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 to its operational semantics, using a logical relation. We also prove type safety and some necessary lemmas for this and soundness of PCF.

Contents

Chapter 1

Introduction

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 an additional element in the model to represent this.

We also want to compare different programs in the model, which we can do by an information ordering, where more defined programs are higher in the ordering and the element representing non-termination is the lowest. This ordering is a partial order.

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

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

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

The usefulness of the ordering will become apparent when we discuss recursion (see 2.5).

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 $\mathbb{N} \to X$, 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 n , 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

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

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

Domains can be defined using either of these structures, as although chains are examples of directed sets, they are enough to define domains:

2.2 Definition of a domain

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 (or 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.

2.3 Examples of Domains

2.3.1 Single Element Domain

In a single element domain the underlying set is $\{x\}$, so $\bot = x$ and $\sqsubseteq$ just contains the pair (x, x)

Now we must prove three conditions to show that $(\{x\}, \sqsubseteq)$ is a domain).

1. $\forall x \in \{x\}.\ x \sqsubseteq x$
   There is only one element, x, and $x \sqsubseteq x$ is in the ordering.
2. $\sqsubseteq$ **is a partial order**
   - Reflexivity
     The only element is x and $x \sqsubseteq x$ is in the ordering
   - Antisymmetry
     Any x and y must both be the element x, as it is the only possible element, so $x = x$.
   - Transitivity
     Any x, y and z must all be x and $x \sqsubseteq x$ is in the ordering.
3. **All chains must have a least upper bound**
   As x is the only possible element, all chains will be of the form

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

   Therefore let $\bigsqcup\{x\} = x$. Then we need an element z such that the following two statements are true, so let $z = \sqcup\{x\} = x$. Then:
   - $\forall i.x_i \sqsubseteq z$
     The only possible x_i is x, so we must have $x \sqsubseteq x$. This is in the ordering.
   - $\forall y.(\forall i.x_i \sqsubseteq y) \Rightarrow z \sqsubseteq y$
     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.

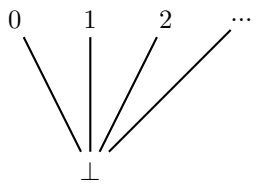
2.3.2 Flat Natural Numbers

A domain is a **flat** domain if the only ordering is between $\bot$ and the elements of the underlying set. In this example the underlying set is $\mathbb{N}$, the set of natural numbers:

In our domain, the set $X = \mathbb{N}_\perp$, so we have each number and a bottom element, $\perp$ that carries less information than a number. The only orderings we have are $\perp \sqsubseteq 0$, $\perp \sqsubseteq 1$, $\perp \sqsubseteq 2$, etc..., so 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 three conditions to show $(\mathbb{N}_\perp, \sqsubseteq)$ is a domain:

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

2. **Prove $\sqsubseteq$ is a partial order**
   For $\sqsubseteq$ to be a partial order, it must be reflexive, antisymmetric and transitive.

   - Reflexivity
     By definition of $\sqsubseteq$, as $\{(x, x) \mid x \in \mathbb{N}_\perp\}$ is a subset of $\sqsubseteq$.

   - Antisymmetry
     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 $x = y = n$.

   - Transitivity
     If $x = \perp$, a possibility for y is $y = n$. Then we must have $z = n$ for (y, z) to be in $\sqsubseteq$. Then we need $\perp \sqsubseteq n$, which we have, as we have $(\perp, n)$, for any $n \in \mathbb{N}$, defined in the relation. y could have also been $\perp$. Then we have both options for z. When $z = n$, we should have $\perp \sqsubseteq n$, which we have, as we have $(\perp, n)$ for any $n \in \mathbb{N}$ defined in the relation. When $z = \perp$, we just 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 $\sqsubseteq$.

3. **All chains must have a least upper bound**
   For any chain of elements of $\mathbb{N}_\perp$, we must prove there exists a $z \in \mathbb{N}_\perp$ satisfying the two statements. We must prove this by case analysis on the different chains:

   (a) $\perp \sqsubseteq \cdots \sqsubseteq \perp$
   For these chains, we never have any numbers, so $z = \perp$. The last element in the chain will always be $\perp$, so for every i we have $\perp \sqsubseteq \perp$. Therefore $\forall i.x_i \sqsubseteq \perp$. For the second statement, as every element is $\perp$, $x_i = \perp$ and $y = \perp$, we have $\perp \sqsubseteq \perp$ for $z \sqsubseteq y$. Therefore $\forall y.(\forall i.x_i \sqsubseteq y) \Rightarrow \perp \sqsubseteq y$ holds.

   (b) $n \sqsubseteq \cdots \sqsubseteq n$
   For the chains just containing 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.

   (c) $\perp \sqsubseteq \cdots \sqsubseteq \perp \sqsubseteq n \sqsubseteq \cdots \sqsubseteq n$
   For these chains, let $z = n$. The last element will be n. We have both $\perp \sqsubseteq n$ 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.

2.3.3 Product of domains

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

Now we must prove three conditions to show that $\mathbb{X} \times \mathbb{Y}$ is a domain:

1. $\forall (x, y) \in X \times Y.\ \perp \sqsubseteq (x, y)$
   Because $\mathbb{X}$ is a domain, we know $\forall x \in X.\ \perp_X \sqsubseteq_X x$ and because $\mathbb{Y}$ is a domain, we know $\forall y \in Y.\ \perp_Y \leq_Y y$.

   Therefore we have $\forall x, y.\ \perp_X \sqsubseteq_X x \wedge \perp_Y \leq_Y y$. This is the same as $\forall (x, y) \in X \times Y.(\perp_X, \perp_Y) \sqsubseteq (x, y)$

2. $\sqsubseteq$ **is a partial order**

   - Reflexivity
     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 $(x, y) \sqsubseteq (x, y)$.

- Antisymmetry
  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 $(x, y) = (x', y')$.

- Transitivity
  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 $(x, y) \sqsubseteq (x'', y'')$.

3. **All chains have a least upper bound**
   Chains of $\mathbb{X} \times \mathbb{Y}$ will be of the form:

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

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

   So we need an element z that makes the following two statements true. Let $z = \sqcup(x_i, y_i) = (\sqcup x_i, \sqcup y_i)$. Then:

   - $\exists z \in X_\perp \times Y_\perp . \forall i . (x_i, y_i) \sqsubseteq z$
     As $\mathbb{X}$ and $\mathbb{Y}$ are domains, we have $\forall i . x_i \sqsubseteq_X \sqcup x_i$ and $\forall i . y_i \leq_Y \sqcup y_i$. Therefore, for any (x, y) we have $\forall i . (x_i, y_i) \sqsubseteq (\sqcup x_i, \sqcup y_i)$.

