![\underline{n}]\!] = n$, which we prove by induction on n:

If $n = 0$, then $[\![\underline{0}]\!] = [\![z]\!] = 0$, by the definition of the Denotational Semantics for zero.

If $n = n + 1$, then $[\![\underline{n+1}]\!] = [\![s(\underline{n})]\!]$. By the inductive hypothesis we have $[\![\underline{n}]\!] = n$, so $[\![s(\underline{n})]\!] = n + 1$. The definition of $[\![s(\underline{n})]\!]$ is $[\![\underline{n}]\!] + 1$. Therefore $[\![\underline{n+1}]\!] = n + 1$.

Therefore $\forall n \in \mathbb{N}.[\![\underline{n}]\!] = n$, so $[\![e]\!] = [\![\underline{n}]\!] = n$.

□

To prove the forwards direction, we could naively try to prove it by induction on each expression. However, although we only consider closed expressions of base type, these expressions may have sub-expressions that are not closed or of type Nat, so we have no inductive hypothesis to provide in these cases.

Therefore, instead we use the Logical Relations approach that we described in Section 2.5.

7.1 Logical Relation

The logical relation we define is a binary logical relation between denotations of PCF expressions and closed expressions of PCF. It is based on a binary logical relation defined in [Streicher, 2006].

There are some differences between our logical relation and the one defined in [Streicher, 2006]. In our proof of the fixpoint case of the Main Lemma, we do not use the observational preorder that they use. Instead we use an Expansion Lemma (see Lemma 7 below). We also work with chains instead of directed sets.

Also, Streicher's logical relation is defined on a version of PCF in which constants replace certain constructors in the language.

For example, in our definition of PCF, we have terms $s(e)$, for some expression e. Here s is a constructor, so s itself is not a term, but once we attach an expression to it, it is a term, $s(e)$.

This is the same for zero, case and fix.

Alternatively, we could have defined these operations as constants. For example, succ : Nat → Nat is a function in the language that has a pre defined definition that never changes. We then also define zero : Nat, case : Nat → Nat → Nat (just on natural numbers) and fix : $(A \to A) \to A$. This is how PCF is defined in [Streicher, 2006].

7.1.1 Definition of the Logical Relation

The definition of the logical relation is the following:

Definition 7. *We create a family of binary relations $R_A \subseteq [\![A]\!] \times \{e \mid\ \vdash e : A\}$, by defining relations by induction on types for each type as follows:*

$$dR_{\text{Nat}}e \Leftrightarrow \forall n \in \mathbb{N}.\ d = n \Rightarrow e \mapsto^* n$$

$$fR_{A\to B}e \Leftrightarrow \forall d \in [\![A]\!].\ \forall e' \in \{e \mid\ \vdash e : A\}.\ dR_A e' \Rightarrow f(d)R_B e\ e'$$

The forwards direction of Adequacy is given by the definition of the relation on expressions of type Nat.

As we previously described in section 2.5, to prove our actual property, we prove a more general Main Lemma on the logical relation first. Our Main Lemma for this relation will be the following:

Lemma 5. *If* $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, $\Gamma \vdash e : A$ *and* $d_1 R_{A_1} e_1,\ d_2 R_{A_2} e_2, \ldots,\ d_n R_{A_n} e_n$, *then*

$$[\![\Gamma \vdash e : A]\!](d_1, \ldots, d_n)\ R_A\ [e_1/x_1, \ldots, e_n/x_n]e$$

In the lemma we are given an expression e that is well typed in a typing context Γ and some denotations d_i that are related to some expressions e_i.

The denotation of that expression applied to substitutions $(d_1, \ldots d_n)$ for all the free variables in Γ will be related to the expression e with the expressions that correspond to each denotation (according to our assumption) substituted for the free variables in e.

We will prove this by induction on each possible expression e.

7.1.2 Substitution Function

In the Main Lemma, we make multiple substitutions in the expression e, but we only defined our substitution function (in Chapter 5) on single substitutions $[e'/x]e$. Therefore we define substituting multiple variables in the following way, where $[\gamma] = [e_1/x_1, \ldots e_n/x_n]$:

$$[\gamma](zero) = zero$$

$$[\gamma](x) = \begin{cases} [\gamma](x) & \text{if } x \in dom(\gamma) \\ x & otherwise \end{cases}$$

This says that if x is present in γ, (there is a substitution for it), replace x with the result of the substitution in γ. Otherwise there is nothing to replace x with, so we return the variable unaltered.

$$[\gamma](s(e)) = s([\gamma](e))$$

$$[\gamma](\text{case}(e, z \mapsto e_0, s(v) \mapsto e_S)) = \text{case}([\gamma](e), z \mapsto [\gamma](e_0), s(v) \mapsto [\gamma](e_S))$$

$$[\gamma](e\ e') = ([\gamma]e)([\gamma]e')$$

$$[\gamma](\lambda x : A.\ e) = \lambda x : A.[\gamma]e$$

$$[\gamma](\text{fix}\ \ x : A.\ e) = \text{fix}\ \ x : A.[\gamma]e$$

For a non empty $\gamma = e_1/x_1, \dots e_n/x_n$, we have:

$$[e_1/x_1, \dots, e_n/x_n]e = [e_1/x_1]([e_2/x_2](\dots [e_n/x_n]e)$$

If we add a variable to the substitution we can define this in the following way:

$$[t/x][\gamma]e = [\gamma, t/x]e$$

7.2 Lemmas for Main Lemma

We are almost ready to prove the Main Lemma, but first we prove some lemmas that we will use in the proof:

7.2.1 Bottom Element Lemma

Lemma 6. *For any type A and $\Gamma \vdash e : A$, $\bot\ R_A\ e$*

Proof. By induction on types. For Nat, we want to show $\forall n \in \mathbb{N}.\ \bot = n \Rightarrow e \mapsto^* n$. Because $\bot \notin \mathbb{N}$, this statement is vacuously true.

For $R_{A\to B}$, we have that $\bot$ is the function $\bot = \lambda x : A.\bot(x)$, so we want to show $\forall d \in [\![A]\!].\ \forall e' \in \{e \mid\ \vdash e : A\}.\ dR_Ae' \Rightarrow (\lambda x : A.\bot(x))(d)R_Be\ e'$. Assume d is an element of the domain representing type A and e' is an expression of type A such that $d\ R_Ae'$. We want to show that $\bot\ R_B\ e\ e'$ and by the inductive hypothesis, we know for any $e : B$ that $\bot\ R_B\ e$, so we can just use this with $e\ e'$ to get our conclusion. □

7.2.2 Expansion Lemma

Lemma 7. *If $\Gamma \vdash e : A$ and $e \mapsto e'$ and dR_Ae' then dR_Ae*

Proof. By induction on types. For base case we assume $dR_{\text{Nat}}e'$, so we have $\forall n \in \mathbb{N}.\ d = n \Rightarrow e' \mapsto^* n$. Let $n = d$. Then, as $e \mapsto e'$ and $e' \mapsto^* n$, we know that $e \mapsto^* n$. Therefore, $dR_{\text{Nat}}e$.

For the inductive case, assume $e_0 \mapsto e_0'$ and $f\ R_{A\to B}\ e_0$, so we have $\forall a \in [\![A]\!].\ \forall e_1 \in \{e \mid\ \vdash e : A\}.\ a\ R_A\ e_1 \Rightarrow f(a)\ R_B\ e_0\ e_1$. Let a be an element in the domain representing A and e_1 be an expression of type A such that aR_Ae_1. Then $f(a)\ R_B\ e_0\ e_1$.

By the inductive hypothesis for expressions of type B and by using the congruence rule with $e_0 \to e_0'$, we have $e_0\ e_1 \mapsto e_0'\ e_1$, so we know both of these expressions are related to the same denotation. Therefore we get $f(a)\ R_B\ e_0'\ e_1$, so we know $\forall a \in [\![A]\!].\ \forall e_1 \in \{e \mid\ \vdash e : A\}.\ a\ R_A\ e_1 \Rightarrow f(a)\ R_B\ e_0'\ e_1$, so $f\ R_{A\to B}\ e_0'$. □

7.2.3 Chains Lemma

This lemma was adapted from a similar lemma on directed sets from [Streicher, 2006].

Lemma 8. *For an expression $e : A$ and a chain $x_0 \sqsubseteq x_1 \sqsubseteq \dots$, if x_iR_Ae, then $\bigsqcup x_n\ R_A\ e$*

Proof. By induction on types. For base type, Nat, we assume we have a chain of elements in $\mathbb{N}_\perp$ such that $x_i\ R_{\text{Nat}}e$. Therefore for any element in the chain we know $\forall n \in \mathbb{N}.\ x_i = n \Rightarrow e \mapsto^* n$. We have two cases depending on the values of $\bigsqcup x_n$:

1. $\bigsqcup x_n = n$. Then we know that $e \mapsto^* \bigsqcup x_n$, as any $x_nR_{\text{Nat}}e$
2. $\bigsqcup x_n = \perp$. By Lemma 6 we know that $\perp R_{\text{Nat}}e$ for any e

The inductive case is for $R_{A\to B}$. Assume we have a chain $f_1 \sqsubseteq f_2 \sqsubseteq \dots$ of elements in $[\![A]\!] \to [\![B]\!]$ and $d\ R_A\ e'$ for some $d \in [\![A]\!]$ and $e : A$. Then we need to show $\bigsqcup f_n(d)\ R_B\ e\ e'$. By induction we know for any expression of type B and a chain $f_1(d) \sqsubseteq f_2(d) \sqsubseteq \dots$ of elements of $[\![B]\!]$, $\bigsqcup f_n(d)$ is related to that expression. Therefore we have $\bigsqcup f_n(d)\ R_B\ e\ e'$. □

7.3 Main Lemma

Finally, we can prove the Main Lemma (Lemma 5 above):

Proof. By induction on the possible values of e.

