Investigating Properties of PCF in Agda

Brendan Hart
Supervised by Martín Escardó
Co-supervised by Tom de Jong

MSci Computer Science 2020,
School of Computer Science, University of Birmingham, UK

Abstract

Using Agda as a proof assistant, we explore properties of PCF. We work in a constructive setting using univalent type theory. We first define PCF, and its operational semantics. We next build upon the current formalisation of domain theory in Agda by Tom de Jong, formalising the necessary constructions which allows us to construct the Scott model of PCF. Finally, we show that this model is computationally adequate with respect to the operational semantics.

Contents

1 Introduction

Our goal is to define PCF, its operational semantics, and the denotational semantics via the Scott model. We proceed to show that the denotational semantics are computationally adequate with respect to the operational semantics. These are not new proofs, and we will be staying close to Streicher's *Domain-Theoretic Foundations of Functional Programming* [Str06]. We deviate in our choice of variable representation, and that we work in Voevodsky's constructive univalent type theory [Uni13], however most proofs follow closely.

We shall be using a dependently typed programming language called Agda. Due to its strong typing and support for dependent types, we can use Agda as a proof assistant. This is since from the Curry-Howard correspondence, we can express logical specifications using dependent types, and the programs we define are proofs [How69]. During our developments, we experience the differences between proofs on paper, and developing our proofs in Agda. We see times where we are required to formalise properties which may be left out of paper proofs - sometimes things which may be considered trivial. We also appreciate the support Agda provides, such as reminding us of our assumptions at a particular stage in a proof, or allowing us to easily case split and ensure we account for every possible case in our proof.

We build upon Tom de Jong's current formalisation of domain theory in Agda, which he used to define the Scott model for a combinatorial version of PCF [Jon19]. We differ from Tom de Jong's implementation of PCF as we provide support for lambda abstraction and variables. For use when we construct our model, we formalise the product between DCPOs, currying of continuous functions, and uncurrying of continuous functions. We also make use of Martín Escardó's developments in Agda of univalent type theory [Esc+] and the lifting monad [EK17].

From this, we then are able to construct the Scott model of PCF [Sco93]. We show that the operational semantics are correct with respect to the Scott model, i.e. terms P and V are denotationally equal whenever $P \Downarrow V$. Finally, we then show that the Scott model is computationally adequate with respect to the operational semantics, i.e. $P \Downarrow \underline{n}$ whenever the terms P and n are denotationally equal.

Why use a proof assistant?

One may wonder why we are bothering to formalise these properties in Agda when there are papers describing proofs of these properties already. The first motivation behind this surrounds how we know that proofs are correct. When we write proofs on paper, we rely on humans to check the proofs, and naturally this introduces human error. There are many examples through history of proofs which claim to show a property, only for it to be refuted at a later date. Even more related to our case, when Sébastien Gouëzel was formalising a proof by Vladimir Shchur in Isabelle, he managed to find a gap in the proof. Later, they published a joint paper to correct this gap [GS19]. It could be argued that the same discovery could be made without the proof assistant, and that

argument would be correct. However, if we formalise our proof in a proof assistant, then these errors would not occur (assuming the compiler is correct). Also, it may take years for a human to find an error in a proof, whereas a computer can decide much quicker.

Using a proof assistant requires that we formalise and prove small details which we may not normally formalise on paper. In mathematics, it's normal to make large steps in proofs, and sometimes leave out proofs that may be seen as trivial - "left as an exercise for the reader" is a common phrase. Proof assistants do not allow this. We must explicitly prove every property we are interested in, and each detail of a proof must be present so it can be machine-checked. During our developments, we will see times where we must come up with a proof of some properties that are mentioned to be true, but no actual proof is given in the papers which we are staying close to.

There are also some convenience features which some proof assistants provide that we lack when proving properties on paper. During our developments, we will see structural induction which provides us with many different cases. On paper, we would have to manually ensure we have considered every possible case, and sometimes this may be difficult as there can be many. If we miss any cases, our proof is incomplete. When using Agda as our proof assistant, we can tell Agda to generate the cases for us, called case splitting. Agda will also warn us if we have missed any cases in our proof. Another feature which makes proof assistants extremely useful is that they can provide us with our current assumptions at a particular stage in a proof. This is especially useful if we are deep in a lengthy proof. On paper, this could become difficult to keep track of and a cognitive burden, as opposed to allowing the proof assistant to keep track of this.

2 Background and Considerations

We first explain some necessary background and considerations which will be required when understanding definitions in this text, particularly some concepts in type theory which we shall be using.

2.1 Agda flags

During our development, we use the following flags, which we specify at the top of each file:

- `--without-K` - This disables Streicher's K axiom, allowing us to work with univalent mathematics.
- `--exact-split` - Makes Agda only accept definitions that behave like definitional equalities.
- `--safe` - Disables postulates.

This will be the only time we mention these flags throughout this text, but it helps to set some context for how we shall be working in Agda.

2.2 Universes

To avoid Russel's paradox, Agda has universes [Agd]. Universes are types whose elements are types themselves. We have an ascending, infinite number of universes. One might consider data types such as Int and Bool. These types may live in the universe $\mathcal{U}_0$, which we use to represent the lowest level universe. If we were to have $\mathcal{U}_0$ belonging in the universe $\mathcal{U}_0$, we would encounter Russel's paradox. As such, we define $\mathcal{U}_0$ as existing in the universe $\mathcal{U}_1$, and then $\mathcal{U}_1$ exists in the universe $\mathcal{U}_2$, etc.

In Agda, by default, the keyword Set is used to represent the lowest level universe, and the larger universes then as Set_1, Set_2, etc. The subscript n is the level of the universe Set_n

We rename the default Agda implementation so we can match our original terminology, staying closer to the notation we tend to use in homotopy type theory. We perform the following renamings:

- Level to Universe.
- lzero (the lowest level in Agda) to $\mathcal{U}_0$.
- lsuc to $^+$, so we can refer to the universe above $\mathcal{U}_0$ as $\mathcal{U}_0{}^+$.
- Setω to $\mathcal{U}\omega$, which is a universe where, for all n, $\mathcal{U}\omega$ is strictly above $\mathcal{U}_n$.
- Given a universe level $\mathcal{U}$, we denote the universe type as $\mathcal{U}^\cdot$ (note the combining dot above).

Throughout, we will use the letters $\mathcal{U}$, $\mathcal{V}$, $\mathcal{W}$, $\mathcal{T}$ to refer to arbitrary universes levels.

2.3 The identity type

We use Martin-Löf's identity type as our notion of equality. We say that the type Id $X\ x\ y$ represents the equality of terms x and y under the type X. We have the sole constructor of this type, refl, which produces an element of the type Id $X\ x\ x$, for any $x : X$.

In our developments, we use an alternative notation. We say $x \equiv y$ to represent the equality of terms x and y under a type X. We note that we do not need to provide the type X, as Agda can infer this. Our constructor, refl, also does not explicitly require us to provide the x in our Agda developments, as this can again be inferred from the type we are trying to construct.

2.4 Propositions

Propositions (sometimes called subsingletons, or truth values) are, in univalent mathematics, defined as a type with at most one element. Another way of saying this is that if we have two elements from a proposition, then by definition they must be equal. Formally, we define this as:

Definition 2.1. A type X is a proposition if for any two elements $x, y : X$, x is equal to y.