   - $\exists z \in X \times Y . \forall (x', y') .\ (\forall i . (x_i, y_i) \sqsubseteq (x', y')) \Rightarrow z \sqsubseteq (x', y')$
     As $\mathbb{X}$ and $\mathbb{Y}$ are domains, we have $\forall x' .\ (\forall i .\ x_i \sqsubseteq_X x') \Rightarrow \sqcup x_i \sqsubseteq_X x'$ and $\forall y' .\ (\forall i .\ y_i \leq_Y y') \Rightarrow \sqcup y_i \leq_Y y'$.

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

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

2.4 Continuous and Monotone 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.*

Given a chain $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$, we can form the chain $f(x_0) \sqsubseteq f(x_1) \sqsubseteq \ldots$
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. Proof?

2.4.1 Domain of Continous Functions

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

Given two domains, $\mathbb{X} = (X, \bot_X, \sqsubseteq_X)$ and $\mathbb{Y} = (Y, \bot_Y, \leq_y)$, we can form the set $\text{Cont}(X, Y) = \{f : X \to Y\}$, where:

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

$\bot_{X \to Y}$ is defined as the function $\bot = \lambda x.\bot(x)$, the function that loops on all inputs. The output of this function will always be $\bot$, 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 $(\text{Cont}(X, Y), \bot_{X \to Y}, \sqsubseteq_C)$. Now we must prove the three conditions:

1. $\forall f \in \text{Cont}(X, Y).\ \bot \sqsubseteq_C f$
   So 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 $\bot \sqsubseteq_C f$.

2. **Prove $\sqsubseteq_C$ is a partial order**
   For $\sqsubseteq_C$ to be a partial order, it must be reflexive, antisymmetric and transitive. As $\mathbb{Y}$ is a domain, we know that $\leq_Y$ is a partial order.

   - Reflexivity
     We need to prove that $\forall f \in \text{Cont}(X, Y).\ f \sqsubseteq_C f$. We can rewrite this using the definition of $\sqsubseteq_C$ to get

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

     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 $f \in \text{Cont}(X, Y)$.

   - Antisymmetry
     We need to prove that $\forall f, g \in \text{Cont}(X, Y).\ f \sqsubseteq_C g \wedge g \sqsubseteq_C f \Rightarrow f = g$. Rewriting this using the definition of $\sqsubseteq_C$ gives us

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

     $\leq_Y$ 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.

   - Transitivity
     We need to prove that $\forall f, g, h \in \text{Cont}(X, Y).\ f \sqsubseteq_C g \wedge g \sqsubseteq_C h \Rightarrow f \sqsubseteq_C h$. Rewriting this using the definition of $\sqsubseteq_C$ gives us

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

     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.

3. **All chains have a least upper bound (limit)**

   For any chain f_n, we must have a $z \in \text{Cont}(X, Y)$ such that:

   - $\forall i. f_i \sqsubseteq_C z$
   - $\forall g. (\forall i. f_i \sqsubseteq_C g) \Rightarrow z \sqsubseteq_C g$

   Let $z = \lambda x.\ \sqcup^Y f_i(x)$, where $\sqcup^Y f_i(x)$ is the limit of the chain (in $\mathbb{Y}$) obtained by applying the functions in $\text{Cont}(X, Y)$ to a certain element $x \in X$.

A chain of functions will be a chain of elements of the set $\text{Cont}(X, Y)$, for example

$$f_1 \sqsubseteq_C f_2 \sqsubseteq_C \ldots \sqsubseteq_C \sqcup f_n$$

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

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

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

$$z = \lambda x.\ \sqcup^Y f_i(x)$$

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 $\forall g.(\forall i.f_i \sqsubseteq_C g) \Rightarrow z \sqsubseteq_C g$

2.5 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 given by repeated iterations of a recursive function. If an input of the function is computed using $f^n(\bot)$ 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 1. *Every continuous function $f : X \to X$ has a least fixpoint, which is the limit of the chain $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$*

Proof. Define the fixpoint function $fix(f) \equiv \sqcup f^i(\bot)$. This is the limit of the chain in the theorem. We know this limit exists because as f is continuous, $(X, \sqsubseteq, \bot)$ must form a domain, and by the definition of domain, all chains of X have a limit.

$\sqcup f^i(\bot)$ **is a fixpoint** For the limit to be a fixpoint we must have $f(\sqcup f^i(\bot)) = \sqcup f^i(\bot)$. As f is continuous, we have $f(\sqcup f^i(\bot)) = \sqcup f(f^i(\bot)) = \sqcup f^{i+1}(\bot)$. The chain formed by f^{i+1} is $f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$. This is the same as our original chain, but without $\bot$ at the start. Because X is a domain, we know that X has a least element, so $\forall x \in X.\ \bot \sqsubseteq x$. 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 $\sqcup f^{i+1}(\bot) = \sqcup f^i(\bot)$.

$\sqcup f^i(\bot)$ **is the least fixpoint** Let x be an element of our chain such that $fix(x) = x$. Then for $\sqcup f^i(\bot)$ to be the least fixpoint, we must have $\sqcup f^i(\bot) \sqsubseteq x$, so x is an upper bound that is higher than $\sqcup f^i(\bot)$. To prove this, 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 continous, 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 $\sqcup f^i(\bot)$ is a least upper bound, we know that $\forall x \in X.\ \forall n.\ (f^n(\bot) \sqsubseteq x) \Rightarrow \sqcup f^i(\bot) \sqsubseteq x$. We have just proved the left hand side of this, so we now have $\sqcup f^i(\bot) \sqsubseteq x$.

Now we have proved that $\sqcup f^i(\bot)$ is the least fixpoint of f. □

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

2.6 Logical Relations

Logical relations are a proof technique that is used for proving properties that cannot be proved by structural induction alone.

They have been used to prove Strong Normalisation of the Simply Typed λ calculus (i.e. that every expression terminates), Original version?, Type Safety and Program Equivalence.

more references

2.6.1 Definition

We can define logical relations on program terms, by defining relations on each type individually. The following definition is as in [Murawski, 2012]:

Define a set of types, $Type$, which is formed by:

$$\theta ::= o \mid \theta \to \theta$$

such that $[\![o]\!] = X$ and $[\![\theta_1 \to \theta_2]\!] = [\![\theta_1]\!] \to [\![\theta_2]\!]$

Definition 7. *An n-ary **logical relation** is a family $\mathcal{R} = \{R_\theta\}_{\theta \in Type}$ of n-ary relations such that $R_\theta \subseteq [\![\theta]\!] \times \ldots [\![\theta]\!]$ (i.e. an n-tuple) for any θ and*

$$R_{\theta_1 \to \theta_2}(f_1, \ldots f_n) \Leftrightarrow$$

for all $(d_1, \ldots d_n) \in [\![\theta_1]\!]^n$, if $R_{\theta_1}(d_1, \ldots d_n)$ then $R_{\theta_2}(f_1(d_1), \ldots f_n(d_n))$

2.6.2 Examples and Non-Examples

Applying certain restrictions on R_o restricts the relations we can define on function types, for example:

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

Union of two logical relations Given two logical relations $\{R_\theta\}$ and $\{S_\theta\}$ of the same arity, we could try to form a logical relation $\{R_\theta \cup S_\theta\}$. This is **not** a logical relation. For example, if we have $(f_1, \ldots, f_n) \in R_{\theta_1 \to \theta_2}$ (and not in $S_{\theta_1 \to \theta_2}$) and $(d_1, \ldots, d_n) \in S_{\theta_1}$, (and not in $R_{\theta_1 \to \theta_2}$) then $(f_1(d_n), \ldots, f_n(d_n))$ cannot be in either $\{R_\theta\}$ or $\{S_\theta\}$.