Variables For a variable $x_1 : A_1, \dots, x_n : A_n \vdash x : A$, assume for any $i = 1, \dots, n$ that $d_iR_{A_i}e_i$. Then we want to show:

$$(\lambda(d_1, \dots, d_n) \in [\![\Gamma]\!].\ \pi_i(d_1, \dots, d_n))(d_1, \dots, d_n)\ R_A\ [e_1/x_1, \dots, e_n/x_n]x$$

As we have $\Gamma \vdash x : A$, by the typing rule for variables we have $\Gamma(x) = A$, so $x \in dom(\Gamma)$ and $\exists i.\ d_i R_{A_i} e_i$. Then on the right hand side we have $[e_i/x]x$. Therefore we want to show

$$d_i R_{A_i} e_i$$

which we have as an assumption.

Zero For $z : \text{Nat}$, we want to show:

$$[\![\Gamma \vdash z : \text{Nat}]\!](d_1, \ldots, d_n)\ R_{\text{Nat}}\ [e_1/x_1, \ldots, e_n/x_n]z$$

Expanding the definitions gives us $0\ R_{\text{Nat}}\ z$, so we must show $\forall n \in \mathbb{N}.0 = n \Rightarrow z \mapsto^* n$, which is the case as z reduces to n in zero steps. Therefore $0\ R_{\text{Nat}}\ z$.

Successor For $x_1 : A_1, \ldots, x_n : A_n \vdash s(e) : \text{Nat}$, assume for any $i = 1, \ldots, n$ that $d_i R_{A_i} e_i$. Then we want to show:

$$[\![\Gamma \vdash s(e) : \text{Nat}]\!](d_1, \ldots, d_n)\ R_{\text{Nat}}\ [e_1/x_1, \ldots, e_n/x_n](s(e))$$

Which expands to $\forall n \in \mathbb{N}.\ [\![\Gamma \vdash s(e) : \text{Nat}]\!](d_1, \ldots, d_n) = n \Rightarrow [e_1/x_1, \ldots, e_n/x_n]s(e) \mapsto^* n$. By the inductive hypothesis, we know:

$$[\![\Gamma \vdash e : \text{Nat}]\!](d_1, \ldots, d_n)\ R_{\text{Nat}}\ [e_1/x_1, \ldots, e_n/x_n]e$$

This expands to $\forall n \in \mathbb{N}.\ [\![\Gamma \vdash e : \text{Nat}]\!](d_1, \ldots, d_n) = n \Rightarrow [e_1/x_1, \ldots, e_n/x_n]e \mapsto^* n$. (If the denotation of e is $\bot$ then this is vacuously true.)

Therefore there will be two cases:

1. If $[\![\Gamma \vdash e : \text{Nat}]\!](d_1, \ldots, d_n) = \bot$, then we must show $\bot R_{\text{Nat}}[e_1/x_1, \ldots, e_n/x_n]s(e)$, which we get from Lemma 6

2. If $[\![\Gamma \vdash e : \text{Nat}]\!](d_1, \ldots, d_n) = v$, then we must show $v + 1\ R_{\text{Nat}}\ [e_1/x_1, \ldots, e_n/x_n]s(e)$. Let $n = v + 1$. From the inductive hypothesis we know that $[e_1/x_1, \ldots, e_n/x_n]e \mapsto^* v$. Using the congruence evaluation rule for successor, we get $s([e_1/x_1, \ldots, e_n/x_n]e) \mapsto s(v)$ and $s(v)$ is the same as $v + 1$. Therefore we have $v + 1\ R_{Nat} s(v)$, so if we use this with the congruence rule in Lemma 7, we have $v + 1\ R_{Nat} s([e_1/x_1, \ldots, e_n/x_n]e)$

Case For $x_1 : A_1, \dots, x_n : A_n \vdash \mathrm{case}(e, z \mapsto e_0, s(x) \mapsto e_S)$, assume for any $i = 1, \dots, n$ that $d_i R_{A_i} e_i$. Then we want to show:

$$[\![\Gamma \vdash \mathrm{case}(e, z \mapsto e_0, s(x) \mapsto e_S) : A]\!](d_1, \dots, d_n)$$

$$R_A$$

$$[e_1/x_1, \dots, e_n/x_n]\,\mathrm{case}(e, z \mapsto e_0, s(x) \mapsto e_S)$$

The result of the denotation depends on the value of e, so we have three cases:

1. $[\![\Gamma \vdash e : \mathrm{Nat}]\!](d_1, \dots, d_n) = \bot$, then we must show $\bot\ R_A\ [e_1/x_1, \dots, e_n/x_n]$ $\mathrm{case}(z \mapsto e_0, s(x) \mapsto e_S)$, which we get by applying Lemma 6.

2. $[\![\Gamma \vdash e : \mathrm{Nat}]\!](d_1, \dots, d_n) = 0$, then we want to show $[\![\Gamma \vdash e_0 : A]\!](d_1, \dots, d_n)$ $R_A[e_1/x_1, \dots, e_n/x_n]\,\mathrm{case}(z \mapsto e_0, s(x) \mapsto e_S)$. As we have $\Gamma \vdash e_0 : A$ from the typing rule of case, we can get $[\![\Gamma \vdash e_0 : A]\!](d_1, \dots, d_n)\ R_A[e_1/x_1, \dots, e_n/x_n]e_0$ by the induction hypothesis. We can now use this in Lemma 7, with the Operational Semantics rule for case when the expression is zero, to get

$$[\![\Gamma \vdash e_0 : A]\!](d_1, \dots, d_n)\ R_A[e_1/x_1, \dots, e_n/x_n]\,\mathrm{case}(z \mapsto e_0, s(x) \mapsto e_S)$$

3. $[\![\Gamma \vdash e : \mathrm{Nat}]\!](d_1, \dots, d_n) = n + 1$ we want to show

$$[\![\Gamma, x : \mathrm{Nat} \vdash e_S : \mathrm{Nat}]\!](d_1, \dots, d_n, d)$$

$$R_A$$

$$[e_1/x_1, \dots, e_n/x_n]\,\mathrm{case}(e, z \mapsto e_0, s(x) \mapsto e_S)$$

From the induction hypothesis we have

$$[\![\Gamma, x : \mathrm{Nat} \vdash e_S : \mathrm{Nat}]\!](d_1, \dots, d_n, d)\quad R_A\ [e_1/x_1, \dots, e_n/x_n, e/x]e_S$$

We can now use this in Lemma 7, with the operational semantics rule for case when the expression is not zero to get

$$[\![\Gamma, x : \mathrm{Nat} \vdash e_S : \mathrm{Nat}]\!](d_1, \dots, d_n, d)$$

$$R_A$$

$$[e_1/x_1, \dots, e_n/x_n]\,\mathrm{case}(e, z \mapsto e_0, s(x) \mapsto e_S)$$

.

Application For $x_1 : A_1, \ldots, x_n : A_n \vdash e\ e' : B$, assume for any $i = 1, \ldots, n$ that $d_i R_{A_i} e_i$. Then we want to show:

$$\llbracket \Gamma \vdash e\ e' : B \rrbracket (d_1, \ldots, d_n)$$

$$R_B$$

$$[e_1/x_1, \ldots, e_n/x_n]e\ e'$$

Using the Denotational Semantics and substitution function we can rewrite this as:

$$\llbracket \Gamma \vdash e : A \to B \rrbracket (d_1, \ldots, d_n)(\llbracket \Gamma \vdash e' : B \rrbracket (d_1, \ldots, d_n))$$

$$R_B$$

$$([e_1/x_1, \ldots, e_n/x_n]e)([e_1/x_1, \ldots, e_n/x_n]e')$$

By the inductive hypothesis we have

$$\llbracket \Gamma \vdash e : A \to B \rrbracket (d_1, \ldots, d_n)\ R_{A \to B}\ [e_1/x_1, \ldots, e_n/x_n]e$$

Expanding this gives us $\forall d \in \llbracket A \rrbracket.\ \forall e' \in \{e \mid\ \vdash e : A\}.\ dR_A e' \Rightarrow \llbracket \Gamma \vdash e : A \to B \rrbracket (d_1, \ldots, d_n)(d)\ R_B\ [e_1/x_1, \ldots, e_n/x_n]e\ e'$. Let $d = \llbracket \Gamma \vdash e' : B \rrbracket (d_1, \ldots, d_n)$ and $e' = [e_1/x_1, \ldots, e_n/x_n]e'$. Then we have

$$\llbracket \Gamma \vdash e : A \to B \rrbracket (d_1, \ldots, d_n)(\llbracket \Gamma \vdash e' : B \rrbracket (d_1, \ldots, d_n)\ R_B\ ([e_1/x_1, \ldots, e_n/x_n]e)([e_1/x_1, \ldots, e_n/x_n]e')$$

λ-Abstraction For $x_1 : A_1, \ldots, x_n : A_n \vdash \lambda x : A.\ e : A \to B$, assume for any $i = 1, \ldots, n$ that $d_i R_{A_i} e_i$. Then we want to show:

$$\llbracket \Gamma \vdash \lambda x : A.\ e : A \to B \rrbracket (d_1, \ldots, d_n)\ R_{A \to B}\ [e_1/x_1, \ldots, e_n/x_n](\lambda x : A.\ e)$$

Expanding the definition of the logical relation gives us

$$\forall d \in \llbracket A \rrbracket.\ \forall e' \in \{e \mid\ \vdash e : A\}.\ dR_A e' \Rightarrow$$

$$\llbracket \Gamma, x : A \vdash e : B \rrbracket (d_1, \ldots, d_n)(d)\ R_B\ [x_1/t_1, \ldots, x_n/t_n](\lambda x : A.\ e)\ e'$$

Let d be an element of the domain of type A and e' be an expression of type A such that dR_Ae'.