There is a way we can "squash" a type down to a proposition. We call this propositional truncation. We use $\|X\|$ to take the propositional truncation of X. For example, let's consider the propositional truncation of the dependent pair for a type family $P : X \to \mathcal{U}$, which we write as $\|\Sigma_{x:X} P(x)\|$. We tend to think of the dependent pair as a proof that the first projection x has a property $P(x)$, and

the proof that x has this property is the second projection. If we take the propositional truncation of this dependent pair, we then think of this truncated type as the fact that there exists some x for which $P(x)$ holds, but we do not remember which particular x.

For a type X, we define the type $\|X\|$ as the propositional truncation of X using the following constructors:

- For any $x : X$, $|x| : \|X\|$.
- For any $x, y : \|X\|$, $x \equiv y$.

The recursion principle of propositional truncation allows us to do case analysis when we are trying to prove another proposition from a proof of a truncated type. In our developments, we use ‖‖-rec for this. Given a type X, we can conclude a proposition P holds if we can provide a proof that P is a proposition, a proof $X \to P$, and a proof of $\| X \|$. A particular case of ‖‖-rec comes when we are trying to show, for types X and Y, $\| Y \|$ from $\| X \|$. Since the propositional truncation is by definition a proposition, we can use ‖‖-functor to show this, which only requires a proof $X \to Y$, and a proof of $\| X \|$.

We note that in our developments we work with an axiomatic approach to propositional truncation, as Martín Escardó does [Esc19], meaning we assume the constructions defined above exist.

2.5 Sets

In homotopy type theory, not all types are sets. Sets are types with a special property.

Definition 2.2. A type X is a set if for any two elements x and y of type X, the identity type $x \equiv y$ is a proposition.

We use this definition later when defining DCPOs, as we require the underlying type to be a set.

3 PCF

In 1977, Gordon Plotkin considered Dana Scott's logic of computable functions as a programming language, called PCF (programming language for computable functions) [Plo77]. PCF can be seen as an extended version of the typed lambda calculus, since it contains extra constructs such as the fixed-point operator to allow for general recursion. One may question why we would consider such a toy language when we have practical languages such as Haskell or ML which we could reason about. However, PCF can be seen as a simplified version of these typed functional languages. Due to the simplistic nature, it is easier to reason about PCF compared to a large language with extra constructs or features such as concurrency.

We begin by formalising PCF with the base type of natural numbers, and a constructor to create function types. Since our terms can contain variables, we determine how we are going to represent them. We proceed to assign types to our terms intrinsically via the typing judgement.

3.1 Types and terms

Definition 3.1. We define the types of PCF inductively as follows:

- The base type ι represents the natural numbers.
- If σ and τ are types in PCF, σ ⇒ τ is the function type from σ to τ.

We give meaning to these types later, when we define the semantics of PCF.

In Agda, we can define this as follows:

```
data type : 𝒰₀̇ where
  ι : type
  _⇒_ : type → type → type
```

The data keyword allows us to inductively define a datatype. We give the name type to this particular datatype, as we are defining our PCF types. The definition states we have two constructors, ι, which takes no arguments, and ⇒, which constructs a new type from two types. The underscores in the definition of ⇒ define where the arguments go when using the constructor. Therefore, _⇒_ defines ⇒ as an infix operator, e.g. σ ⇒ τ. If we leave the underscores in the operator name when we use it, we can use the constructor in the same manner as standard application, i.e. _⇒_ σ τ. The definition also states that our datatype type lives in the universe $\mathcal{U}_0$, which is our lowest level universe.

We then should consider the terms which we will be assigning these types to. The following grammar captures the terms we will be considering:

L, M, N ::= v x | ƛ M | M · N | Zero | Succ M | Pred M | IfZero L M N | Fix M

From the above, we have the following terms that operate on the natural numbers:

- Zero - Represents the natural number zero.
- Pred M - The predecessor of M (e.g. the predecessor of 4 would be 3).
- Succ M - The successor of M (e.g. the successor of 2 would be 3).
- IfZero L M N - Allows for conditional statements, and returns M or N depending on the value of L.

We then have the constructs for lambda abstraction, function application, and variables:

- ƛ M - Represents a lambda abstraction, with body M.
- M · N - Represents applying a term M to a term N.
- v x - Represents a variable, with identifier x.

Finally, the construct for general recursion:

- Fix M - Represents the fixed-point combinator, which returns the fixed point of M, allowing for general recursion.

3.2 Representing variables

Since a PCF term may contain free variables, we need some way of representing the variables that may occur inside a term. To do this, we use a context. A context is a list of variables and their associated types. They are of the form:

$$\Gamma = x_0 : \sigma_0, ..., x_n : \sigma_n$$

where σ_i is the type of the variable with identifier x_i. We tend to use the symbol Γ, and sometimes Δ, to represent an arbitrary context.

One uncertainty about the definition of the terms above is how should we identify variables, i.e. what should x_i be. When we consider programming languages, we tend to use strings to refer to variables, which are called named variables. Whilst valid, let's consider the following program:

```
example : ℕ -> ℕ -> ℕ
example = λ x → λ x → x
```

Of course, the variable referred to in the body should be the one of the innermost lambda. The inner variable is said to mask the outer variable of the same name. Whilst obvious, this still requires insight to implement and overcome the issue of variable shadowing. When implementing substitution, if we wanted to replace the variable x in the term example above, it requires we solve this complication of which x we should, or should not, replace. One way of removing name conflicts is to use α-conversion, which is the renaming of bound variables. This has its own implementation complications. For example, if we rename a bound variable, we need to ensure the new identifier does not occur in the scope of the bound variable already, otherwise the meaning of the original term will not be preserved.

A common alternative is to use de Bruijn indices [Bru72]. Rather than using strings to identify variables, we use an index which identifies which variable in the context we are referring to. Since a variable in the context will have a unique index, the problem of variable shadowing disappears as terms written using these indices are invariant with respect to α-conversion.

Therefore, since the index of the variable is itself the identifier, we can instead represent variables as lists of just types:

$$\Gamma = \sigma_0, ..., \sigma_n$$

Now we decide how to formalise this in Agda. One may first consider representing a context as a list of types. However, the issue with this is when we want to extract the i^{th} variable via the i^{th} projection map, this function would not be defined for lists of length less than i. Agda is total, meaning we cannot define functions which are partial or throw exceptions. To ensure that we only allow projections which are valid, we embed the length of the list within the type. This is an example of the advantage of dependent types.

From this, we can consider a context to correspond to a vector of length n, where the type of elements of the vector is fixed to be PCF types. We define our context to grow to the right, as is convention.

```
data Vec (X : 𝒰₀˙) : ℕ → 𝒰₀˙ where
  ⟨⟩ : Vec X zero
  _'_ : {n : ℕ} → Vec X n → X → Vec X (succ n)

Context : ℕ → 𝒰₀˙
Context = Vec type
```

Here, we see our first use of implicit arguments. Sometimes, an argument can be worked out by Agda based on the context, or other arguments to the function. In such cases, we may choose to make the argument implicit, which means when using the definition, we do not need to provide the implicit argument as Agda will infer it. In the definition of the symbol ', we can see that n is an implicit argument. We define implicit arguments by surrounding them with curly brackets. The reason Agda can work out the argument which should be given for n is because it is uniquely determined by the constructor's second argument of type Vec X n.