Intersection of two logical relations Therefore if we restrict the logical relation to only contain tuples that are in both $\{R_\theta\}$ and $\{S_\theta\}$ 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 $\{R_\theta\}$ of arity 3 and $\{S_\theta\}$ of arity 5, then have, for example, $(f, g, h) \in \{R_\theta\}$ and $(f, g, h, i, j) \in \{S_\theta\}$, then we cannot say that (f, g, h) is in both, as this would not be in $\{S_\theta\}$ anymore. Therefore, this would not be a logical relation.

Composition of two binary logical relations We define $\{R_\theta; S_\theta\}$ in the following way:

$$(R_\theta; S_\theta)(x, y) \Leftrightarrow \exists z.\ (R_\theta(x, z)\ \wedge\ S_\theta(z, y))$$

So for base type we have:

$$(R_o; S_o)(x, z) \Leftrightarrow \exists y.\ (R_o(x, y)\ \wedge\ S_o(y, z))$$

and for functions we have

$$(R_\theta; S_\theta)(f, h) \Leftrightarrow \exists g.\ (R_\theta(f, g)\ \wedge\ S_\theta(g, h))$$

where for any $(x, z) \in (R_o; S_o)$ we have

$$(R_\theta(f(x), g(y))\ \wedge\ S_\theta(g(y), h(z)))$$

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.**

2.6.3 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 (or sometimes the Fundamental Property/Theorem/Lemma). For program terms, we define the main lemma in the following way, (as in [Murawski, 2012]:
Lemma 1. *Let* $\Gamma \vdash M : \theta$ *where* $\Gamma = \{x_1 : \theta_1, \ldots x_m : \theta_m\}$ *and* $f = [\![\Gamma \vdash M : \theta]\!]$. *Suppose* $\{R_\theta\}$ *is an n-ary logical relation and*

$$\gamma_i = (d_{i1} \ldots d_{im}) \in [\![\theta_1]\!] \times \cdots \times [\![\theta_m]\!]$$

where $i = 1, \ldots n$, *are such that* $R_{\theta_j}(d_{1j} \ldots d_{nj})(1 \leq j \leq m)$. *Then*

$$R_\theta(f\gamma_1, \ldots, f\gamma_n)$$

This says that for a well typed term M, that has a denotation equal to a function f, if we have a logical relation for types, then any possible set of substitutions for the variables in Γ (of which there are n), that are in that term will be a tuple, γ_i, in the relation, and applying f to each of these tuples gives another tuple that will be in the relation. The resulting tuple will contain the result of f for each substitution.

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 various versions of PCF in the literature, where some use Booleans and Natural Numbers for the base types. Here we just use Natural numbers, where 0 is accepted as false and any non zero number will be true. We define our types using the following grammar:

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

3.1.2 Expressions

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

case takes an expression e, which we assume is a numerical value, then 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 which applies a function e 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:

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

3.2 Type System for PCF

Γ is a context for types. It 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 $\Gamma \vdash e$ for the context associated with an expression e.

Therefore we need a typing rule for each expression in PCF. The typing rules are given as inference rules, where our assumptions are on the top and the conclusion underneath:

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

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

As z is a constant, 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.

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

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 $x : \text{Nat}$ to the context of e_S.

$$\text{APPLICATION } \frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e \ e' : B} \qquad \text{ABSTRACTION } \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.e : A \to B}$$

For application, e must have a function type and e' must have the same type as the domain of e. Then we end up with an 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 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 exactly the same as λ abstraction, where the x is bound to the fixpoint 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 derivation of this judgement.

Chapter 4

Operational Semantics of PCF

We define the operational semantics of PCF using 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 the transition relation $e \mapsto e'$ 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:

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

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

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 (zero, successors of values, and lambda abstractions). This ensures that there is only one possible rule to apply to any expression:

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

This says that we substitute e' for x in the expression e.

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

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

This says that if $e = z$, then we get e_0. If $e = s(v)$, then we get 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.

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

This says that we replace the bound 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:

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

We give an example of this behaviour below:

4.1 Example programs in PCF

4.1.1 Addition

We can define mathematical operators on numbers as recursive functions. For example addition is the following function:

$$\text{add}\, 0\ y = y$$
$$\text{add}\, s(n)\ y = s(\text{add}\, n\ y)$$

We can write this as a λ abstraction:

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

This is a recursive function, so must be the fixpoint of another function A:

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

Therefore we can define add as add $x\ y = (\text{fix}\, A)\ x\ y$

So we write this in PCF as:

$$\text{fix}\, f : \text{Nat} \to \text{Nat} \to \text{Nat}\,.\; A : \text{Nat} \to \text{Nat} \to \text{Nat}$$

.

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

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

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

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

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 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 before proving Type Safety.

5.1.1 Weakening

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

Theorem 2. *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 exists a derivation tree for the conclusion. We can use the inductive hypothesis

$$\Gamma \vdash e : A \Rightarrow \Gamma, x : C \vdash e : A$$

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 is also a set of (variable, type) pairs. Therefore $\Gamma, x : C$ can be thought of as the set $\Gamma \cup \{(x, C)\}$. Therefore we define the function $(\Gamma, x : C)$, where $(\Gamma, x : C)(y) = A$ and for any other 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 a derivation tree for $\Gamma, x : C \vdash y : A$, 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 for any Γ, we 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 a derivation tree for $\Gamma, x : C \vdash z : \text{Nat}$

z is a term of ground type, so can only have the type Nat, so we have proved this for all possible types of z.

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 a derivation tree that gives us $\Gamma, x : C \vdash s(e) : \text{Nat}$, so weakening holds for the successor function, as it is only defined on terms of type Nat.

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

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

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:

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

Now we have a derivation tree that gives us $\Gamma, x : C \vdash$ case $(e, z \to e_0, s(y) \to e_S) \ : A$, so weakening holds for the case expression.

Application We assume $\Gamma \vdash e_0 \ e_1$ 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 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.