Then by the using the denotational semantics for λ abstraction, we want to show:

$$(\lambda d \in [\![A]\!].\ [\![\Gamma, x : A \vdash e : B]\!](d_1, \dots, d_n, d))(d)\ R_B\ [e_1/x_1, \dots, e_n/x_n](\lambda x : A.\ e)\ e'$$

which is the same as:

$$[\![\Gamma, x : A \vdash e : B]\!](d_1, \dots, d_n, d)\ R_B\ [e_1/x_1, \dots, e_n/x_n, e'/x]\ e$$

which we get by the inductive hypothesis, because from the evaluation rule we have

$$[e_1/x_1, \dots, e_n/x_n](\lambda x : A.\ e)\ e' \mapsto [e'/x][e_1/x_1, \dots, e_n/x_n]e = [e_1/x_1, \dots, e_n/x_n, e'/x]e$$

so we can use Lemma 7.

Fixpoint For $x_1 : A_1, \dots, x_n : A_n \vdash \operatorname{fix} x : A.\ e : A$, assume for any $i = 1, \dots, n$ that $d_i R_{A_i} e_i$. Then we want to show:

$$[\![\Gamma \vdash \operatorname{fix} x : A.\ e : A]\!](d_1, \dots, d_n)\ R_A\ [e_1/x_1, \dots, e_n/x_n](\operatorname{fix} x : A.\ e : A)$$

The denotation of the left hand side is $\mathsf{fix}(\lambda a \in [\![A]\!].\ [\![\Gamma, x : A \vdash e : A]\!](d_1, \dots, d_n, a))$.

By the fixpoint theorem, we know this is the same as

$$\bigsqcup_n (\lambda a \in [\![A]\!].\ [\![\Gamma, x : A \vdash e : A]\!](d_1, \dots, d_n, a))^n(\bot)$$

If we can prove that

$$\forall n \in \mathbb{N}.\ (\lambda a \in [\![A]\!].\ [\![\Gamma, x : A \vdash\ e : A]\!](d_1 \dots d_n, a))^n(\bot)\ R_A\ [e_1/x_1, \dots e_n/x_n] \operatorname{fix} x : A.\ e : A$$

then we can use Lemma 8, to get the above result.

We prove this by induction on n. When $n = 0$, we need to show $\bot\ R_A\ [e_1/x_1, \dots e_n/x_n] \operatorname{fix} x : A.\ e : A$, for which we just use Lemma 6.

For the inductive case, we must show

$$(\lambda a \in [\![A]\!].\ [\![\Gamma, x : A \vdash e : A]\!](d_1 \ldots d_n, a))^{n+1}(\bot)\ R_A\ [e_1/x_1, \ldots e_n/x_n]\,\mathrm{fix}\,x : A.\ e : A$$

We can rewrite the left hand side as $(\lambda a \in [\![A]\!].\ [\![\Gamma, x : A \vdash e : A]\!](d_1 \ldots d_n, a))(\lambda a \in [\![A]\!].\ [\![\Gamma, x : A \vdash e : A]\!](d_1 \ldots d_n, a))^n(\bot))$, which is the same as:

$$[\![\Gamma, x : A \vdash e : A]\!](d_1 \ldots d_n, (\lambda a \in [\![A]\!].\ [\![\Gamma, x : A \vdash e : A]\!](d_1 \ldots d_n, a))^n(\bot)))$$

We know that

$$(\lambda a \in [\![A]\!].\ [\![\Gamma, x : A \vdash e : A]\!](d_1 \ldots d_n, a))^n(\bot)))\ R_A\ [e_1/x_1, \ldots e_n/x_n]\,\mathrm{fix}\,x : A.\ e : A$$

by the inductive hypothesis, so we can use this as an assumption in the inductive hypothesis of the Main Lemma to get:

$$[\![\Gamma, x : A \vdash e : A]\!](d_1 \ldots d_n, (\lambda a \in [\![A]\!].\ [\![\Gamma, x : A \vdash e : A]\!](d_1 \ldots d_n, a))^n(\bot)))$$

$$R_A$$

$$[e_1/x_1, \ldots e_n/x_n, [e_1/x_1, \ldots e_n/x_n]\,\mathrm{fix}\,x : A.\ e : A/x]e$$

because

$$[e_1/x_1, \ldots e_n/x_n]\,\mathrm{fix}\,x : A.\ e : A$$

$$= \mathrm{fix}\,x : A.[e_1/x_1, \ldots e_n/x_n]e : A$$

$$\mapsto [\mathrm{fix}\,x : A.[e_1/x_1, \ldots e_n/x_n]e/x][e_1/x_1, \ldots e_n/x_n]e : A$$

$$= [[e_1/x_1, \ldots e_n/x_n]\,\mathrm{fix}\,x : A.e/x][e_1/x_1, \ldots e_n/x_n]e : A$$

$$= [e_1/x_1, \ldots e_n/x_n, [e_1/x_1, \ldots e_n/x_n]\,\mathrm{fix}\,x : A.e/x]e : A$$

We can the use Lemma 7 with this, to get

$$(\lambda a \in [\![A]\!].\ [\![\Gamma, x : A \vdash e : A]\!](d_1 \ldots d_n, a))^{n+1}(\bot)\ R_A\ [e_1/x_1, \ldots e_n/x_n]\,\mathrm{fix}\, x : A.\ e : A$$

Therefore $\forall n \in \mathbb{N}.\ (\lambda a \in [\![A]\!].\ [\![\Gamma, x : A \vdash\ e : A]\!](d_1 \ldots d_n, a))^n(\bot)\ R_A\ [e_1/x_1, \ldots e_n/x_n]\,\mathrm{fix}\, x : A.\ e : A$, so the Main Lemma holds for fixpoint.

□

7.4 Other Proofs of Adequacy

Other proofs for Adequacy of PCF exist in the literature. Here we discuss one approach to the proof that contrasts ours:

7.4.1 Proof using "Computable Terms"

The following definition of a computable term in PCF is given by [Gunter, 1992] and is adapted from [Plotkin, 1977]:

Definition 8. *A computable term is defined by the following statements:*

1. *If e is a closed PCF term of type* Nat*, then e is computable if $[\![e]\!] = n \Rightarrow e \mapsto^* n$*
2. *If e is a closed PCF term of type $A \to B$, it is computable if whenever e' is a closed computable term of type A, then $e\ e'$ is a closed computable term of type B*
3. *If e is an open term with free variables $x_1, \ldots, x_n$ of types $A_1, \ldots A_n$ then it is computable if $[e_1/x_1, \ldots e_n/x_n]e$ is computable when $\forall i.\ e_i$ are closed computable terms*

As described by [Gunter, 1992], by combining the second two parts of the definition of computable terms, we know that an expression e is computable iff there is some substitution σ of closed computable expressions for all the free variables of e (i.e. part 3 of the definition) and there are closed computable terms $e_1, \ldots e_n$ that make $(\sigma e)e_1, \ldots e_n$ a closed term of ground type, then $(\sigma e)e_1, \ldots e_n$ is computable (using part 2 of the definition).

This is used to prove that PCF terms of types other than the base type are computable:

Theorem 11. *Every PCF term is computable*

Proof. By induction on e. We show just a couple of cases here, for case and fix. The rest of the cases are given in [Gunter, 1992].

For case, we use the statement above and prove $\sigma\,\mathrm{case}(e, z \mapsto e_0, s(v) \mapsto e_S)(e_1, \ldots e_n)$ is a closed computable expression of ground type (where $(e_1, \ldots e_n)$ are all closed computable and σ contains only closed computable terms) to prove all possible case terms are computable. (This also works in closed case terms, where σ is just the empty substitution).

Assume $[\![\sigma\,\mathrm{case}(e, z \mapsto e_0, s(v) \mapsto e_S)(e_1, \ldots e_n)]\!] = [\![v]\!]$ for some v. Therefore $[\![\sigma e]\!] \neq \bot$, so there are two cases, one for True and one for False:

If $[\![\sigma e]\!] = n$ for some $n > 0$, then the entire denotation is $[\![\sigma e_S(e_1, \ldots e_n)]\!]$ and by the inductive hypothesis, $\sigma e_S(e_1, \ldots e_n) \mapsto v$. By the inductive hypothesis we also have $\sigma e \mapsto n$. By using the congruence rule and evaluation rule in the operational semantics, we get $\sigma\,\mathrm{case}(e, z \mapsto e_0, s(v) \mapsto e_S)(e_1, \ldots e_n) \mapsto v$.

The False case is similar, but instead we use the induction hypothesis on e to get $\sigma e \mapsto 0$ and $\sigma e_0 \mapsto v$.

Fixpoint is the hardest case. We want to prove $(\sigma\,\mathrm{fix}\,x : A.\ e)(e_1 \ldots e_n)$ is computable if $e_1 \ldots e_n$ are closed computable and all the free variables of e have been substituted. To do this we use a syntactic approximation described by the following lemma, which is proved in [Gunter, 1992] :

Lemma 9. *Let $e = \mathrm{fix}\,x : A.\ e' : A$ be a well-typed expression and define:*

$$e^0 = \mathrm{fix}\,x : A.\ x$$
$$e^{k+1} = (\lambda x : A.e')(e^k)$$

Then

1. $[\![e]\!] = \bigsqcup_k [\![e^k]\!]$
2. *If $e^k\ e_1, \ldots e_n \mapsto v$ for some v and e is of ground type, then $e\ e_1, \ldots e_n \mapsto v$*

So we now want to first prove σe^k is closed computable.