We then formalise the lookup judgement, for which we follow the definition in [KSW20]. We define a datatype $\Gamma \ni A$ for variables of type A which exist in the context Γ. The inhabitants of these types can be seen as corresponding to the index of the variable in the context. We note that the zeroth index is the rightmost element of our context, and incrementing as we proceed left.

```
data _∋_ : {n : ℕ} → Context n → type → 𝒰₀˙ where
  Z : ∀ {n : ℕ} {Γ : Context n} {σ : type} → (Γ ' σ) ∋ σ
  S : ∀ {n : ℕ} {Γ : Context n} {σ τ : type}
      → Γ ∋ σ
      → (Γ ' τ) ∋ σ
```

Below are two examples of proofs of $\Gamma \ni \iota$, where Γ is our Agda representation of ι,(ι ⇒ ι),ι,(ι ⇒ ι):

```
ex₁ : (((((⟨⟩ ' ι) ' (ι ⇒ ι)) ' ι) ' (ι ⇒ ι)) ∋ ι
ex₁ = S Z

ex₂ : (((((⟨⟩ ' ι) ' (ι ⇒ ι)) ' ι) ' (ι ⇒ ι)) ∋ ι
ex₂ = S (S (S Z))
```

We can see that S Z is similar to referring to the first index of the context, and S (S (S Z)) is similar to referring to the third index.

3.3 Assigning types to terms

We next look at our typing judgement, which are a set of rules applied inductively to generate well-typed terms. For our definition, we follow a Church-style system, where types are an intrinsic part of the language semantics [Pie02].

Definition 3.2. We inductively define the type PCF $\Gamma\ \sigma$ to refer to the well-typed PCF terms which have type σ under the context Γ by the rules below:

$$\frac{x : \Gamma \ni \sigma}{v\ x : \mathrm{PCF}\ \Gamma\ \sigma} \qquad \frac{M : \mathrm{PCF}\ (\Gamma\ ,\ \sigma)\ \tau}{\lambda\ M : \mathrm{PCF}\ \Gamma\ (\sigma \Rightarrow \tau)} \qquad \frac{M : \mathrm{PCF}\ \Gamma\ (\sigma \Rightarrow \tau) \qquad N : \mathrm{PCF}\ \Gamma\ \sigma}{M \cdot N : \mathrm{PCF}\ \Gamma\ \tau}$$

$$\frac{}{\mathrm{Zero} : \mathrm{PCF}\ \Gamma\ \iota} \qquad \frac{M : \mathrm{PCF}\ \Gamma\ \iota}{\mathrm{Succ}\ M : \mathrm{PCF}\ \Gamma\ \iota} \qquad \frac{M : \mathrm{PCF}\ \Gamma\ \iota}{\mathrm{Pred}\ M : \mathrm{PCF}\ \Gamma\ \iota}$$

$$\frac{M : \mathrm{PCF}\ \Gamma\ (\sigma \Rightarrow \sigma)}{\mathrm{Fix}\ M : \mathrm{PCF}\ \Gamma\ \sigma} \qquad \frac{M_i : \mathrm{PCF}\ \Gamma\ \iota}{\mathrm{IfZero}\ M_1\ M_2\ M_2 : \mathrm{PCF}\ \Gamma\ \iota}\ \text{i=1,2,3}$$

In Agda, the definition looks very similar.

```
data PCF : {n : ℕ} (Γ : Context n) (σ : type) → 𝒰₀̇ where
  Zero : {n : ℕ} {Γ : Context n} → PCF Γ ι
  Succ : {n : ℕ} {Γ : Context n} → PCF Γ ι → PCF Γ ι
  Pred : {n : ℕ} {Γ : Context n} → PCF Γ ι → PCF Γ ι
  IfZero : {n : ℕ} {Γ : Context n} → PCF Γ ι → PCF Γ ι → PCF Γ ι → PCF Γ ι
  ƛ    : {n : ℕ} {Γ : Context n} {σ τ : type} → PCF (Γ , σ) τ → PCF Γ (σ ⇒ τ)
  _·_  : {n : ℕ} {Γ : Context n} {σ τ : type} → PCF Γ (σ ⇒ τ) → PCF Γ σ → PCF Γ τ
  v    : {n : ℕ} {Γ : Context n} {A : type} → Γ ∋ A → PCF Γ A
  Fix  : {n : ℕ} {Γ : Context n} {σ : type} → PCF Γ (σ ⇒ σ) → PCF Γ σ
```

4 Operational Semantics

To execute code on a computer, there must be rules defined which a computer can then apply to arrive at a result. One might consider the operational semantics of Haskell to be the rules which the Haskell interpreter applies when executing a program. So, we can take the operational semantics of a language to be a model which represents the execution of valid programs in that language.

There are different approaches to defining this model, such as small-step semantics, or big-step semantics. The big-step semantics and small-step semantics are shown to coincide by Streicher [Str06, Section 2]. Streicher goes on to consider the big-step semantics throughout the paper since they are more abstract, as they lose the intermediate computation steps unlike small-step semantics. We shall also consider the big-step semantics for the same reasons, and to keep our proofs similar.

4.1 Substitution

Before we define the big-step semantics, we need the concept of substitution. When we consider the term $\lambda\ M \cdot N$, it should evaluate to the same term as M with N substituted for the first variable. Due to our choice of variable implementation, the implementation of substitution is fairly simple.

We implement substitution in Agda as in [KSW20]. We define the function subst, which defines how

to apply a mapping, from variables in a context to terms in a new context, to a term. This is called simultaneous substitution. When we come across a lambda term, we essentially are introducing a new variable into the context. As a result, we want to extend our mapping by shifting the existing indices up by one, and leave the new zeroth index untouched when we apply the substitution to the subterm. The exts helper function performs this operation, although is omitted for brevity.

```
subst : ∀ {m n} {Γ : Context m} {Δ : Context n} → (∀ {A} → Γ ∋ A → PCF Δ A)
                                                → (∀ {A} → PCF Γ A → PCF Δ A)
subst f Zero = Zero
subst f (Succ t) = Succ (subst f t)
subst f (Pred t) = Pred (subst f t)
subst f (IfZero t t₁ t₂) = IfZero (subst f t) (subst f t₁) (subst f t₂)
subst f (ƛ t) = ƛ (subst (exts f) t)
subst f (t · t₁) = (subst f t) · (subst f t₁)
subst f (v x) = f x
subst f (Fix t) = Fix (subst f t)
```

We then define a function which applies the substitution of replace-first, which replaces the first variable with a given term, and maps the variable S x to x, providing us with a function that replaces only the first variable in a term.

```
_[_] : {n : ℕ} {Γ : Context n} {σ A : type}
       → PCF (Γ , A) σ → PCF Γ A → PCF Γ σ
_[_] {n} {Γ} {σ} {A} M N = subst (replace-first A Γ N) M
```

4.2 Big-step semantics of PCF

From our understanding that the operational semantics relates to the evaluation of terms, we can define the big-step semantics for PCF, as well as consider some properties which follow from the definition.

For each natural number n, we write the canonical numeral as $\underline{n}$. When we are writing Agda code, we construct these canonical expressions with the function ℕ-to-ι. We can imagine this as a function which takes a natural number, and produces the simplest representation possible in PCF. E.g. ℕ-to-ι *2* reduces to Succ (Succ Zero).