Abstraction We rename x to y using α equivalence. We assume $\Gamma \vdash \lambda y : A.\ e : A$, 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 a derivation tree that gives us $\Gamma, x : C \vdash \lambda y : A.\ e : A \to B$, 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

Before we prove the Substitution Theorem, we must actually define the Substitution Rules for PCF, which are all instances of $[e/x]e'$. This 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 end up with $\lambda x + s(x)$ 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 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:

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

But we must be careful with variable capture, as the derivation of e_S is $\Gamma, x : Nat \vdash e_S$. 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 $\Gamma, x : Nat$. 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:

$(\text{case } (e', z \to e_0, s(x) \to e_s)) =$

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

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

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 in the free variables of 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 when the bound x is equal to x, for which we just rewrite the function using α equivalence. Therefore this rule does nothing:

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

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

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

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

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 in the context Γ and an expression e' in the context Γ including a variable $x : A$, then the expression obtained by substituting e for x in e' will be derivable in the context Γ:

Theorem 3. *If* $\Gamma \vdash e : A$ *and* $\Gamma, x : A \vdash e' : C$ *then* $\Gamma \vdash [e/x]e' : C$

Proof. We can slso prove this by induction on derivation trees. We can rewrite the theorem as "If $D :: \Gamma \vdash e : A$ and $E :: \Gamma, x : A \vdash e' : C$ then $F :: \Gamma \vdash [e/x]e' : C$, where D, E and F are derivation trees for each of the well typed terms. We use induction on the tree E, as we are interested in the possible expressions e' could be. We can use the inductive hypothesis:

$$(\Gamma \vdash e : A\ \wedge \Gamma, x : A \vdash e' : C) \Rightarrow\ \Gamma \vdash [e/x]e' : C$$

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 is 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 type 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, 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 $\Gamma, x : A \vdash \text{case } (e', z \to e_0, s(y) \to e_S) \ : C$. The second derivation tree gives

us the subtrees $\Gamma, x : A \vdash e' : \text{Nat}$, $\Gamma, x : A \vdash e_0 : C$ and $\Gamma, y : Nat, x : A \vdash e_s : C$. 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 weakening, 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:

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

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

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

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

which contains the subtrees $\Gamma, x : A \vdash e_0 : A \to B$ and $\Gamma, x : A \vdash e_1 : A$. Combining each of these trees with $\Gamma \vdash e : A$ and the inductive hypothesis gives us $\Gamma \vdash [e/x]e_0 : A \to B$ and $\Gamma \vdash [e/x]e_1 : A$. 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.

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

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

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

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

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

.

Now there are two cases:

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

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 $fix\ z : C.[e/x]([z/y]e')$. 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 Lemmas

Now we must prove the two type safety lemmas we previously discussed:

5.2.1 Type Preservation

Type preservation says that if an expression e has type A in a context Γ, and it evaluates in one step to e', then e' must also have type A in Γ (i.e. we get the same type at the end as at the start):

Theorem 4. *If* $\Gamma \vdash e : A$ *and* $e \mapsto e'$*, then* $\Gamma \vdash e' : A$

Proof. We can prove this by induction on derivation trees, so we rewrite the theorem as "If $D :: \Gamma \vdash e : A$ and $E :: e \mapsto e'$ (in one step) then $\exists . F :: \Gamma \vdash e' : A$:

There will be a derivation tree E for each rule in the operational semantics, where D is the tree for the typing derivation of the first half of the rule represented by E.

Therefore we can check this statement for every evaluation rule on every possible expression:

Vsriables 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 have the following tree for D , given by the typing rule of successor:

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

The tree for E will be 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 use the typing rule for case as D:

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

and congruence evaluation rule for case as E:

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

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 $\Gamma \vdash \textit{case } (e', z \to e_0, s(x) \to e_S) : A$

2. If $e = \text{case } (z, z \to e_0, s(x) \to e_S)$, then D is

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

and E is the tree for the evaluation rule case $(z, z \to e_0, s(x) \to e_S) \mapsto e_0$. $\Gamma \vdash e_0 : A$ is a subtree of D, so we already have the tree for $\Gamma \vdash e_0 : A$

3. If $e = \text{case } (s(v), z \to e_0, s(x) \to e_S)$, then D is

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

and E is the tree for the evaluation rule case $(z, s(v), \to e_0, s(x) \to e_S) \mapsto [v/x]e_s$.

We have the subtrees for $\Gamma \vdash v : \text{Nat}$, $\Gamma \vdash s(v) : \text{Nat}$, $\Gamma \vdash e_0 : A$, $\Gamma, x : \text{Nat} \vdash e_s : \text{Nat}$ from the tree D.

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 use the congruence evaluation rule for application, which gives us the following tree for D:

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

and the following tree for E:

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

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 for D:

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

And the tree for $(\lambda x : A).e\ e' \mapsto [e'/x]e$ for E. 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

Fixpoint When $e = \text{fix}\ x : A.e$, we have the following tree for D:

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

and E is the tree for the rule fix $x : A.\ e \mapsto [\text{fix}\ x : A.e/x]e$. 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. □

5.2.2 Type Progress

Type progress says that if an expression e has type A in a context Γ, then it must evaluate to another expression e' in one step, or be a value (so 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.(So note this is not saying it terminates, or that it has normal form.):

Theorem 5. *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, so we rewrite the theorem as "If $D :: \vdash e : A$ then $E :: e \mapsto e'$ or e is a value".

When we have a derivation tree D for a closed term e, we either get a derivation tree E, for evaluating it in one step to another expression, or e is a value.

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 can 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:

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

,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 $(e', z \to e_0, s(x) \to e_S) \ : A$. Therefore our original expression maps to some e', so we have case $(e, z \mapsto e_0, s(x) \mapsto e_s) \mapsto e'$ for some e'
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 $(z, z \to e_0, s(x) \to e_S)$ 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 $(s(v), z \to e_0, s(x) \to e_S)$ 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 $(e, z \to e_0, s(x) \to e_S) \mapsto e'$ 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 $x : A$ ane $e : A$

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 [fix $x : A.e/x]e$. Therefore we know that fix $x : A.\ e$ always maps to another expression, so fix $x : A.\ e$ maps to another expression e' in one step, or fix $x : A.\ e$ is a value is true.

Now we have proved type progress for all possible expressions. □

5.3 Type Safety

Type Safety can be proved as a corollary of the two lemmas we have just proved. Usually this is just assumed because of the two lemmas.

The theorem for type safety says that an closed term either evaluates to a value in a finite number of steps or loops forever:
Theorem 6. *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. Now we use the domain theory we discussed in *(Section 2.1)*.

6.1 Denotation of Types

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

$$[\![-]\!] : Type \to Domain$$

that maps a type to a domain. We have two possible ways to define a type, so there are two different constructions of domains we use:

1. The type of Natural numbers is the ground type, so they are modelled by a single domain. We use the flat domain of Natural numbers, where $\perp$ represents a term that does not terminate:

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

2. Function types are formed of other types. We model them using the domain of continuous functions.

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

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

6.2 Denotation of Typing Contexts

In our Denotational Semantics ,we also have a domain for the typing contexts, 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 domain, which we already proved is a domain (in Section 2.3.3).

The empty context is given by

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

the single element domain, which we also previously proved is a domain (in Section 2.3.1).

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

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

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 $\gamma \in [\![\Gamma]\!]_{Ctx}$.