We prove this by induction on k. $[\![e^0]\!] = \bot$, so there is no value that e^0 evaluates to, so computability is vacuously true.

For the inductive case, we assume σe^{k-1} is closed computable, and then evaluate σe^k in the operational semantics to get $[\sigma, \sigma e^{k-1}/x]e'$.

By the inductive hypothesis, we know σe^{k-1} and e are closed computable, so $[\sigma, \sigma e^{k-1}/x]e'$ is too. Therefore σe^k is computable.

Now we know σe^k is computable, we use this to show that $\sigma e(e_1, \ldots, e_n)$ is a closed computable term of ground type, which we can then use to show that σe is computable.

Assume that $e_1, \dots e_n$ are closed computable terms such that $\sigma e(e_1, \dots e_n)$ is of ground type and that $[\![\sigma e(e_1, \dots e_n)]\!] = [\![v]\!]$. By Lemma 9 part *1*, we have $\bigsqcup [\![\sigma e^k(e_1, \dots e_n)]\!] = [\![v]\!]$. Since this term is of ground type, there will be some k such that $[\![\sigma e^k(e_1, \dots e_n)]\!] = [\![v]\!]$. Then since we know σe^k is computable, and that $e_1, \dots e_2$ are closed computable, we have $\sigma e^k(e_1, \dots e_n) \mapsto v$ (from the definition of computable). By using Lemma 9 part *2*, we have $\sigma e(e_1, \dots e_n) \mapsto v$.

Therefore $\sigma e(e_1, \dots e_n)$ is a closed computable term of ground type where σ substitutes the free variables with closed computable terms and $(e_1, \dots e_n)$ are all closed computable terms, so by the remark before the proof, we know that $\sigma e = \sigma(\mathrm{fix}\, x : A.\ e : A)$ is a closed computable term.

□

7.4.2 Differences from our Logical Relation

The definition of computable could also be considered as a unary logical relation (or logical predicate), where PCF expressions satisfying the definition of computable are in the relation.

This predicate would only be defined on the syntax of PCF and not on the semantics, which makes it much more difficult to prove the fixpoint case, as we have to use the Lemma 9 which is quite complicated to prove. [Gunter, 1992] uses another version of PCF with bounded recursion and an Unwinding Theorem (of which Lemma 9 is a corollary) to translate it back to our PCF, which adds complexity to their proof.

Chapter 8

Non-Definability of Parallel-Or

The following chapter is based on contents from the chapters "The Full Abstraction Problem" and "Logical Relations" from the book [Streicher, 2006].

In Chapter 6, we described how our denotational model can represent any PCF expression. We can show that this model also contains extra functions that do not correspond to any expression. One such example is the function parallel or, written as por.

Parallel or (por) is the function defined in the following way (where $0 = \mathit{true}$):

$$\text{por}\ \ x\ y = \begin{cases} 0 & \text{if } x = 0 \lor y = 0 \\ 1 & \text{if } x = y = 1 \\ \bot & \text{otherwise} \end{cases}$$

which gives us the following truth table:

	0	1	$\bot$
0	0	0	0
1	0	1	$\bot$
$\bot$	0	$\bot$	$\bot$

By observation, parallel or is not definable as we would have to evaluate its arguments in parallel and PCF is a sequential language.

We can formally prove that por is non-definable by using logical relations.

8.0.1 Logical Relation Definition

We specify a logical relation on denotations of program terms in PCF by defining relations on each type individually. We use Streicher's definition, as given in [Streicher, 2006]:

Definition 9. *Let W be an arbitrary set. A W-ary logical relation on the model of PCF is a family of relations*

$$R = (R_A \in \mathcal{P}(\llbracket A \rrbracket^W) \mid A \in Type)$$

such that

$$f \in R_{A \to B} = \forall d \in R_A.\ \lambda i \in W.\ f(i)d(i) \in R_B$$

where $Type$ is the set of all possible types our programs can have, defined by induction.

Logical relations for different properties are constructed by defining the relation differently on the base type for each property. For example, if the types are defined by the base type Nat and other types $A \to B$ (formed any other types A and B), then the relation is defined by the definition of R_{Nat}.

Therefore we say that a logical relation R of arity W is uniquely determined by R_{Nat}, so for all subsets of $\llbracket \text{Nat} \rrbracket^W$ there is a unique R equal to the set.

Logical Relations at Function Types For a function $f = (f_1, \dots f_n)$ to be in the relation, if we apply it to **any** value that is in the relation of the type of its domain, for example, for arity 3:

$$(x, y, z) \in R_A$$

Then f applied to everything in these elements will be in the relation of the codomain, so we must have:

$$(f_1(x), f_2(y), f_3(z)) \in R_B$$

Then $f \in R_{A \to B}$.

8.0.2 Examples and Counter-Examples

The examples in this section are our answers to exercises from Lecture 1 of [Murawski, 2012]:

Applying certain restrictions on R_{Nat} restricts the relations we can define on function types, for example:

- If $R_{\mathrm{Nat}} = [\![\mathrm{Nat}]\!]^n$, relations on function types can only have n arguments, if n is specified, otherwise there is no difference to the definition.
- If $R_{\mathrm{Nat}} = \emptyset$, then all relations in R are the empty relation.

Union of two logical relations Given two logical relations R and S of the same arity, we could try to form a logical relation $R \cup S$. This is **not** a logical relation. For example, if we have $(f_1, \ldots, f_n) \in R_{A\to B}$ (and not in $S_{A\to B}$) and $(d_1, \ldots, d_n) \in S_A$, (and not in $R_{A\to B}$) then $(f_1(d_n), \ldots, f_n(d_n))$ cannot be in either R or S.

Intersection of two logical relations Therefore if we restrict the logical relation to only contain tuples that are in both R and S then we do not have this problem, so this is a logical relation.

If the relations did not have the same arity, there will be no tuples that are in both relations that satisfy the definition, as if we have R of arity 3 and S of arity 5, then have, for example, $(f, g, h) \in R_{A\to B}$ and $(f, g, h, i, j) \in S_{A\to B}$, then we cannot say that (f, g, h) is in both, as this would not be in S anymore. Therefore, this would not be a logical relation.

Composition of two binary logical relations We define $R;S$ in the following way:

$$(R;S)(x,y) \Leftrightarrow \exists z.\ (R(x,z)\ \wedge\ S(z,y))$$

So for base type we have:

$$(R_{\mathrm{Nat}};S_{\mathrm{Nat}})(x,z) \Leftrightarrow \exists y.\ (R_{\mathrm{Nat}}(x,y)\ \wedge\ S_{\mathrm{Nat}}(y,z))$$

and for functions we have

$$(R_{A\to B};S_{A\to B})(f,h) \Leftrightarrow \exists g.\ (R_{A\to B}(f,g)\ \wedge\ S_{A\to B}(g,h))$$

where for any $(x,z) \in (R_A;S_A)$ we have

$$(R_B(f(x), g(y)) \ \wedge \ S_B(g(y), h(z)))$$

So as functions in R are only applied to inputs that are in R and the same for S, and functions in both are only applied to inputs that are in both, then this is a logical relation.

8.0.3 Main Lemma

When using logical relations, we usually prove a general theorem about the relation, which we then give specific inputs to, to get the proof of our original property. This is called the Main Lemma.

For any denotation of a PCF term, we want to show that it is in a given logical relation at its type, so we want to show that $[\![\Gamma \vdash e : A]\!] \in R_{\Gamma \to A}$. An element of $[\![\Gamma]\!]$ is any tuple of substitutions $d^* = (d_1, \dots d_n)$ for $x_1 : A_1, \dots x_n : A_n = \Gamma$. So $R_\Gamma \in \mathcal{P}([\![A_1]\!] \times \cdots \times [\![A_n]\!])$.

We want all substitutions in a set W to be in the relation, so by using the definition of $[\![\Gamma \vdash e : A]\!] \in R_{\Gamma \to A}$, we want to show that

$$\forall (d^*(1), \dots d^*(W)) \in R_\Gamma . \ \lambda i \in W . [\![\Gamma \vdash e : A]\!] d^*(i)$$

This says that for any position in the W-tuple, we have the denotation of e using the substitution $d^*(i)$ (the substitution from the ith position in the tuple $(d^*(1), \dots, d^*(W)) \in R_\Gamma$).

For W different substitutions we want to have

$$([\![\Gamma \vdash e : A]\!] d^*(1), \dots, [\![\Gamma \vdash e : A]\!] d^*(W)) \in R_B$$

Therefore for program terms, we define the Main Lemma in the following way, as in [Streicher, 2006], where we only need to define the lemma on λ terms. This is because any closed PCF term can be written as a λ-term in the following way:

- z
- $\lambda x : \mathrm{Nat} \, . \ s(x)$
- $\lambda x : \mathrm{Nat}, e_0 : A, e_S : A. \ \mathrm{case}(e, z \mapsto e_0, s(x) \mapsto e_S)$
- $\lambda x : A. \ \lambda e : A. \ \mathrm{fix}\, x : A \ e$

The statement of the Main Lemma is the following:

Lemma 10. *Let R be a logical relation of arity W on the Scott Model of PCF. Then for λ terms $\Gamma \vdash e : A$ and $d_j \in R_{A_j}$ for $j = 1, \ldots, n$*

$$\lambda i \in W. [\![\Gamma \vdash e : A]\!] d^*(i) \in R_A$$

where $d^*(i) = (d_1(i) \ldots d_n(i))$ and $\Gamma = x_1 : A_1, \ldots x_n : A_n$

We include the proof given by [Streicher, 2006], additionally explaining how the case for variables is proved:

Proof. By induction on λ terms:

Variables Assume $x_1 : A_1, \ldots, x_n : A_n \vdash x_j : A_j$ and $d_j \in R_{A_j}$ for every j. We need to show that $\lambda i \in W.\ [\![x_1 : A_1, \ldots x_n : A_n \vdash x_j : A_j]\!](d^*(i)) \in R_{A_j}$. $d^*(i)$ includes $d_j(i)$, so by the denotational semantics for variables, we want to show that $\lambda i\ \in W.d_j(i) \in R_{A_J}$. As we have assumed that $d_j \in R_{A_j}$, then at each ith position, we already have $d_j(i) \in R_{A_j}$ as an assumption, so we have $\lambda i\ \in W.d_j(i) \in R_{A_j}$.