Definition 4.1. We define the big-step relation between terms M and N, $M \Downarrow N$, using the inductive rules below:

$$\frac{}{\text{Zero} \Downarrow \text{Zero}} \qquad \frac{M \Downarrow \underline{n}}{\text{Succ}\, M \Downarrow \underline{n+1}} \qquad \frac{M \Downarrow \underline{0}}{\text{Pred}\, M \Downarrow \underline{0}} \qquad \frac{M \Downarrow \underline{n+1}}{\text{Pred}\, M \Downarrow \underline{n}}$$

$$\frac{}{v\ i \Downarrow v\ i} \qquad \frac{}{ƛ\ t \Downarrow ƛ\ t} \qquad \frac{M \Downarrow ƛ E \qquad E[N] \Downarrow V}{M \cdot N \Downarrow V} \qquad \frac{M \cdot (\text{Fix}\, M) \Downarrow V}{\text{Fix}\, M \Downarrow V}$$

$$\frac{M \Downarrow \underline{0} \qquad M_2 \Downarrow V}{\text{IfZero}\, M\ M_1\ M_2 \Downarrow V} \qquad \frac{M \Downarrow \underline{n+1} \qquad M_1 \Downarrow V}{\text{IfZero}\, M\ M_1\ M_2 \Downarrow V}$$

From the above definition, it might be clear to see why it's called big-step. Rather than considering the individual steps of a computation like small-step would, the rules for big-step make assumptions about how the constituents of a term evaluate, in order to determine the value that the term itself evaluates to.

In Agda, we first define the relation $\Downarrow'$ as follows. Since the definition spans many lines and is very similar to the rules above, we only show select cases for comparison.

```
data _⇓'_ : ∀ {n : ℕ} {Γ : Context n} {σ : type} → PCF Γ σ → PCF Γ σ → 𝒰₀̇ where
  -- Most rules omitted for brevity
  IfZero-zero : {n : ℕ} {Γ : Context n}
              {M : PCF Γ ι} {M₁ : PCF Γ ι} {M₂ : PCF Γ ι} {V : PCF Γ ι}
              → M ⇓' ℕ-to-ι {n} {Γ} zero
              → M₁ ⇓' V
              → IfZero M M₁ M₂ ⇓' V
  Fix-step : {n : ℕ} {Γ : Context n} {σ : type} {M : PCF Γ (σ ⇒ σ)} {V : PCF Γ σ}
              → (M · (Fix M)) ⇓' V
              → Fix M ⇓' V
  ·-step   : {n : ℕ} {Γ : Context n} {σ τ : type}
              {M : PCF Γ (σ ⇒ τ)} {E : PCF (Γ , σ) τ} {N : PCF Γ σ} {V : PCF Γ τ}
              → M ⇓' ƛ E
              → (E [ N ]) ⇓' V
              → (M · N) ⇓' V
```

One consideration is that we need the big-step relation to be a proposition. Whilst this should be true, it can be difficult to prove. As a result, we use propositional truncation which squashes a type down to a mere proposition, allowing us to define $\Downarrow$ as:

```
_⇓_ : {n : ℕ} {Γ : Context n} {σ : type} → PCF Γ σ → PCF Γ σ → 𝒰₀̇
M ⇓ N = ∥ M ⇓' N ∥
```

From this definition, we can then show that an application of big-step semantics always reduces a term to its normal form, which is called a value. For the following proofs, we omit the Agda code as the proofs are trivial.

Definition 4.2. A PCF term is a value if it is either a lambda abstraction, numeral, or a variable.

Lemma 4.3. *For every PCF term M, if $M \Downarrow' V$ then V is a value.*

Proof. By induction on $M \Downarrow' V$. □

It is also interesting to note that there is only ever one unique reduction that can be applied to any given term. This means that the big-step relation is deterministic.

Proposition 4.4 (Big-step is deterministic)**.** *If $M \Downarrow' N$ and $M \Downarrow' L$, then N and L are equal.*

Proof. By induction on $M \Downarrow' N$. □

5 Domain Theory

We have defined the semantics of PCF from an operational view. Next, we intend to look at the denotational view. To do this, we need to formalise the necessary domain theory which will allow us to construct the Scott model of PCF. We begin by providing some intuition for the use of domains, which are partially ordered sets with special structure.

5.1 Intuition

When trying to define a mathematical model of a programming language, the first idea one might come up with is to interpret types as sets. For example, we may first try to interpret the PCF type ι as the set of natural numbers. The issue which arises is what if we were to produce a program which did not terminate? We would not want to associate any natural number with this non-terminating program. We take Scott's approach of introducing the least element $\bot$ into each domain to represent programs which do not terminate.

Now we have an idea of what the elements of our representation for base types may be. We suggested the notion of a "least" element, but this only makes sense in the context of an ordering. So, we introduce a partial order which can be seen as the information ordering. Elements that are greater than other elements in this ordering can be seen as being "more defined" than the lesser elements. In classical mathematics, this could be defined as:

$$x \sqsubseteq y = (x \equiv \bot) \vee (x \equiv y)$$

This definition is useful for intuition, but is not useful in our constructive setting as discussed by Tom de Jong [Jon19]. Later, we will reinterpret this definition using the lifting monad.

A domain for the PCF type ι would contain the element $\bot$ representing the non-terminating programs of type ι. $\bot$ would be less than every other element in the domain. The domain would also contain the natural numbers, which are all incomparable to each other as they are the total elements of the domain, but greater than the element $\bot$. We will call this domain $\mathbb{N}_\bot$, which can be seen in Figure 1.

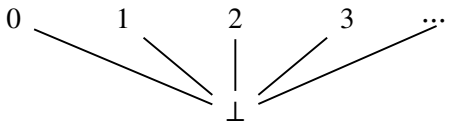

Figure 1: $\mathbb{N}_\bot$

After defining how we could represent programs of a base type, we next turn to modelling functions. Following from our intuition of definedness, we first consider which order we should associate with

the function space. For two functions $f, g : A \to B$, where A and B are domains, we define the pointwise order between these two functions as:

$$f \sqsubseteq g = \forall (x : A) \to f\,x \sqsubseteq g\,x$$

Intuitively, this means that functions which are less defined are below functions which are more defined.

Let's consider the function space $\mathbb{N}_\perp \to \mathbb{N}_\perp$. Under the pointwise order, it should be apparent that we will no longer have a flat domain like $\mathbb{N}_\perp$. Consider the identity function. Let's call id_k the identity function that's defined only up to the first k natural numbers, and every other natural number input is undefined. From the pointwise ordering, $\mathrm{id}_k \sqsubseteq \mathrm{id}_{k+1}$. The identity function that's defined for all natural numbers we shall call id_∞. We can then view this as a chain, from the function $\perp$ which maps all elements of $\mathbb{N}_\perp$ to $\perp_\mathbb{N}$, where $\perp_\mathbb{N}$ is the bottom element of the domain $\mathbb{N}_\perp$, up to id_∞ at the very top of this chain. We can think of this as the higher up the chain we go, the more defined the function is, and the better it is as an approximation of the identity function, until we get to id_∞. This function is the least upper bound of this chain.

One can also consider that we can approach id_∞ from another direction, where we first build up the function in another order rather than id_0, id_1, etc. We can view all these chains as forming a directed set where id_∞ is the least upper bound. From this, our intuition becomes that we can represent our types as directed-complete partial orders, which we define concretely later.

One note is that in our developments, we work with families instead of subsets. We do this because they were found to be more convenient in our constructive setting [Jon19].

Definition 5.1. An indexed family, or just family, is a function from an index set I to a set X.

We can view the image of the function as defining a subset of a set X. There is also the notion of a pointwise family, which we shall use later in our developments.