6.3 Denotation of well typed terms

The Denotational Semantics maps the terms of PCF to a domain. Given a well typed term $\Gamma \vdash e : A$ we have

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

So $[\![\Gamma \vdash e : A]\!]\gamma$ gives us an element of $[\![A]\!]$, the domain that models e's type. We can define this on each possible value of e individually:

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

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

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 γ is:

$$[\![\Gamma \vdash z : Nat]\!]\gamma = 0$$

Successor When $\Gamma \vdash s(e) : \text{Nat}$ is a well typed term, then so is $\Gamma \vdash e : \text{Nat}$, 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 $\perp$:

$[\![\Gamma \vdash s(e) : \text{Nat}]\!]\gamma = \text{Let } v = [\![\Gamma \vdash e : \text{Nat}]\!]\gamma \text{ in}$

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

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

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

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

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, $\lambda x \in X.\bot_Y$. 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:

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

λ **abstraction** For λ abstraction, by its typing rule, we already have a denotation for $[\![\Gamma, x : A \vdash e : B]\!]\gamma$. 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 the argument to our function, which is $(\gamma, a/x)$:

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

Fixpoint For fixpoint, by its typing rule we already have a denotation for $[\![\Gamma, x : A \vdash e : A]\!]\gamma$ 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, $fix_{[\![A]\!]}$, 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:

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

6.4 Substitution Theorem

The following theorem says that given a well typed expression e and another expression e', 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 7. *If* $\Gamma \vdash e : A$ *and* $\Gamma, x : A \vdash e' : C$ *and* $\gamma \in [\![\Gamma]\!]$*, then* $[\![\Gamma \vdash [e/x]e' : C]\!]\gamma = [\![\Gamma, x : A \vdash e' : C]\!](\gamma, [\![\Gamma \vdash e : A]\!]\gamma/x)$

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 $[\![\Gamma \vdash [e/x]x : A]\!]\gamma = [\![\Gamma \vdash e : A]\!]\gamma$, from the substitution rule.

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

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

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

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

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

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

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

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

Therefore $[\![\Gamma \vdash [e/x]s(e') : \mathrm{Nat}]\!]\gamma = [\![\Gamma, x : A \vdash s(e') : C]\!](\gamma, [\![\Gamma \vdash e : A]\!]\gamma/x)$

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

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

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

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

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

Application Using the substitution rule for application, $[\![\Gamma \vdash [e/x](e_0\ e_1) : B]\!]\gamma = [\![\Gamma \vdash [e/x]e_0([e/x]e_1) : B]\!]\gamma$. 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 $f = [\![\Gamma, x : A \vdash e_0 : A \to B]\!](\gamma, [\![\Gamma \vdash e : A]\!]\gamma/x)$ in

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

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

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

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

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

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

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

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

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

$$fix_{[\![C]\!]}(\lambda c \in [\![C]\!].[\![\Gamma, y : C \vdash [e/x]e' : C]\!](\gamma, c/y)$$

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

$$fix_{[\![C]\!]}(\lambda c \in [\![C]\!].[\![\Gamma, y : C, x : A \vdash e' : C]\!](\gamma, c/y, [\![\Gamma \vdash e : A]\!]\gamma/x)$$

This is also the definition of $[\![\Gamma, x : A \vdash (fix\ y : C.e' : C)]\!](\gamma, [\![\Gamma \vdash e : C]\!]\gamma/x)$.

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

Now we have proved the theorem for every case of e'. □

6.5 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 be equal 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 8. *If* $\Gamma \vdash e : A$ *and* $e \mapsto e'$ *and* $\gamma \in [\![\Gamma]\!]$*, then* $[\![\Gamma \vdash e : A]\!]\gamma = [\![\Gamma \vdash e' : A]\!]\gamma$

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 $[\![\Gamma \vdash e : A]\!]\gamma = [\![\Gamma \vdash e' : A]\!]\gamma$.

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

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

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

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

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 $[\![\Gamma \vdash e : \text{Nat}]\!]\gamma = [\![\Gamma \vdash e' : \text{Nat}]\!]\gamma$.

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

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

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

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

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

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

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

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

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

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

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

where $n = [\![\Gamma \vdash v : Nat]\!]\gamma$.

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 $n = [\![\Gamma \vdash v : \text{Nat}]\!]\gamma$. With this we can simplify the definition of the expression to $[\![\Gamma, y : \text{Nat} \vdash e_S : C]\!](\gamma, n/y)$

This is the same as:

$$[\![\Gamma, y : Nat \vdash e_S : C]\!](\gamma, [\![\Gamma \vdash v : Nat]\!]\gamma/y)$$

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

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 $[\![\Gamma \vdash e_0\ e_1 : A]\!]\gamma = [\![\Gamma' \vdash e_0'\ e_1 : A]\!]\gamma$.

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

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

   $$\text{Let } v = [\![\Gamma \vdash e_1 : A]\!]\gamma$$
   $$\text{in } f(v)$$

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

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

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

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

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

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

   By the substitution lemma, this is the same as $[\![\Gamma \vdash [e'/x]e]\!]\gamma$

λ **abstraction** is a value, so has no evaluation rules. Therefore there are no cases.

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

$$(\lambda a \in [\![A]\!].[\![\Gamma, x : A \vdash e : A]\!](\gamma, a/x))[fix_{[\![A]\!]}(\lambda a \in [\![A]\!].[\![\Gamma, x : A \vdash e : A]\!](\gamma, a/x))]$$

which is equal to:

$$[\![\Gamma, x : A \vdash e : A]\!](\gamma, fix_{[\![A]\!]}(\lambda a \in [\![A]\!].[\![\Gamma, x : A \vdash e : A]\!](\gamma, a/x))/x)$$

which is equal to:

$$[\![\Gamma, x : A \vdash e : A]\!](\gamma, [\![\Gamma \vdash \text{fix } x : A.e : A]\!]\gamma/x)$$

Using the substitution lemma, this is the same as

$$[\![\Gamma \vdash [fix\ x : A.e/x]e : A]\!]\gamma$$

The evaluation rule is

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

so this is the denotation we need. □

Chapter 7

Adequacy

Now that we have proved Correctness, (as Theorem 8) we have proved half of Adequacy, as it is the following theorem:

Theorem 9. *If* $\vdash e : \text{Nat}$ *(i.e. e is a closed term of type* Nat*), then* $[\![e]\!] = n \Leftrightarrow e \mapsto^* \underline{n}$

where $[\![e]\!] = [\![\vdash e : \text{Nat}]\!]$ and $\underline{n}$ represents the numeral n, instead of the natural number n.

The right to left direction of Theorem 9 is a corollary of the correctness proof:

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

Proof. Rewrite $e \mapsto^* \underline{n}$ as $e \mapsto e_0 \mapsto \cdots \mapsto e_m \mapsto \underline{n}$, for any $n \geq 0$. Applying correctness 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 $[\![z]\!] = 0$ and z is the syntactic representation of 0.