λ-Abstraction Assume that $\Gamma \vdash \lambda x : A.\ e : B$ and $\forall j \leq n.\ d_j \in R_{A_j}$. We need to show that $\lambda i \in W.\ [\![\Gamma \vdash \lambda x : A.\ e : B]\!](d^*(i)) \in R_{A \to B}$. As this is a function type, we expand the definition of the logical relation to get:

$$\forall d \in R_A.\ \lambda i \in W.\ [\![\Gamma \vdash \lambda x : A.e : B]\!](d^*(i))(d(i)) \in R_B$$

Let d be a substitution in R_A. We use the induction hypothesis with $\Gamma, x : A \vdash e : B$ (obtained by using the typing rule of λ-abstraction) and $d_j \in R_{A_j}$ and $d \in R_A$ to get:

$$\lambda i \in W.\ [\![\Gamma, x : A \vdash e : B]\!](d^*(i), d(i)) \in R_B$$

By the definition of the Denotational Semantics for λ-abstraction, this is the same as:

$$\lambda i \in W.\ [\![\Gamma \vdash \lambda x : A.e : B]\!](d^*(i))(d(i)) \in R_B$$

Therefore, $\forall d \in R_A.\ \lambda i \in W.\ [\![\Gamma \vdash \lambda x : A.e : B]\!](d^*(i))(d(i)) \in R_B$, so we have

$$\lambda i \in W.\ [\![\Gamma \vdash \lambda x : A.\ e : B]\!](d^*(i)) \in R_{A \to B}$$

which is what we wanted to prove.

Application Assume that $\Gamma \vdash e\ e' : B$ and $\forall j \leq n.\ d_j \in R_{A_j}$. We need to show that $\lambda i \in W.\ [\![\Gamma \vdash e\ e' : B]\!](d^*(i)) \in R_B$.

By using the induction hypothesis on $\Gamma \vdash e : A \to B$ and $\Gamma \vdash e' : A$ (which we obtain by using the typing rule of function application), we get $\lambda i \in W.\ [\![\Gamma \vdash e : A \to B]\!](d^*(i)) \in R_{A \to B}$. and $\lambda i \in W.\ [\![\Gamma \vdash e' : A]\!](d^*(i)) \in R_A$.

Expanding the definition of the first result gives us:

$$\forall d \in R_A.\ \lambda i \in W.\ [\![\Gamma \vdash e : A \to B]\!](d^*(i))(d(i)) \in R_B$$

Let $d = \lambda i \in W. [\![\Gamma \vdash e' : A]\!](d^*(i)) \in R_A$. Then we have:

$$\lambda i \in W.\ [\![\Gamma \vdash e : A \to B]\!](d^*(i))([\![\Gamma \vdash e' : A]\!](d^*(i)) \in R_B$$

Using the definition of the Denotational Semantics for function application, we have:

$$\lambda i \in W.\ [\![\Gamma \vdash e\ e' : B]\!](d^*(i)) \in R_B$$

which is what we wanted to prove. □

To use our relation to prove por is not definable in PCF, we must first define a property that a logical relation may or may not satisfy:

8.1 R-invariant

When the denotation of an expression e of type A is R-**invariant**, there is an element in R_A of the form $([\![\Gamma \vdash e : A]\!]d^*, \ldots, [\![\Gamma \vdash e : A]\!]d^*) \in R_A$

We can define this in general for an denotation $d \in [\![A]\!]$:

Definition 10. *[Streicher, 2006]Let R be a logical relation of arity W. Then $d \in [\![A]\!]$ is called R-invariant if*

$$\delta_W(d) = \lambda i \in W.\ d \in R_A$$

To understand how this works for λ terms, we can prove the following as a corollary of the Main Lemma:

Corollary 2. *[Streicher, 2006] If R is a logical relation of arity W and e is a λ term of type A, then $\lambda i \in W.\ [\![e]\!] \in R_A$*

Therefore any element of the denotational model that is the denotation of a closed λ term is R-invariant.

By proving another corollary of the Main Lemma, it can be proved that any λ expression is R-invariant (which is done in [Streicher, 2006]). For non-closed expressions, we can prove that as long as the denotations of all the expressions in the substitution are R-invariant, then the denotation of the entire non closed expression is also R-invariant, as a corollary of the main lemma (see Lemma 10):

Corollary 3. *[Streicher, 2006] Let R be a logical relation on the Scott Model of arity W and $\Gamma \vdash e : B$ be a λ term. Then $[\![\Gamma \vdash e : B]\!](d^*(i))$ is R-invariant whenever all $d \in d^*$ are.*

Now we know that all the denotations of λ terms are R-invariant, so any **closed** PCF term that can be written as a λ term will have an R-invariant denotation.

The syntax of PCF can be written as λ terms in the following way, as previously described:

- z
- $\lambda x : \mathrm{Nat}\,.\ s(x)$
- $\lambda x : \mathrm{Nat}, e_0 : A, e_S : A.\ \mathrm{case}(e, z \mapsto e_0, s(x) \mapsto e_S)$
- $\lambda x : A.\ \lambda e : A.\ \mathrm{fix}\, x : A\ e$

so these λ terms can be composed to create λ terms for any closed PCF term, so any closed PCF term will be R-invariant.

Therefore, we want to ensure that all of the λ terms describing PCF expressions are R-invariant. The most difficult expression to check is R-invariant is $\lambda x : A.\ \lambda e : A.\ \mathrm{fix}\, x : A\ e$. For this we require another property on logical relations:

8.2 Admissible Logical Relations

An admissible logical relation is the following:

Definition 11. *[Streicher, 2006] A logical relation R of arity W is called **admissible** if $\delta_W(\bot) \in R_{Nat}$ and for all chains $d_1 \sqsubseteq d_2 \sqsubseteq \dots$, if $d_i \in R_{\mathrm{Nat}}$, then $\bigsqcup d_n \in R_{\mathrm{Nat}}$.*

In [Streicher, 2006], the above definition is defined for directed sets. If R_{Nat} is closed under directed suprema (i.e. the least upper bound of a directed set), then this is the same as saying that if we form chains $d_0 \sqsubseteq d_1 \sqsubseteq \dots$ of W-tuples of elements in $\mathbb{N}_\perp$'s underlying set, if $d_i \in R_{\text{Nat}}$, then $\bigsqcup d_n \in R_{\text{Nat}}$. This is how we obtained our definition of admissible.

It is also stated in [Streicher, 2006] that for finite W, there are no non trivial chains of $[\![\text{Nat}]\!]^W$, so in these cases, we can just say the logical relation is admissible if $\delta_W(\perp) \in R_{Nat}$.

To show that $\lambda x : A.\ \lambda e : A.\ \text{fix}\, x : A.\ e : A$ is R-invariant, we prove the following theorem, from [*Streicher*, 2006], which is modified for our definition of fixpoint and the definition of domains and admissible logical relations using chains. Therefore the proof is quite different from the original:

Theorem 12. *[Streicher, 2006] Let R be an admissible logical relation of arity W. Then for all types A we have:*

1. *$\delta_W(\perp) \in R_A$ and R_A for all chains $d_1 \sqsubseteq d_2 \sqsubseteq \dots$, if $d_i \in R_A$, then $\bigsqcup d_n \in R_A$*
2. *The denotation of $\lambda x : A.\lambda e : A.\,\text{fix}\, x : A.\ e : A$ is R-invariant*

Proof. We prove *1.* by induction on types. For base type, Nat, our goal is the same as the definition of admissible and we know R is admissible.

For function types $A \to B$, using the inductive hypothesis, we know that $\delta_W(\perp) \in R_A$ and $\delta_W(\perp) \in R_B$ and that for R_A, for all chains $d_1 \sqsubseteq d_2 \sqsubseteq \dots$, if $d_i \in R_A$, then $\bigsqcup d_n \in R_A$ and that this holds for R_B too, as we assume they are admissible.

We want to show that $\delta_W(\perp) \in R_{A\to B}$, which is the same as $\lambda i \in W.\perp \in R_{A\to B}$. As $A \to B$ is a function type, $\perp$ here is the function $\perp = \lambda x.\perp(x)$. Therefore we must show that $\forall d \in R_A.\lambda i \in W.(\lambda x.\perp(x))(d(i)) \in R_B$, which is the same as $\forall d \in R_A.\lambda i \in W.\perp \in R_B$.

Let $d \in R_A$. Then we must show $\lambda i \in W.\perp \in R_B$. This is the same as $\delta_W(\perp) \in R_B$, which we already have.