Definition 5.2. Given a family $\alpha : I \to (D \to E)$, and an element $d : D$, we can form the pointwise family $\lambda i.\alpha(i)(d)$.

However, we cannot simply allow every possible function between two domains. Consider a function from $\mathbb{N}_\perp$ to $\mathbb{N}_\perp$ which maps $\perp$ to 1 and all natural numbers n to $n + 1$. There is a fundamental issue with such a construction in that we can use it to solve the halting problem. A similar issue is discussed by Knapp and Escardó [EK17]. Therefore, we endow our interpretation of functions with some extra constraints.

The first constraint is that we consider monotone functions only. A way of viewing this constraint is that if we provide more information in the input, then we get more information in the output.

Definition 5.3. A function f is monotone if $f\,x \sqsubseteq f\,y$ when $x \sqsubseteq y$.

The problematic function we considered earlier is no longer a valid consideration, as we can consider that in the domain $\mathbb{N}_\perp$, for all n it holds that $\perp \sqsubseteq n$. The function is not monotonic, since $f \perp \not\sqsubseteq f n$.

The second constraint arises when we consider recursion. Fix f returns the least fixed point of the function f. In general, such a fixed point is not guaranteed to exist. To rectify this, we add the constraint that functions must be continuous, as it can be shown that continuous functions have fixed points [Plo83].

Definition 5.4. A function $f : A \to B$ is Scott-continuous if for any directed family $\alpha : I \to A$, $f(\coprod_{i:I} \alpha(i)) \equiv \coprod_{i:I} f(\alpha(i))$.

Monotonicity follows from continuity, so we only need to consider functions which are continuous [Str06, Section 4]. It can also be shown that all computable functions are continuous [Wei95], so we can be sure that all functions in PCF can be interpreted by this model.

5.2 Defining DCPOs

Definition 5.5. We define a poset $(X, \sqsubseteq)$ to be a set X with a proposition-valued binary relation $\sqsubseteq$, such that $\sqsubseteq$ is:

- Reflexive - $\forall(x : X) \to x \sqsubseteq x$
- Antisymmetric - $\forall(x, y : X) \to x \sqsubseteq y \to y \sqsubseteq x \to x \equiv y$
- Transitive - $\forall(x, y, z : X) \to x \sqsubseteq y \to y \sqsubseteq z \to x \sqsubseteq z$

Definition 5.6. Given a poset P, an element u of P is called an upper bound of a family $\alpha : I \to P$ if $\forall(i : I) \to \alpha(i) \sqsubseteq u$.

Definition 5.7. Given a poset P, an upper bound b of a family $\alpha : I \to P$ is called a least upper bound, or a supremum, if for any other upper bound u of α, it holds that $b \sqsubseteq u$.

Definition 5.8. A family $\alpha : I \to P$ of a poset P is called directed if the type I is inhabited, and $\forall(i, j : I) \to \exists(k : I) \to \alpha(i) \sqsubseteq \alpha(k) \wedge \alpha(j) \sqsubseteq \alpha(k)$.

We give the definition of is-directed in Agda, as it makes considerable use of propositional truncation.

```
is-directed : {I : 𝒱 ̇ } → (I → D) → 𝒱 ⊔ 𝒯 ̇
is-directed {I} α = ∥ I ∥ × ((i j : I) → ∃ \(k : I) → (α i ⊑ α k) × (α j ⊑ α k))
```

The first projection of is-directed α represents the proof that the index set I is inhabited. It is the propositional truncation of the set I, as defined in Section 2.4. We use this as we only care about the fact that an element of I exists - we do not care which, i.e. all elements are equal proofs of the inhabitance of I.

The second projection is our second property, that for all i, j, there exists a k such that $\alpha(i) \sqsubseteq \alpha(k)$ and $\alpha(j) \sqsubseteq \alpha(k)$. In this definition, $\exists$ is the propositional truncation of the dependent pair, which is again explained in Section 2.4. As such, a proof of is-directed α does not give a particular k for each i, j, just the knowledge that one exists.

Definition 5.9. A poset P is called directed-complete if every directed family in P has a least upper bound in P.

These definitions have been formalised in Agda by Tom de Jong, which we shall be using as a basis for further constructions. A DCPO can be defined in Agda, omitting the prior definitions, as:

```
module _ {𝓤 𝓣 : Universe} {D : 𝓤 ̇ } (_⊑_ : D → D → 𝓣 ̇ ) where

  dcpo-axioms : 𝓤 ⊔ (𝓥 ⁺) ⊔ 𝓣 ̇ -- Prior definitions omitted for brevity.
  dcpo-axioms = is-set D × is-prop-valued × is-reflexive
                × is-transitive × is-antisymmetric × is-directed-complete

module _ {𝓤 𝓣 : Universe} where

  DCPO-structure : 𝓤 ̇ → 𝓤 ⊔ (𝓥 ⁺) ⊔ (𝓣 ⁺) ̇
  DCPO-structure D = Σ \(_⊑_ : D → D → 𝓣 ̇ ) → dcpo-axioms {𝓤} {𝓣} _⊑_

  DCPO : (𝓤 ⁺) ⊔ (𝓥 ⁺) ⊔ (𝓣 ⁺) ̇
  DCPO = Σ \(D : 𝓤 ̇ ) → DCPO-structure D
```

We note the use of anonymous modules in Agda. Anonymous modules are useful when defining many statements which rely on the same assumptions. For example, is-reflexive, is-antisymmetric, and is-transitive all rely on the assumption of a relation $\sqsubseteq$ to define a property on. Therefore, we can make the assumption as a module parameter, saving us from having to add it as a parameter to each of the individual definitions explicitly. We can see this in action when we define DCPO-structure in terms of the dcpo-axioms. We do not explicitly define in the definition of dcpo-axioms that we must pass the relation $\sqsubseteq$, but it is required due to the parameter `_⊑_ : D → D → 𝓣 ̇` of the anonymous module which dcpo-axioms resides in. During this document, we will try to omit the anonymous module definitions where possible, as they do not add much to the understanding, but it is useful to see where some parameters to functions are coming from.

5.3 DCPOs with $\bot$

From our intuition, we will be representing types as DCPOs with the least element $\bot$ to represent undefined. Therefore, we need a way to represent this, so we next construct a representation of DCPOs with a least element. We begin by defining what it means to be a least element. In Agda, we define that an element x of D is least if for any other element y of D, $x \sqsubseteq y$. We then use the dependent pair to construct has-least, which represents a proof that there is an element x with the property that it is the least element of D.

```
is-least : D → 𝓤 ⊔ 𝓣 ̇
is-least x = ∀ (y : D) → x ⊑ y

has-least : 𝓤 ⊔ 𝓣 ̇
has-least = Σ (\(x : D) → is-least x)
```

Next, we define DCPO⊥ to represent a DCPO with a least element.

```
DCPO⊥ : (𝓥 ⁺) ⊔ (𝓤 ⁺) ⊔ (𝓣 ⁺)̇
DCPO⊥ = Σ \(𝒟 : DCPO) → has-least (underlying-order 𝒟)
```

5.4 DCPOs of continuous functions

We can implement our definition of continuity in Agda as follows:

```
is-continuous : (𝒟 : DCPO {𝓤} {𝓣}) (ℰ : DCPO {𝓤'} {𝓣'})
              → (⟨ 𝒟 ⟩ → ⟨ ℰ ⟩) → 𝓥 ⁺ ⊔ 𝓤 ⊔ 𝓣 ⊔ 𝓤' ⊔ 𝓣'
is-continuous 𝒟 ℰ f = (I : 𝓥̇) (α : I → ⟨ 𝒟 ⟩) (δ : is-Directed 𝒟 α)
                    → is-sup (underlying-order ℰ) (f (∐ 𝒟 δ)) (f ∘ α)
```

We omit the definition of is-sup, but is-sup ⊑ b α is a proof that b is a least upper bound of the family α, with respect to the order ⊑.