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

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

□

7.1 Logical Relation

For the other direction of Theorem 9, we want to show that if $[\![e]\!] = n$ then $e \mapsto^* n$. We cannot prove this by induction, so we need to define a **logical predicate** to use in the proof,

inductively on types, which is the following:

$$\text{Adeq}_{\text{Nat}} = \{e \mid \vdash e : \text{Nat} \ \wedge \ (\llbracket e \rrbracket = n \Rightarrow e \mapsto^* \underline{n})\}$$

$$\text{Adeq}_{A \to B} = \{e \mid \vdash e : A \to B \ \wedge \ \forall e' \in \text{Adeq}_A (e \ e') \in \text{Adeq}_B\}$$

Now to prove adequacy, we just prove that every well typed term is in Adeq, so we also defined Adeq on typing contexts:

$$\text{Adeq}_{Ctx}(\cdot) = \{<>\}$$

where $<>$ is the empty substitution and $\cdot$ is the empty context.

$$\text{Adeq}_{Ctx}(\Gamma, A) = \{(\gamma, e/x) \mid \gamma \in \text{Adeq}_{Ctx}(\Gamma) \ \wedge \ e \in \text{Adeq}_A\}$$

For example, if we have the context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ then:

$$\text{Adeq}_{Ctx}(x_1 : A_1, \ldots, x_n : A_n) = \{(e_1/x_1, \ldots, e_n/x_n) \mid e_i \in \text{Adeq}_{A_i}\}$$

From this we know that:

- The set of free variables in the expression $[\gamma](e)$, $FV([\gamma](e)) = \emptyset$, because for any expression e, $FV(e) \subseteq \Gamma$ and $[\gamma]$ substitutes all the free variables in Γ with expressions.
- If $\gamma \in \text{Adeq}_{Ctx}(\Gamma)$ then $\gamma(x_i)$ has no free variables, because every expression in γ that substitutes some x_i is in Adeq_{A_i}, so is a closed term.

Now we have this, we can prove the **Fundamental Lemma**, which is the following:

Lemma 2. *If* $\Gamma \vdash e : A$ *and* $\gamma \in \text{Adeq}_{Ctx}(\Gamma)$*, then* $[\gamma](e) \in \text{Adeq}_A$

7.1.1 Substitution Function

$[\gamma](e)$ is the expression obtained by applying a substitution γ to an expression e. We can define it inductively in the following way:

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

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

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 $\gamma = e_1/x_1, \ldots e_n/x_n$, we have:

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

7.2 Expansion Lemma

To prove this lemma, we need another lemma for the λ-abstraction case. This lemma is called the **Expansion Lemma**:

Lemma 3. *If* $\vdash e : A$ *and* $e \mapsto e'$ *and* $e' \in \text{Adeq}_A$ *then* $e \in \text{Adeq}_A$

Proof. By induction on types.

The base case will be when $\vdash e : \text{Nat}$. We have $e' \in \text{Adeq}_{\text{Nat}}$, so we have $\vdash e' : \text{Nat}$ and $[\![e']\!] = n \Rightarrow e' \mapsto^* \underline{n}$. We need to show that $e \in \text{Adeq}_{\text{Nat}}$. We already have $\vdash e : \text{Nat}$ as it is one of our assumptions, so we just prove $[\![e]\!] = n \Rightarrow e \mapsto^* \underline{n}$.

Assume $[\![e]\!] = n$. Correctness in the empty context is $e \mapsto e' \Rightarrow [\![e]\!] = [\![e']\!]$, so we use this to get $[\![e]\!] = [\![e']\!] = n$.

We can use $[\![e']\!] = n$ to get $e' \mapsto^* \underline{n}$ from $e' \in \text{Adeq}_{\text{Nat}}$. As $e \mapsto e'$ was an assumption, we now have $e \mapsto e' \ \wedge \ e' \mapsto^* \underline{n}$, so we have $e \mapsto^* \underline{n}$. Therefore $e \in \text{Adeq}_{\text{Nat}}$.

The inductive case will be when $\vdash e : A \to B$. We have $e \mapsto e'$ and $e' \in \text{Adeq}_{A\to B}$ and we need to show that $e \in \text{Adeq}_{A\to B}$. We already have $\vdash e : A \to B$, as it is one of our assumptions, so we just prove $\forall a \in \text{Adeq}_A \,.\, e\ a \in \text{Adeq}_B$.

Let $a : A$ be an expression such that $a \in \mathit{Good}_A$. Then we have $e'\ a \in \mathit{Good}_B$ from $e' \in \mathit{Good}_A$. We use $\vdash e : A \to B$ and $\vdash a : A$ (obtained from $a \in \mathit{Good}_A$) with the typing rule for function application to get $\vdash e\ a : B$. We use the congruence rule on $e \mapsto e'$ to get $e\ a \mapsto e'\ a$.

Now we can apply the inductive hypothesis on $\vdash e\ a : B$, $e\ a \mapsto e'\ a$ and $e'\ a \in \mathit{Good}_B$ to get $e\ a \in \mathit{Good}_B$

Therefore $\forall a \in \mathit{Good}_A . e\ a \in \mathit{Good}_B$, so $e \in \mathit{Good}_{A\to B}$

Now we have proved all of the cases, so we know the lemma holds for expressions of any type A.

□

7.3 Lemmas for Main Lemma

We also prove some other lemmas to help in the main lemma. The first covers the cases where e does not terminate, to prove they are still in Adeq:

7.3.1 Non-Termination Lemma

Lemma 4. *If* $[\![e]\!] = \bot$ *and* $\vdash e : A$ *and* $e \mapsto^\infty$, *then* $e \in \text{Adeq}_A$

Proof. By induction on types. The base case will be for the type Nat. We need to show that $\vdash e : \text{Nat}$ (which we already have as an assumption) and $[\![e]\!] = n \Rightarrow e \mapsto^* \underline{n}$.

We can rewrite this as $[\![e]\!] \neq n \ \vee\ e \mapsto^* \underline{n}$

As we have $[\![e]\!] = \bot$ as an assumption and $\bot \notin \mathbb{N}$, we know that $[\![e]\!] \neq n$ for any $n \in \mathbb{N}$. Therefore $[\![e]\!] \neq n \ \vee\ e \mapsto^* \underline{n}$, so $[\![e]\!] = n \Rightarrow e \mapsto^* \underline{n}$ and $e \in \text{Adeq}_{\text{Nat}}$.

For the inductive case, we need to show that $e \in \text{Adeq}_{A\to B}$, so we need $\vdash e : A \to B$ (which we have as an assumption) and $\forall e' \in \text{Adeq}_A \,.\, e\ e' \in \text{Adeq}_B$.