To show that the limit of chains of elements in $R_{A\to B}$ is also in $R_{A\to B}$, we first assume we have a chain of functions $f_1 \sqsubseteq f_2 \sqsubseteq \dots$, where $f_m \in R_{A\to B}$ for every element in the chain. Expanding any one of these definitions gives us:

$$\forall d \in R_A.\ \lambda i \in W.f_m(i)d(i) \in R_B$$

Let d be a tuple of elements of $[\![A]\!]$'s underlying set that is in R_A. Then we have $\lambda i \in W.\ f_m(i)d(i) \in R_B$, so we can have a chain of elements of R_B (by monotonicity). By the inductive hypothesis, we know that the limit of a chain formed of any elements of R_B is also in R_B. Therefore the element $\lambda i \in W.\ \bigsqcup f_n(i)d(i) \in R_B$ (by continuity).

Therefore $\forall d \in R_A.\ \lambda i \in W.\ \bigsqcup f_n(i)d(i) \in R_B$, so $\bigsqcup f_n \in R_{A\to B}$.

Note that this is a more general form of the function types case of the Chains Lemma (see Lemma 8)

To prove *2*, we must show

$$\lambda i \in W.\ [\![\lambda x : A.\ \lambda e : A.\ \mathrm{fix}\, x : A.\ e : A]\!] \in R_{A \to A \to A}$$

By the definition of the denotational semantics, this is the same as:

$$\lambda i \in W.\ (\lambda x' \in [\![A]\!].\lambda e' \in [\![A]\!].\ [\![x : A, e : A \vdash \mathrm{fix}\, x : A.\ e : A]\!](x', e')) \in R_{A \to A \to A}$$

As this is a function type, we should have:

$$\lambda i \in W.(\forall d_x \in R_A.\forall d_e \in R_A.$$
$$\lambda i \in W.\ (\lambda x' \in [\![A]\!].\lambda e' \in [\![A]\!].\ [\![x : A, e : A \vdash \mathrm{fix}\, x : A.\ e : A]\!](e', x'))(d_x(i))(d_e(i)) \in R_A$$

Therefore we assume that we have a logical relation R_A of arity W and that $d_x \in R_A$ and $d_e \in R_A$. Then we want to show:

$$\lambda i \in W.\ [\![x : A, e : A \vdash \mathrm{fix}\, x : A.\ e : A]\!](d_x(i), d_e(i)) \in R_A$$

Using the denotational semantics for the fixpoint case, we have:

$$\lambda i \in W.\ \mathsf{fix}(\lambda a \in [\![A]\!].[\![x : A, e : A \vdash e : A]\!](d_x(i), d_e(i), a) \in R_A$$

Which reduces to the following by using the denotational semantics for variables:

$$\lambda i \in W.\ \mathsf{fix}(\lambda a \in [\![A]\!].d_e(i)) \in R_A$$

By using the fixpoint theorem (Lemma 2.4), we know that this is the same as:

$$\lambda i \in W.\ \bigsqcup(\lambda a \in [\![A]\!].d_e(i))^n(\bot) \in R_A$$

So we prove that $\forall n.\lambda i \in W.(\lambda a \in [\![A]\!].d_e(i))^n(\bot) \in R_A$, by induction on n.

When $n = 0$, we have $\lambda i \in W.\bot$, which we know is in R_A from part *1* of this proof.

When $n = n + 1$, we know $\lambda i \in W.(\lambda a \in [\![A]\!].d_e(i))^n(\bot) \in R_A$, as this is the inductive hypothesis. As all denotations of PCF functions are continuous, and continuous functions are monotone, we can have $\lambda i \in W.(\lambda a \in [\![A]\!].d_e(i))(\lambda a \in [\![A]\!].d_e(i)^n(\bot)) = \lambda i \in W.d_e(i) \in R_A$.

Therefore $\lambda i \in W. \bigsqcup(\lambda a \in [\![A]\!].d_e(i))^n(\bot) \in R_A$, so the whole fixpoint λ term is R-invariant and we have completed the proof.

□

Now we have proved this, we can prove the following:

Theorem 13. *[Streicher, 2006] Let R be an admissible logical relation such that the denotations of the terms:*

- z
- $\lambda x : \mathrm{Nat}\,.\; s(x)$
- $\lambda x : \mathrm{Nat}, e_0 : A, e_S : A.\; \mathrm{case}(e, z \mapsto e_0, s(x) \mapsto e_S)$

are all R-invariant. Then all denotations of closed PCF terms are R-invariant.

Proof. Assume we have an admissible logical relation R and the given λ terms have R-invariant denotations. Then we just need to show the denotation of $\lambda x : A.\lambda e : A.\, \mathrm{fix}\, x : A.\; e : A$, which we get by Theorem 12. Therefore all denotations of closed PCF terms are R-invariant. □

8.3 Admissible Logical Relations for Non-Definability of por

Now we have defined admissible logical relations and proved that closed PCF terms are R-invariant for any such logical relation, we give two logical relations, defined in [Streicher, 2006] which we will use to prove that por is non-definable in PCF.

The following two logical relations are specified by defining them at base types in the following way:

$$(x, y, z) \in R^1_{Nat} = x \uparrow y \;\wedge\; z = x \sqcap y$$

$$(x, y, z) \in R^2_{Nat} = x = \bot \;\vee\; y = \bot \;\vee\; z = \bot \;\vee\; x = y = z$$

(where $x \sqcap y$ is the greatest lower bound of x and y and $x \uparrow y = \exists z.\; x \sqsubseteq z \;\wedge\; y \sqsubseteq z$)

At function type, the definitions will be as for any logical relation:

$$f \in R^1_{A\to B} = \forall d \in R^1_A.\ \lambda i \in W.f(i)(d(i)) \in R^1_B$$

$$f \in R^2_{A\to B} = \forall d \in R^2_A.\ \lambda i \in W.f(i)(d(i)) \in R^2_B$$

We want to show that the denotations of all closed PCF terms are invariant in them. We use Theorem 12, so we must prove two things. First we prove that both logical relations are admissible. As $W = 3$, from the remark before, we just prove that $(\bot, \bot, \bot)$ is in each logical relation, to show this.

R_1 is admissible For R_1, $z = \bot$, so $x \uparrow y$ and $\bot \sqcap \bot = z$, so $\delta_W(\bot)$ is in R^1.

R_2 is admissible $\delta_W(\bot)$ is in R^2, as everything is equal to $\bot$.

Secondly, we must show that the denotations of all the λ definitions of the constructors (except fix) are R-invariant for each relation. (This is left as exercise in [Streicher, 2006], we prove it fully here):

8.3.1 Constructors are in R_1

Lemma 11. *$[\![z]\!]$ is R^1 invariant*

Proof. We must show that $(0, 0, 0) \in R^1_{\text{Nat}}$. Let $z = 0$. Then $0 \sqsubseteq 0$, so $x \uparrow y$. As x and y are both 0, their greatest lower bound will be too, so $x \bigsqcap y = 0 = z$. Therefore $(0, 0, 0) \in R^1_{\text{Nat}}$. □

Lemma 12. *$[\![\lambda x : \text{Nat}\,.\ s(x)]\!]$ is R^1 invariant.*

Proof. We want to show that $(\lambda n \in \mathbb{N}_\bot.\ [\![x : \text{Nat} \vdash s(x) : \text{Nat}]\!](n/x), \lambda n \in \mathbb{N}_\bot.\ [\![x : \text{Nat} \vdash s(x) : \text{Nat}]\!](n/x), \lambda n \in \mathbb{N}_\bot.\ [\![x : \text{Nat} \vdash s(x) : \text{Nat}]\!](n/x)) \in R^1_{\text{Nat}\to\text{Nat}}$.

As these are functions of type Nat $\to$ Nat, we must show that for any $(x, y, z) \in R^1_{\text{Nat}}$, that the denotations we obtain when these values replace n in each of the functions are still related by the relation. If any of these values are $\bot$, the denotation will also be $\bot$. If any of (x, y, z) are n, then the result of their denotation function will be $n + 1$. For $x \uparrow y$ to be true, if either x or y are n then the other one must also be n. If $z = x \sqcap y$, then if x or y is

n, then z must be n. The conditions in R^1_{Nat} still be true if we add 1 to the value that is n, as numbers can only be related to themselves or $\bot$, so it does not matter what number n is.

Therefore, for any $(x, y, z) \in R^1_{\text{Nat}}$,

$$((\lambda n \in \mathbb{N}_\bot.\ [\![x : \text{Nat} \vdash s(x) : \text{Nat}]\!](n/x))(x),$$
$$(\lambda n \in \mathbb{N}_\bot.\ [\![x : \text{Nat} \vdash s(x) : \text{Nat}]\!](n/x))(y),$$
$$(\lambda n \in \mathbb{N}_\bot.\ [\![x : \text{Nat} \vdash s(x) : \text{Nat}]\!](n/x))(z))$$
$$\in R^1_{\text{Nat}}$$

as the $\bot$s stay the same and we add 1 to the ns, so all the properties are preserved. This means that the denotation of the λ term for successor is R^1 invariant. □

Lemma 13. *$[\![\lambda e : \text{Nat}, x : \text{Nat}, e_0 : A, e_S : A.\, \text{case}(e, z \mapsto e_0, s(x) \mapsto e_S)]\!]$ is R^1 invariant.*

Proof. We want to show that

$$\lambda i \in W.($$
$$\lambda n \in \mathbb{N}_\bot, x' \in \mathbb{N}_\bot, e'_0 \in [\![A]\!], e'_S \in [\![A]\!].$$
$$[\![e : \text{Nat}, x : \text{Nat}, e_0 : A, e_S : A \vdash \text{case}(e, z \mapsto e_0, s(x) \mapsto e_S)]\!]$$
$$(n/e, x'/x, e'_0/e_0, e'_S/e_S))$$

.

$$\in R^1_{\text{Nat}\to\text{Nat}\to A\to A\to A}$$

As this is a function type, we assume that we have $(x, y, z) \in R^1_{\text{Nat}}$. Then we can replace n with x, y and z. If any of these values were $\bot$, the resulting denotation is $\lambda x' \in \mathbb{N}_\bot, e'_0 \in [\![A]\!], e'_S \in [\![A]\!].\bot$.