We then define continuous functions in Agda. We follow the definitions given in the previous section for continuity and monotonicity. Since monotonicity follows from continuity, we only need to ensure that each function is continuous. So, we can define our continuous functions as follows, where the term ⟨ 𝒟 ⟩ refers to the underlying set of the DCPO 𝒟:

```
DCPO[_,_] : DCPO {𝓤} {𝓣} → DCPO {𝓤'} {𝓣'} → 𝓥 ⁺ ⊔ 𝓤 ⊔ 𝓣 ⊔ 𝓤' ⊔ 𝓣'
DCPO[ 𝒟 , ℰ ] = Σ (\(f : ⟨ 𝒟 ⟩ → ⟨ ℰ ⟩) → is-continuous 𝒟 ℰ f)
```

We then define the continuous functions for DCPO⊥. This is more for convenience than anything else, as DCPO⊥ is just a DCPO with a proof that it does indeed have a bottom element. We use ⟪ 𝒟 ⟫ to extract the DCPO from DCPO⊥.

```
DCPO⊥[_,_] : DCPO⊥ {𝓤} {𝓣} → DCPO⊥ {𝓤'} {𝓣'} → (𝓥 ⁺) ⊔ 𝓤 ⊔ 𝓣 ⊔ 𝓤' ⊔ 𝓣'
DCPO⊥[ 𝒟 , ℰ ] = DCPO[ ⟪ 𝒟 ⟫ , ⟪ ℰ ⟫ ]
```

The pointwise order can be formalised as the type from elements d of the underlying set of a DCPO 𝒟 to a proof of $f\ d \sqsubseteq\langle\ \mathcal{E}\ \rangle\ g\ d$, where ⊑⟨ ℰ ⟩ refers to the underlying order for the DCPO ℰ.

```
_hom-⊑_ : DCPO[ 𝒟 , ℰ ] → DCPO[ 𝒟 , ℰ ] → 𝓤 ⊔ 𝓣'
(f , _) hom-⊑ (g , _) = ∀ d → f d ⊑⟨ ℰ ⟩ g d
```

Now we have defined the set of continuous functions between two DCPOs, we can next show this produces a DCPO when accompanied by the pointwise order. Since this is a previous development by Tom de Jong, we omit the proof of the DCPO axioms, but show the type of the definition and the underlying order used.

```
_⟹ᵈᶜᵖᵒ_ : DCPO {𝓤} {𝓣} → DCPO {𝓤'} {𝓣'}
         → DCPO {(𝓥 ⁺) ⊔ 𝓤 ⊔ 𝓣 ⊔ 𝓤' ⊔ 𝓣'} {𝓤 ⊔ 𝓣'}
𝒟 ⟹ᵈᶜᵖᵒ ℰ = DCPO[ 𝒟 , ℰ ] , _⊑_ , d -- Proof of DCPO axioms d omitted.
  where
    _⊑_ = 𝒟 hom-⊑ ℰ
```

5.5 The product of two DCPOs is a DCPO

One construction which we have to formalise is the product between DCPOs. This is a new formalisation by us, as it was not needed in Tom de Jong's implementation since he considers a combinatorial implementation of PCF. We will, however, need the product when constructing our implementation of contexts, as we shall see when defining the Scott model of PCF.

The underlying set of the product between DCPOs will simply be the Cartesian product between the underlying sets. For the order, we shall use the component-wise ordering. Between DCPOs $\mathscr{D}$ and $\mathscr{E}$, this can be defined as:

$$(a, b) \sqsubseteq_{\mathscr{D}\times\mathscr{E}} (c, d) = (a \sqsubseteq_{\mathscr{D}} c) \wedge (b \sqsubseteq_{\mathscr{E}} d)$$

In Agda, this can be represented as a type where its inhabitants are a pair of proofs, one which states $a \sqsubseteq\langle\ \mathscr{D}\ \rangle\ c$, and another that $b \sqsubseteq\langle\ \mathscr{E}\ \rangle\ d$. However, we need not define this order to depend on the definition of DCPOs. We can just define the function that says given a relation R_1 on the type D, and a relation R_2 on the type E, then two pairs from type $D \times E$ are related if the first projections are related by R_1, and the second projections are related by R_2.

```
_⊑-×_ : (D → D → 𝒯 ̇) → (E → E → 𝒯 ′) → (D × E → D × E → 𝒯 ⊔ 𝒯 ′)
_⊑-×_ R₁ R₂ (a , b) (c , d) = R₁ a c × R₂ b d
```

Before we show that, from two DCPOs, the Cartesian product between the two underlying sets with the component-wise ordering forms a DCPO, we need to show that composing the first projection with a directed family is also a directed family, and similarly for the second projection.

Lemma 5.10. *If we have an order $\sqsubseteq_D$ on a type D, an order $\sqsubseteq_E$ on a type E, and a directed family $\alpha : I \to D \times E$ under the component-wise ordering $\sqsubseteq_{D\times E}$, then we can form the directed family $pr_1 \circ \alpha : I \to D$ under the order $\sqsubseteq_D$.*

Proof. We first define what we are trying to prove. This is fairly similar to our definition on paper. The only difference is that we've used superscripts instead of subscripts to identify the orders. This is due to it not being possible to type $\sqsubseteq_d$ in Agda.

```
pr₁∘α-is-directed : {I : 𝓥 ̇} → {α : I → D × E} → (_⊑ᵈ_ : D → D → 𝒯 ̇) → (_⊑ᵉ_ : E → E → 𝒯 ′)
                  → is-directed (_⊑ᵈ_ ⊑-× _⊑ᵉ_) α → is-directed _⊑ᵈ_ (pr₁ ∘ α)
```

Next, we give our proof of pr₁∘α-is-directed. We use underscores to avoid giving names to the parameters we do not use in the body of our proof. δ refers to the proof that a family α is directed. Since we are trying to prove that $\text{pr}_1 \circ \alpha$ is directed, we provide a pair as the result of the function. The first component is trivial - it is a proof that type I is inhabited. This is given to us by the fact that α is directed. More specifically, in our definition it is the first projection of δ, but to make our proof more readable, we use a function is-directed-gives-inhabited to extract this fact. The proof of our second property is more involved, so we prove it under a where clause so our proof is easier to read.

```
pr₁∘α-is-directed {_} {_} {I} {α} _⊑ᵈ_ _⊑ᵉ_ δ
                  = is-directed-gives-inhabited (_⊑ᵈ_ ⊑-× _⊑ᵉ_) α δ , o
  where
```