Let $e' \in \text{Adeq}_A$. As $[\![e]\!] = \bot$ is the bottom element of $[\![A]\!] \to [\![B]\!]$, this is the same as $\lambda a \in A.\bot_B$. We can apply the inductive hypothesis to get $e\ e' \in \text{Adeq}_B$, as we know $[\![e\ e']\!] = \bot_B$ and $\vdash e\ e' \in B$ (from the typing rule for function application with $e' \in \mathit{Good}_A$), so to do this, we just need to prove $e\ e' \mapsto^\infty$, which we prove by contradiction:

Assume $e\ e' \mapsto \underline{n}$, for some $n \in \mathbb{N}$. Then by correctness, we have $[\![e\ e']\!] = n$. But $[\![e\ e']\!] = \bot_B$, so we have a contradiction. So $e\ e' \mapsto^{\infty}$.Now we apply the inductive hypothesis and get $e\ e' \in \text{Adeq}_B$.

So for any $e' \in \text{Adeq}_A$, we have $e\ e' \in \text{Adeq}_B$.

Now we have proved the lemma for any type. □

We also prove the following lemma on substitutions:

7.3.2 Substitution Lemma

Lemma 5. *If* $\gamma \in \text{Adeq}_{Ctx}(\Gamma)$ *and* $\Gamma \vdash e : A$ *then* $\vdash [\gamma](e) : A$

Proof. By induction on Γ. The base case is when Γ is the empty context. We have $\gamma = <>$, so we need to prove $\vdash <> e : A$. The empty substitution does nothing, so this is the same as $\vdash e : A$, which we already have as an assumption.

The inductive case is where we have $\gamma \in \text{Adeq}_{Ctx}(\Gamma, x : A)$. Let $\gamma = (\gamma', e'/x)$ where $\gamma' \in \text{Adeq}_{Ctx}(\Gamma)$ and $e' \in \text{Adeq}_A$.

From $e' \in \text{Adeq}_A$, we have $\vdash e' : A$. We can use weakening on each variable in the context individually to get $\Gamma \vdash e' : A$. We use type substitution with this and the assumption $\Gamma, x : A \vdash e : A$ to get $\Gamma \vdash [e'/x]e : A$.

Now we apply the inductive hypothesis of the theorem with $\gamma' \in \text{Adeq}_{Ctx}(\Gamma)$ to get $\vdash [\gamma'][e'/x']e : A = \vdash [\gamma]e : A$

Now we have proved the lemma for any type. □

7.4 Main Lemma

Now we can prove the Main Lemma:

Proof. By induction on $\Gamma \vdash e : A$

Variables Given $\Gamma \vdash x : A$, from the typing rule we know that $\Gamma(x) = A$. Therefore we always have $x \in dom[\gamma]$.

As we have also assumed $\gamma \in \text{Adeq}_{Ctx}(\Gamma)$, we must have e'/x in γ, where $e' \in \text{Adeq}_A$.

By the definition of substitution on variables, if the variable we are substituting is not x, then we just return x, so $[\delta, e'/x, \delta']x = [\delta, e'/x]x = [\delta]e'$.

As $e' \in \text{Adeq}_A$, we know e' is a closed term, so there are no free variables to substitute. Therefore $[\delta]e' = e'$, and $[\gamma]x \in \text{Adeq}_A$.

Zero $[\gamma](zero) = zero$, so we need to prove that $zero \in \text{Adeq}_{\text{Nat}}$. Therefore we first prove $\vdash zero : \text{Nat}$. As $zero$ is a constant, then we have it in any typing context, including the empty context. We must also prove $[\![zero]\!] = n \Rightarrow zero \mapsto^* \underline{n}$. 0 is the only possible value of n, as defined by the denotational semantics, so we need $zero \mapsto^* \underline{0}$. $zero$ is our representation of the numeral $\underline{0}$, so it maps to this in 0 steps. Therefore $zero \in \text{Adeq}_{\text{Nat}}$.

Successor $[\gamma]s(e) = s([\gamma]e)$, so we must prove $s([\gamma]e) \in \text{Adeq}_{\text{Nat}}$. As we have a derivation of $\Gamma \vdash s(e) : \text{Nat}$, from the typing rule we also have $\Gamma \vdash e : \text{Nat}$. Therefore we can apply the inductive hypothesis to this and $\gamma \in \text{Adeq}_{Ctx}(\Gamma)$ to get $[\gamma]e \in Good_{\text{Nat}}$. Now there are two things we must show:

1. $\vdash s([\gamma]e) : \text{Nat}$
   We have $\vdash [\gamma]e : \text{Nat}$ from $[\gamma]e \in \text{Adeq}_{\text{Nat}}$, so we use the typing rule on this to get $\vdash s([\gamma]e) : \text{Nat}$
2. $[\![s([\gamma]e)]\!] = n \Rightarrow s([\gamma]e) \mapsto^* \underline{n}$
   Assume $[\![s([\gamma]e)]\!] = n$. By the definition of $[\![s([\gamma]e)]\!]$, we must have $[\![[\gamma]e]\!] = n - 1$, because this is the only case that does not give $\bot$ as the final output. From $[\gamma]e \in \text{Adeq}_{\text{Nat}}$ and this we have $[\gamma]e \mapsto^* \underline{n-1}$. Using the congruence rule for successor, with this as our assumption, we have $s([\gamma]e) \mapsto^* s(\underline{n-1})$, which is the same as the numeral $\underline{n}$. Therefore $s([\gamma]e) \mapsto^* \underline{n}$

Therefore $s([\gamma]e) \in \text{Adeq}_{\text{Nat}}$, so by the definition of $[\gamma]$ we have $[\gamma]s(e) \in \text{Adeq}_{\text{Nat}}$.

Case $[\gamma](\text{case}(e, z \mapsto e_0, s(x) \mapsto e_S)) = \text{case}([\gamma](e), z \mapsto [\gamma](e_0), s(x) \mapsto [\gamma](e_S)$, so we need to prove that $\text{case}([\gamma](e), z \mapsto [\gamma](e_0), s(x) \mapsto [\gamma](e_S)) \in \text{Adeq}_A$, for some type A. As we have a derivation of $\Gamma \vdash \text{case}(e, z \mapsto e_0, s(x) \mapsto e_S : A)$, from the typing rule we also have derivations for $\Gamma \vdash e : \text{Nat}$, $\Gamma \vdash e_0 : A$ and $\Gamma, x : \text{Nat} \vdash e_S : A$.

Therefore we can apply the inductive hypothesis to this and $\gamma \in \text{Adeq}_{Ctx}(\Gamma)$ to get $[\gamma]e \in \text{Adeq}_{\text{Nat}}$ and $[\gamma]e_0 \in \text{Adeq}_A$.

Now we have three cases for the proof, depending on the value of $[\gamma]e$:

1. When $[\gamma]e = zero$, the evaluation rule gives us $[\gamma]e_0$, so we must prove $[\gamma]e_0 \in \text{Adeq}_A$. We have the derivation tree for $\Gamma \vdash e_0 : A$, so we apply the inductive hypothesis with this and $\gamma \in \text{Adeq}_{Ctx}(\Gamma)$ to get $[\gamma]e_0 \in \text{Adeq}_A$
2. When $[\gamma]e = s(v)$, the evaluation rule gives us $[v/x]([\gamma]e_S)$, so we must prove $[v/x]([\gamma]e_S) \in \text{Adeq}_A$. We have the derivation tree for $\Gamma, x : \text{Nat} \vdash e_S : A$, so we apply the inductive

hypothesis with this and $\gamma, v/x \in \mathit{Good}_{Ctx}(\Gamma, \mathrm{Nat})$ to get $[\gamma']e_S \in \mathrm{Adeq}_A$

3. When e does not evaluate to a value, we need to show that for any expression e, its case statement is still in Adeq_A, so we will ultimately use **Lemma 4**.