If it is 0, then we have

$$\lambda x' \in \mathbb{N}_\bot, e'_0 \in [\![A]\!], e'_S \in [\![A]\!].$$
$$[\![e : \text{Nat}, x : \text{Nat}, e_0 : A, e_S : A \vdash e_0]\!]$$
$$(n/e, x'/x, e'_0/e_0, e'_S/e_S)$$

and for e_S we have the same but we swap out the denotation of e_0 for the denotation of e_S.

The denotation of the function with $\perp$ will have less information than the other two functions, which are equally informative, so the properties in $(x, y, z) \in R^1_{\text{Nat}}$ will be preserved. Therefore the λ term for case is R^1 invariant. $\square$

8.3.2 Constructors are in R_2

Lemma 14. *$[\![z]\!]$ is R^2 invariant*

Proof. We want to show that $(0, 0, 0) \in R^2_{\text{Nat}}$, which we know because $0 = 0 = 0$. $\square$

Lemma 15. *$[\![\lambda x.\ s(x)]\!]$ is R^2 invariant*

Proof. Given some $(x, y, z) \in R^2_{\text{Nat}}$, then if any of these values are $\perp$, then

$$(\lambda n \in \mathbb{N}_\perp.\ [\![x : \text{Nat} \vdash s(x) : \text{Nat}]\!](n/x))(\perp)$$

will also be $\perp$, so the successor term is in R^2_{Nat}.

If they are equal, then the denotation of the successor terms will either all be the same $n+1$, or will all be $\perp$, so will also be in R^2_{Nat}.

Therefore for all $(x, y, z) \in R^2_{\text{Nat}}$ that are given as parameters to $\lambda i \in W.[\![\lambda x.\ s(x)]\!]$, the resulting denotations will also be in R^2_{Nat}, so $\lambda i \in W.[\![\lambda x.\ s(x)]\!] \in R^2_{\text{Nat}\to\text{Nat}}$, and $[\![\lambda x.\ s(x)]\!]$ is R^2 invariant. $\square$

Lemma 16. *$[\![\lambda e : \text{Nat}, x : \text{Nat}, e_0 : A, e_S : A.\, \text{case}(e, z \mapsto e_0, s(x) \mapsto e_S)]\!]$ is R^2 invariant.*

Proof. For some $(x, y, z) \in R^2_{\text{Nat}}$, if one of them is equal to $\perp$, then the denotation of the case expression with it will be $\lambda x \in \text{Nat},\ e_0 \in [\![A]\!], e_S \in [\![A]\!].\ \perp$, which is the function that does not terminate, so will be $\perp$ in $R^2_{\text{Nat}\to\text{Nat}\to A\to A\to A}$.

Otherwise, $x = y = z = n$, so the denotation of all the case expressions is either

$$\lambda x' \in \mathbb{N}_\perp, e'_0 \in [\![A]\!], e'_S \in [\![A]\!].$$

$$[\![e : \text{Nat}, x : \text{Nat}, e_0 : A, e_S : A \vdash e_0]\!]$$

$$(n/e, x'/x, e'_0/e_0, e'_S/e_S)$$

or

$$\lambda x' \in \mathbb{N}_\perp, e_0' \in [\![A]\!], e_S' \in [\![A]\!].$$

$$[\![e : \text{Nat}, x : \text{Nat}, e_0 : A, e_S : A \vdash e_S]\!]$$

$$(n/e, x'/x, e_0'/e_0, e_S'/e_S)$$

so they will all be the same function. Therefore the λ term is R^2 invariant.

□

Now we know that all PCF constructor λ terms are all R^1 invariant, so by Theorem 13, all denotations of closed PCF terms are R^1 invariant.

8.3.3 Stable PCF functions

Now we can prove a lemma which says that all first order PCF terms are *stable*, from which the non-definability of por is a corollary. Stable means that binary infima (greatest lower bounds) of consistent pairs are preserved (i.e. given x and y such that $x \uparrow y$, then if $z = x \sqcap y$, then $f(z) = f(x) \sqcap f(y)$):

Lemma 17. *[Streicher, 2006] For every expression e of first order type, (i.e. of type* Nat $\to$ Nat $\to \cdots \to$ Nat*, (for some finite number of* Nat*s)), we have:*

$$[\![e]\!](x_1 \sqcap y_1)\dots(x_k \sqcap y_k) = [\![e]\!](x_1)\dots(x_k) \sqcap [\![e]\!](y_1)\dots(y_k)$$

for all x^* and $y^* \in [\![\text{Nat} \times \cdots \times \text{Nat}]\!]$, with $x^i \uparrow y^i$ for all $i = 1, \dots, k$

We give the proof as in [Streicher, 2006], explaining it in more detail:

Proof. As we know that all the λ terms of the constructors are R^1 invariant and R^1 is admissible, then the denotation of any closed PCF term is R^1 invariant. This means for any function $[\![e]\!]$ that describes the denotation of a closed PCF term, and any $(x, y, z) \in R^1_{\text{Nat}}$, that

$$[\![e]\!](x) \uparrow [\![e]\!](y) \;\wedge\; [\![e]\!](z) = [\![e]\!](x) \sqcap [\![e]\!](y)$$

We know every $x^i \uparrow y^i$ so we have $(x^i, y^i, x^i \sqcap y^i) \in R^1_{\text{Nat}}$.

Applying the definition of $([\![e]\!], [\![e]\!], [\![e]\!]) \in R^1_{\text{Nat}\to\dots\text{Nat}}$ we get:

$$([\![e]\!](x^*), [\![e]\!](y^*), [\![e]\!](x^* \sqcap y^*)) \in R^1_{\text{Nat}}$$

Therefore we have $(\llbracket e \rrbracket(x_1, \ldots, x_n), \llbracket e \rrbracket(y_1, \ldots, y_n), \llbracket e \rrbracket(x_1 \sqcap y_1), \ldots, (x_n \sqcap y_n)) \in R^1_{\mathrm{Nat}}$.

By the second part of R^1_{Nat}'s definition, we have:

$$\llbracket e \rrbracket(x_1 \sqcap y_1), \ldots, (x_n \sqcap y_n) = \llbracket e \rrbracket(x_1, \ldots x_n) \sqcap \llbracket e \rrbracket(y_1, \ldots, y_n)$$

□

Now we can use this to show the non definability of parallel or:

Corollary 4. *[Streicher, 2006] There are no PCF definable functions f of type* Nat $\to$ Nat $\to$ Nat *where:*

- $f\ 0\ \perp = 0$
- $f\ \perp\ 0 = 0$
- $f\ 1\ 1 = 1$

Proof. [Streicher, 2006] By contradiction. Assume there is a function f that satisfies the above conditions and it is definable in PCF. Then $f\ 0\ \perp = 0 = f\ \perp\ 0$, so using Lemma 17 we have

$$f\ \perp\ \perp = f\ (0 \sqcap \perp)\ (\perp \sqcap 0) = f\ 0\ \perp \sqcap f\ \perp\ 0 = 0 \sqcap 0 = 0$$

However we also have $f\ 1\ 1 = 1$ and by monotonicity of f we should have $f\ \perp\ \perp \sqsubseteq f\ 1\ 1$ which is $0 \sqsubseteq 1$ which is not in the relation for $\mathbb{N}_\perp$. Therefore we have a contradiction and there is no PCF definable f satisfying the constraints. □

The constraints in the above corollary characterize parallel or, so we cannot define parallel or in our PCF.

We have not used R^2 yet. R^2 is another relation with which we can show that por is non-definable in PCF, as a corollary of Lemma 17.

Corollary 5. *Parallel Or is not definable in PCF*

Proof. [Streicher, 2006] As all the λ-terms for PCF constructors are in R^2 and it is admissible, it is also the case that the denotation of any closed PCF term of type Nat $\to$ Nat $\to$ Nat is R^2 invariant. If this term is f, then we cannot satisfy all the por conditions (from the other corollary).

This is because $(\perp, 0, 1)$ and $(0, \perp, 1)$ are in R^2_{Nat} and as both tuples contain $\perp$, we should have $f(\perp, 0, 1)(0, \perp, 1) \in R^2_{\mathrm{Nat}}$. However, the expansion of this is $(f\ \perp\ 0, f\ 0\ \perp, f\ 1\ 1) = (0, 0, 1)$, so it is not in R^2_{Nat}. Therefore we cannot define an f that satisfies the constraints, so parallel or is not definable. □

Chapter 9

Evaluation

9.1 What did I learn/achieve?

Before I started this project I knew very little about Domain Theory, but I knew it featured prominently in research in programming language semantics. Now I know what a domain is (see 2.1), how to prove given structures are domains and how certain domains can be used to give models of programming languages (see Chapter 6).

I also knew nothing about Logical Relations (see 2.5), but now know how to construct them and prove properties using them.

I now also know that Type Safety is expressed by two lemmas (see 5.2.1 and 5.2.2) and how to prove them. Type Safety is an important theorem in programming language semantics, so now I can apply this knowledge to prove type safety for other type systems.

I previously proved Soundness (see 6.3) for an imperative language in my mini-project and now I have proved it for a new language and semantics, so I am applying what I learned in that project to a new situation. Type Safety and Soundness are common theorems in semantics, so I am also finding it easier to read new papers in the area now.

Finishing the Adequacy proof (see Theorem 10) was rewarding, as in the more simple language I previously studied, we proved a theorem called Completeness, that was less challenging to prove and required less mathematical background.