From our definition of directedness in Agda, δ does not actually provide us with a k for a given i, j such that $\alpha(i) \sqsubseteq_{D \times E} \alpha(k)$ and $\alpha(j) \sqsubseteq_{D \times E} \alpha(k)$, just with the knowledge of its existence. However, we are able to use ∥∥-functor to reason about this hypothetical k, and conclude that is is the same k such that $(pr_1 \circ \alpha)i \sqsubseteq_D (pr_1 \circ \alpha)k$ and $(pr_1 \circ \alpha)j \sqsubseteq_D (pr_1 \circ \alpha)k$, as we can extract these proofs from our assumption that α is directed.

```
    o : (i j : I) → ∃ (λ k → ((pr₁ ∘ α) i) ⊑ᵈ ((pr₁ ∘ α) k) × ((pr₁ ∘ α) j) ⊑ᵈ ((pr₁ ∘ α) k))
    o i j = ∥∥-functor f (is-directed-order (_⊑ᵈ_ ⊑-× _⊑ᵉ_) α δ i j)
      where
        f : Σ (λ k → (_⊑ᵈ_ ⊑-× _⊑ᵉ_) (α i) (α k) × (_⊑ᵈ_ ⊑-× _⊑ᵉ_) (α j) (α k))
          → Σ (λ v → Σ (λ v₁ → pr₁ (α j) ⊑ᵈ pr₁ (α v)))
        f (k , (i₁⊑k₁ , i₂⊑k₂) , (j₁⊑k₁ , j₂⊑k₂)) = k , i₁⊑k₁ , j₁⊑k₁
```
□

Lemma 5.11. *If we have an order $\sqsubseteq_D$ on a type D, an order $\sqsubseteq_E$ on a type E, and a directed family $\alpha : I \to D \times E$ under the component-wise ordering $\sqsubseteq_{D \times E}$, then we can form the directed family $pr_2 \circ \alpha : I \to E$ under the order $\sqsubseteq_E$.*

Proof. This proof follows the same process as the previous, apart from we take the second projection of the proofs that $\alpha(i) \sqsubseteq_{D \times E} \alpha(k)$ and $\alpha(j) \sqsubseteq_{D \times E} \alpha(k)$. □

Now we construct the product for DCPOs.

Proposition 5.12. *Given a DCPO $\mathscr{D}$ with the underlying set D and order $\sqsubseteq_D$, and a DCPO $\mathscr{E}$ with the underlying set E and order $\sqsubseteq_E$, the Cartesian product of the underlying sets with the component-wise ordering forms a DCPO $\mathscr{D} \times^{DCPO} \mathscr{E}$.*

Proof. We begin our proof by stating that our new DCPO will have an underlying set of the Cartesian product between the underlying sets of the DCPOs $\mathscr{D}$ and $\mathscr{E}$. We also say that the order of this new DCPO is the component-wise order, which we construct from the underlying orders of $\mathscr{D}$ and $\mathscr{E}$, using ⊑-×. We give this order an alias of ⊑-D×E, which we use as an infix operator.

```
_×ᵈᶜᵖᵒ_ : DCPO {𝒰} {𝒯} → DCPO {𝒰 '} {𝒯 '} → DCPO {𝒰 ⊔ 𝒰 '} {𝒯 ⊔ 𝒯 '}
𝒟 ×ᵈᶜᵖᵒ ℰ = (⟨ 𝒟 ⟩ × ⟨ ℰ ⟩) , _⊑-D×E_ , axioms
  where
    _⊑-D×E_ = (underlying-order 𝒟) ⊑-× (underlying-order ℰ)
    axioms : dcpo-axioms _⊑-D×E_
    axioms = s , p , r , t , a , c -- Proofs s, p, r, t, a omitted.
      where
```

We next move into the proofs of the axioms. Directed-completeness requires more thought than

the others which are more trivial, so we only show the proof c. We first make use of our proofs that if α is a directed family, then the projection maps composed with α also form a directed family. We use these proofs to construct our least upper bound component wise, as we now have access to two least upper bounds by our assumption that both $\mathscr{D}$ and $\mathscr{E}$ are directed-complete. We then provide a proof is-lub, which is a proof that our constructed least upper bound is indeed a least upper bound of the directed family α.

```
c : is-directed-complete _⊑-D×E_
c I α δ = (∐ 𝒟 pr₁∘α-is-dir , ∐ ℰ pr₂∘α-is-dir) , is-lub
  where
    pr₁∘α-is-dir : is-Directed 𝒟 (pr₁ ∘ α)
    pr₁∘α-is-dir = pr₁∘α-is-directed (underlying-order 𝒟) (underlying-order ℰ) δ
    pr₂∘α-is-dir : is-Directed ℰ (pr₂ ∘ α)
    pr₂∘α-is-dir = pr₂∘α-is-directed (underlying-order 𝒟) (underlying-order ℰ) δ
```

We provide a proof for each of the conditions that an element requires to be a least upper bound. The first is that it's an upper bound, and the second that it's the least of all upper bounds.

```
is-lub : is-sup _⊑-D×E_ (∐ 𝒟 pr₁∘α-is-dir , ∐ ℰ pr₂∘α-is-dir) α
is-lub = u , v
  where
```

We first consider showing that our constructed least upper bound is indeed an upper bound. This requires constructing a proof that for every $i : I$, $\mathrm{pr}_1\ (\alpha\ i) \sqsubseteq\langle\ \mathscr{D}\ \rangle \coprod \mathscr{D}$ pr₁∘α-is-directed, and also $\mathrm{pr}_2\ (\alpha\ i) \sqsubseteq\langle\ \mathscr{E}\ \rangle \coprod \mathscr{E}$ pr₂∘α-is-directed. This is trivial - following from our assumption that $\mathscr{D}$ and $\mathscr{E}$ are directed-complete, we can construct both of these proofs.

```
u : is-upperbound _⊑-D×E_ (∐ 𝒟 pr₁∘α-is-dir , ∐ ℰ pr₂∘α-is-dir) α
u i = (∐-is-upperbound 𝒟 pr₁∘α-is-dir i) , (∐-is-upperbound ℰ pr₂∘α-is-dir i)
```

We next show that it is also less than any other upper bounds of α under the component-wise ordering. For this, we assume an upper bound u, and the property that this u is an upper bound of the directed family α. We provide a pair of proofs, the first is that ∐ 𝒟 pr₁∘α-is-dir is less than $\mathrm{pr}_1\ u$, and the second that ∐ ℰ pr₂∘α-is-dir is less than $\mathrm{pr}_2\ u$.

```
v : (u : ⟨ 𝒟 ⟩ × ⟨ ℰ ⟩) → is-upperbound _⊑-D×E_ u α
                        → (∐ 𝒟 pr₁∘α-is-dir , ∐ ℰ pr₂∘α-is-dir) ⊑-D×E u
v u u-is-upperbound = lub-in-pr₁ , lub-in-pr₂
  where
```

From the definition of least upper bound, we have that ∐ 𝒟 pr₁∘α-is-dir is less than all upper bounds of $\mathrm{pr}_1 \circ \alpha$. We can show that $\mathrm{pr}_1\ u$ is an upper bound of $\mathrm{pr}_1 \circ \alpha$ as it follows simply from the definition of the component-wise ordering that if $\alpha\ i \sqsubseteq\text{-D×E}\ u$, then $\mathrm{pr}_1\ (\alpha\ i) \sqsubseteq\langle\ \mathscr{D}\ \rangle \mathrm{pr}_1\ u$. Similarly for the second projection.

```
lub-in-pr₁ = ∐-is-lowerbound-of-upperbounds 𝒟 pr₁∘α-is-dir (pr₁ u) p
  where
    p : is-upperbound (underlying-order 𝒟) (pr₁ u) (pr₁ ∘ α)
    p i = pr₁ (u-is-upperbound i)
lub-in-pr₂ = ∐-is-lowerbound-of-upperbounds ℰ pr₂∘α-is-dir (pr₂ u) p
  where
    p : is-upperbound (underlying-order ℰ) (pr₂ u) (pr₂ ∘ α)
    p i = pr₂ (u-is-upperbound i)                                          □
```