   By the definition of the denotational semantics, when we have $[\![[\gamma]e]\!] = \bot$, then $[\![\mathit{case}([\gamma]e, z \mapsto [\gamma]e_0, s(v) \mapsto [\gamma]e_S]\!] = \bot$.

   We know that our case expression is a well typed closed term by **Lemma 5**, as we use our original assumption for $\Gamma \vdash \mathrm{case}(e, z \mapsto e_0, s(v) \mapsto e_S)$ to get $\vdash [\gamma]\,\mathrm{case}(e, z \mapsto e_0, s(v) \mapsto e_S)$.

   Now we know it is a closed term, we can use the **Lemma 4**, to get $\mathrm{case}([\gamma]e, z \mapsto [\gamma]e_0, s(v) \mapsto [\gamma]e_S) \in \mathrm{Adeq}_A$

Application $[\gamma](e_0\ e_1) = ([\gamma]e_0)([\gamma]e_1)$, so we need to prove that $([\gamma]e_0)([\gamma]e_1) \in \mathrm{Adeq}_B$. As we have a derivation of $\Gamma \vdash e_0\ e_1 : B$, from the typing rule, we have derivations for $\Gamma \vdash e_0 : A \to B$ and $\Gamma \vdash e_1 : A$. Therefore we can apply the inductive hypothesis to these derivations and $\gamma \in \mathrm{Adeq}_{Ctx}(\Gamma)$ to get $[\gamma]e_0 \in \mathrm{Adeq}_{A\to B}$ and $[\gamma]e_1 \in \mathrm{Adeq}_A$.

From $[\gamma]e_0 \in \mathrm{Adeq}_{A\to B}$, we know that $\forall e' \in \mathrm{Adeq}_A .([\gamma]e_0\ e') \in \mathrm{Adeq}_B$. As $[\gamma]e_1 \in \mathrm{Adeq}_A$, then $([\gamma]e_0)([\gamma]e_1) \in \mathrm{Adeq}_B$.

λ-Abstraction $[\gamma](\lambda x : A.\ e) = \lambda x : A.\ [\gamma]e$ so we must prove $\lambda x : A.\ [\gamma]e \in \mathrm{Adeq}_{A\to B}$.

There are two things we need to show:

1. $\vdash \lambda x : A.\ [\gamma]e : A \to B$
   We can prove this using **Lemma 5**, as we have $\Gamma \vdash \lambda x : A.\ e : A \to B$. By using the lemma, we have $\vdash [\gamma]\lambda x : A.e : A = \vdash \lambda x : A.\ [\gamma]e : A \to B$

2. $\forall e' \in \mathrm{Adeq}_A\ .\ (\lambda x : A.\ [\gamma]e)\ e' \in \mathrm{Adeq}_B$
   Let $e' : A$ be an expression such that $e' \in \mathrm{Adeq}_A$.

   As we have a derivation of $\Gamma \vdash \lambda x : A.\ e : A \to B$, from the typing rule, we have $\Gamma, x : A \vdash e : B$. Let $\gamma' = (\gamma, e'/x)$, now we have $\gamma' \in \mathrm{Adeq}_{Ctx}(\Gamma, A)$, so we use the induction hypothesis to get $[\gamma']e \in \mathrm{Adeq}_B$.

   Using the evaluation rule for λ abstraction we have $(\lambda x : A.\ [\gamma]e)\ e' \mapsto [e'/x][\gamma]e$, which can be simplified to $[\gamma, e'/x]e = [\gamma']e$. We have $\vdash e' : A$ from $e' \in \mathrm{Adeq}_A$ and $\vdash \lambda x : A.\ [\gamma]e : A \to B$ from the previous case, so we use the typing rule for function application to get $\vdash (\lambda x : A.\ [\gamma]e)\ e' : B$.

   As $[\gamma']e \in \mathrm{Adeq}_B$, we can use the Expansion Lemma to get $(\lambda x : A.\ [\gamma]e)\ e' \in \mathrm{Adeq}_B$. Therefore we know $\forall e' \in \mathrm{Adeq}_A\ .\ (\lambda x : A.\ [\gamma]e)\ e' \in \mathrm{Adeq}_B$.

Fixpoint $[\gamma](\text{fix } x : A.e) = \text{fix } x : A.[\gamma]e$ so we must prove $\text{fix } x : A.[\gamma]e \in \text{Adeq}_A$. As we have a derivation of $\Gamma \vdash \text{fix } x : A.\ e : A$, from the typing rule, we have $\Gamma, x : A \vdash e : A$. Let $\gamma' = (\gamma, e'/x)$ for some expression $e' \in \text{Adeq}_A$ (so we have $\gamma' \in \text{Adeq}_{Ctx}(\Gamma, A)$). Then using the induction hypothesis we have $[\gamma']e \in \text{Adeq}_A$. Let $e' = \text{fix } x : A\ e$. Then $[\gamma']e = [\gamma, \text{fix } x : A\ e/x]e = [\gamma][\text{fix } x : A\ e/x]e$. Then as $[\gamma](\text{fix } x : A.e) \mapsto [\gamma]([\text{fix } x : A\ e/x]e)$ and $[\gamma]([\text{fix } x : A\ e/x]e) \in \mathit{Good}_A$, we use the Expansion Lemma with $\vdash \text{fix } x : A\ e : A$ to get $[\gamma](\text{fix } x : A.e) \in \text{Adeq}_A$.

□

We can now prove the second direction of Adequacy as a corollary of the Main Lemma:

Corollary 2. *If* $\vdash e : \text{Nat}$ *and* $[\![e]\!] = n$ *then* $[\![e]\!] = n \Rightarrow e \mapsto^* \underline{n}$

Proof. Using the Main Lemma, as we have $\vdash e : \text{Nat}$ and the empty substitution will be in $\text{Adeq}_{Ctx}(\cdot)$, we will have $[<>]e \in \text{Adeq}_{\text{Nat}}$. $[<>]e = e$, so we have $e \in \text{Adeq}_{\text{Nat}}$.

Expanding this definition gives us $[\![e]\!] = n \Rightarrow e \mapsto^* \underline{n}$, which is what we wanted to prove. □

Chapter 8

Non-Definability of Parallel-Or

Chapter 9

Evaluation

Chapter 10

Conclusion and Future Work

Bibliography

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

G. Hutton. Introduction to domain theory (course notes). 2014.

A. Murawski. Logical relations. Course Notes from MGS 2012, 2012.