9.2 Evaluation of the Product and Presentation

My product is the proofs and analysis of these proofs. The proofs on domains mostly involved checking that the domain definition was satisfied when a new domain (such as the product of two domains and domain of continuous functions) was constructed. Proving that a given relation on a set was a partial order was quite easy as the properties they must satisfy were studied in the Maths Techniques module in 2nd year. Proving that all chains have limits was the most difficult bit. The proof of the fixpoint theorem (see 2.4) was the most difficult domain theory proof, but it had clear stages that made it easier to prove.

Many of the proofs on the Operational Semantics and typing rules were by induction. Having now completed more difficult proofs, they seem easier than they were at the time. The number of cases involved make the proofs very long, but each case uses the inductive hypothesis in a similar way.

The Adequacy proof was the hardest and took a lot of consideration to get right. Our first attempt to prove Adequacy was based on the following logical predicate (unary logical relation) defined inductively on types in the following way:

$$\mathrm{Adeq}_{\mathrm{Nat}} = \{e \mid \ \vdash e : \mathrm{Nat} \ \wedge ([\![e]\!] = n \Rightarrow e \mapsto^* \underline{n})\}$$

$$\mathrm{Adeq}_{A \to B} = \{e \mid \ \vdash e : A \to B \ \wedge \ \forall e' \in \mathrm{Adeq}_A (e\ e') \in \mathrm{Adeq}_B\}$$

It was also defined on typing contexts by induction on their size:

$$\mathrm{Adeq}_{Ctx}(\cdot) = \{<>\}$$

where $<>$ is the empty substitution and $\cdot$ is the empty context.

$$\mathrm{Adeq}_{Ctx}(\Gamma, A) = \{(\gamma, e/x) \mid \gamma \in \mathrm{Adeq}_{Ctx}(\Gamma) \ \wedge \ e \in \mathrm{Adeq}_A\}$$

Then we tried to prove the following Main Lemma:

Lemma 18. *If* $\Gamma \vdash e : A$ *and* $\gamma \in \mathrm{Adeq}_{Ctx}(\Gamma)$*, then* $[\gamma](e) \in \mathrm{Adeq}_A$

where $[\gamma] = [e_1/x_1, \ldots e_n/x_n]$.

We proved a similar Expansion Lemma to Lemma 7 and we used the following non termination lemma which is a bit like Lemma 6, but was entirely syntactic:

Lemma 19. *If* $[\![e]\!] = \perp$ *and* $\vdash e : A$ *and* $e \mapsto^{\infty}$*, then* $e \in \mathrm{Adeq}_A$

Proving the Main Lemma by induction on terms worked for every PCF expression except in the fixpoint case. We could attempt to prove this by induction, as from the typing rule of fixpoint we have $\Gamma, x : A \vdash e : A$. We could try to use induction with this and $[\gamma, e'/x]$, but when we evaluate $\text{fix}\, x : A.\ e : A$, we get $[\text{fix}\, x : A.\ e : A/x]e$, so e' must be $\text{fix}\, x : A.e : A$. This would require us to already know that the whole fixpoint expression is in Adeq!

Therefore we tried defining a fixpoint approximation like the one described in Lemma 9. We managed to prove the Main Lemma for this successfully and I presented this in my demonstration, but then it was noted that this does not actually prove the Main Lemma for the fixpoint term, it just proves it for the approximation!

What I had done was take the definition of the term, but I had not actually proved the lemma that it is from. To prove this lemma, our logical predicate would have to be equivalent to the definition of a 'computable' term. However the Main Lemma we want to prove is actually part *3* of the definition in Section 7.4. Therefore we were trying to prove part of the definition, so we could not use this approach.

As we got completely stuck with the syntactic predicate and we had proved the fixpoint theorem (Lemma 2.4), it made much more sense to consider using a binary logical relation, so we could actually use this theorem in our proof. Such a logical relation is defined in [Streicher, 2006] , albeit for a different version of PCF with constants, as explained in Section 7.1.

We adjusted this relation to work for our definition of PCF with constructors and used it in our proof. This gave proofs for the simpler terms that were not too different from our original proofs, and we could still use an Expansion Lemma like the one we proved for our original logical predicate. Being able to use the fixpoint theorem gave a much simpler proof than the one that uses the 'computable' terms, described in Section 7.4.

After proving Adequacy, we studied how the function por can be defined in the denotational model of PCF but not as a PCF term. This involved knowing more about logical relations and another property of domains, which was completely new to me, so we just reviewed the content of two chapters of [Streicher, 2006]. There were a few things we added to this review. The Main Lemma for the logical relation (see Lemma 10) did not include the variables case, so we explained this case too. We expanded on a trivial exercise given in the book, which was proving the PCF constructors were R-invariant in Section 8.3. We also had to rephrase these chapters for our definition of PCF and to do this, we reproved Lemma 12 to fit our PCF definition (particularly in the fixpoint case) and to use the definition of domains with chains (as opposed to directed sets). Finally, we explained the lemma on stable functions and its corollary that gives the non-definability of por in more detail than in the book (see Lemma 17).

9.3 Evaluation of the Process

My problem statement was to prove Adequacy for PCF. I completed this by defining the Operational and Denotational Semantics for PCF, then proving both directions of the theorem. Therefore I believe I have fully solved the problem stated in the introduction.

Everything we proved in the preceding sections is necessary to show that our domains are appropriate for modelling the language, its typing system is correct and that the semantics can be related. The report is ordered in such a way that no proof needs unspecified content given later in the report in order for the reader to understand it.

9.4 Review of Project Plan

I followed my project plan up until the later stages, when we got to Adequacy. This proof took much longer than expected, the reason being that I underestimated its difficulty. As I was unfamiliar with the Logical Relations technique, it was hard to know exactly how long it would take to prove a theorem in this way. I had no further work planned past this point, as Adequacy was the main goal of the report. This ensured I had enough time to fall back on this, as I had not over-planned my time with no room to manoeuvre.

As we had some time left at the end, we were able to discuss other approaches to the proof of Adequacy (in Section 7.4) and discuss some aspects of the denotational model (in Chapter 8).

9.5 Limitations and ideas for future work

There is no original work in my project, but I have proved theorems in Domain Theory from the basic definitions, selected a version of PCF for which I have developed Operational and Denotational Semantics for, with the help of my first supervisor, and related these by proving a theorem I had not encountered before, with the help of my second supervisor. Almost all of the mathematical theory in this report was entirely new to me and not taught in any undergraduate or postgraduate Computer Science module. Therefore I believe my report represents a substantial amount of work for a Computer Science student of my level.

Having now proved Adequacy for PCF in the Domain Theoretic model, I could consider other models of PCF and/or other languages. For example FPC, described in [Harper, 2016], is a language similar to PCF in which we have recursive types, so I could try to prove the same theorem with this addition to the possible types we can form. I could also construct the logical relation to describe Adequacy in a different way.

Chapter 10

Conclusion

In the report we defined the syntax of PCF and gave the typing rules, which all syntactically correct PCF programs must satisfy. We specified the Operational Semantics of PCF and we proved Type Safety, which states that all correctly typed PCF programs must have a valid evaluation in the Operational Semantics if they are expected to. We defined our Denotational Semantics of PCF and proved that the Operational Semantics are sound with respect to this. We proved the Adequacy theorem on our semantics and we discussed other approaches to the proof of Adequacy. We also discussed how there are functions in the denotational model that cannot be defined in PCF.

Bibliography

M. Gordon. *Evaluation and denotation of pure LISP programs: a worked example in semantics*. PhD thesis, The University of Edinburgh, 1973.

M. Gordon. From lcf to hol: A short history. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction*, pages 169–185. MIT Press, Cambridge, MA, USA, 2000.

C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016. 2nd. Edition.

C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

G. Hutton. Introduction to Domain Theory. Course Notes, 2014.

P. J Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4): 308–320, 1964.

P. J. Landin. Correspondence between algol 60 and church's lambda-notation: Part i. *Commununications of the ACM*, 8(2):89–101, February 1965.

S. Marlow et al. Haskell 2010 language report. Accessed online at http://https://www.haskell.org/definition/haskell2010.pdf, 9th September 2016, 2010.

R. Milner. Logic for Computable Functions: Description of a Machine Implementation. Technical report, Stanford University, Stanford, CA, USA, 1972a.

R. Milner. Implementation and Applications of Scott's Logic for Computable Functions. In *Proceedings of ACM Conference on Proving Assertions About Programs*, pages 1–6, New York, NY, USA, 1972b. ACM.

R. Milner. Models of LCF. Technical report, Stanford University, Stanford, CA, USA, 1973.

R. Milner. *The Definition of Standard ML: Revised.* MIT Press, 1997. ISBN 9780262631815.

R. Milner and R. Weyhrauch. Proving Compiler Correctness in a Mechanized Logic. *Machine Intelligence*, 7:51–70, 1972.

A. Murawski. Logical Relations. Course Notes from Midlands Graduate School, Birmingham, 2012.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

G. D. Plotkin. Lambda-definability and logical relations. Memorandum SAI–RM–4, University of Edinburgh, Edinburgh, Scotland, October 1973.

G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

G. D. Plotkin. A Structural Approach to Operational Semantics, 1981.

D. S. Scott. A Type-theoretical Alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1-2):411–440, December 1993.

R. Statman. Logical relations and the typed -calculus. *Information and Control*, 65(2):85 – 97, 1985. ISSN 0019-9958.

Thomas Streicher. *Domain-theoretic foundations of functional programming.* World Scientific, 2006.

W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32:198–212, 8 1967. ISSN 1943-5886.

A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38 – 94, 1994. ISSN 0890-5401.