We then show that given two DCPOs with least elements, the product between these DCPOs also has a least element.

```
_×ᵈᶜᵖᵒ⊥_ : DCPO⊥ {𝒰} {𝒯} → DCPO⊥ {𝒰'} {𝒯'} → DCPO⊥ {𝒰 ⊔ 𝒰'} {𝒯 ⊔ 𝒯'}
𝒟 ×ᵈᶜᵖᵒ⊥ ℰ = (⟪ 𝒟 ⟫ ×ᵈᶜᵖᵒ ⟪ ℰ ⟫) , least , p
  where
```

The least element is a pair, where the first component is the least element from $\mathscr{D}$, and the second component is the least element from $\mathscr{E}$. The property of this construction being the least element follows trivially from the fact that each component is the least element of their respective DCPOs.

```
    least : ⟨ ⟪ 𝒟 ⟫ ×ᵈᶜᵖᵒ ⟪ ℰ ⟫ ⟩
    least = the-least 𝒟 , the-least ℰ
    p : is-least (underlying-order (⟪ 𝒟 ⟫ ×ᵈᶜᵖᵒ ⟪ ℰ ⟫)) least
    p (d , e) = (least-property 𝒟 d) , (least-property ℰ e)
```

5.6 Curry and uncurry

We can represent a function which takes multiple arguments as a function where its input is a pair consisting of these arguments. For example, a function which takes two arguments might have the general type $f : A \times B \to C$. However, we can construct a function $g : A \to (B \to C)$ such that for any pair (a, b), $f(a, b) \equiv g(a)(b)$. We can also show that given g, we can construct the function f. From this, we have shown that the function space $A \times B \to C$ is isomorphic to that of $A \to (B \to C)$, i.e. that they are in one-to-one correspondence. We have names for these particular operations. Constructing $A \to (B \to C)$ from $A \times B \to C$ is called currying, and from $A \to (B \to C)$ to $A \times B \to C$ is called uncurrying. We show that the currying of a continuous function produces another continuous function, and that uncurrying also produces a continuous function.

We make use of the following lemmas, which we have proved in Agda, but omitted due to length.

Lemma 5.13. *Given a continuous function* `f : DCPO[ 𝒟 ×ᵈᶜᵖᵒ ℰ , ℱ ]`*, then it is continuous in both of its arguments.*

Lemma 5.14. *Given a function* `f : ⟨ 𝒟 ×ᵈᶜᵖᵒ ℰ ⟩ → ⟨ ℱ ⟩`*, if it is continuous in both arguments*

then f is continuous.

Theorem 5.15 (Curry)**.** *From a continuous function of the form $f : \langle\ \mathscr{D} \times^{dcpo} \mathscr{E}\ \rangle \to \langle\ \mathscr{F}\ \rangle$, we can construct a continuous function of the form $g : \langle\ \mathscr{D}\ \rangle \to \langle\ \mathscr{E} \Longrightarrow^{dcpo} \mathscr{F}\ \rangle$, such that for all pairs $(d\ ,\ e)$, $f\ (d\ ,\ e) \equiv g\ d\ e$.*

Proof. We define curry^{dcpo} as follows. We pattern match on the continuous function definition, and give the name f to the underlying function, and *f-is-continuous* to the proof that f is continuous. We construct g as a function that, given $d : \langle\ \mathscr{D}\ \rangle$, produces a continuous function from $\mathscr{E}$ to $\mathscr{F}$. Although, this is just the same as our definition of continuous→continuous-in-pr$_2$ where we do not provide a particular $d : \langle\ \mathscr{D}\ \rangle$ to fix. We also provide a proof that g is continuous.

```
curry^dcpo : DCPO[ (𝒟 ×^dcpo ℰ) , ℱ ] → DCPO[ 𝒟 , ℰ ⟹^dcpo ℱ ]
curry^dcpo (f , f-is-continuous) = g , g-is-continuous
  where
    g : ⟨ 𝒟 ⟩ → ⟨ ℰ ⟹^dcpo ℱ ⟩
    g = continuous→continuous-in-pr₂ 𝒟 ℰ ℱ (f , f-is-continuous)
```

We now begin to construct our proof that g is continuous. We further break this proof down into a proof u that g $(\coprod\ \mathscr{D}\ \delta)$ is the upper bound of g ∘ α, and a proof v that for any other u_1 which is an upper bound of g ∘ α, then g $(\coprod\ \mathscr{D}\ \delta) \sqsubseteq\langle\ \mathscr{E} \Longrightarrow^{dcpo} \mathscr{F}\ \rangle\ u_1$.

```
    g-is-continuous : (I : 𝒱̇) (α : I → ⟨ 𝒟 ⟩) (δ : is-Directed 𝒟 α) →
                        is-sup (underlying-order (ℰ ⟹^dcpo ℱ)) (g (⨆ 𝒟 δ)) (g ∘ α)
    g-is-continuous I α δ = u , v
      where
```

We want to show that (g ∘ α) $i \sqsubseteq\langle\ \mathscr{E} \Longrightarrow^{dcpo} \mathscr{F}\ \rangle$ g $(\coprod\ \mathscr{D}\ \delta)$ for all i. By the definition of the pointwise order and the definition of g, this simplifies to showing $f\ (\alpha\ i\ ,\ e) \sqsubseteq\langle\ \mathscr{F}\ \rangle\ f\ (\coprod\ \mathscr{D}\ \delta\ ,\ e)$ for all e. Our proof begins by constructing a continuous function where the second argument is fixed to be e. We call this continuous function f-fixed-e. From the continuity of this function, it follows that the least upper bound is $f\ (\coprod\ \mathscr{D}\ \delta\ ,\ e)$, and proof of this we give the name p. We can then show $f\ (\alpha\ i\ ,\ e) \sqsubseteq\langle\ \mathscr{F}\ \rangle\ f\ (\coprod\ \mathscr{D}\ \delta\ ,\ e)$ by applying the definition of the least upper bound via is-sup-gives-upperbound.

```
      u : (i : I) → underlying-order (ℰ ⟹^dcpo ℱ) ((g ∘ α) i) (g (⨆ 𝒟 δ))
      u i e = is-sup-gives-is-upperbound (underlying-order ℱ) p i
        where
          f-fixed-e : DCPO[ 𝒟 , ℱ ]
          f-fixed-e = continuous→continuous-in-pr₁ 𝒟 ℰ ℱ (f , f-is-continuous) e
          p : is-sup (underlying-order ℱ) (f (⨆ 𝒟 δ , e)) (λ i → f (α i , e))
          p = continuity-of-function 𝒟 ℱ f-fixed-e I α δ
```

The proof that g $(\coprod\ \mathscr{D}\ \delta)$ is the least of all upper bounds of g ∘ α requires that we show for any u_1,

if u_1 is an upper bound of $g \circ \alpha$, then $g\ (\coprod\ \mathscr{D}\ \delta) \sqsubseteq\langle\ \mathscr{E} \Longrightarrow^{dcpo} \mathscr{F}\ \rangle\ u_1$. This simplifies to showing $f\ (\coprod\ \mathscr{D}\ \delta\ ,\ e) \sqsubseteq\langle\ \mathscr{F}\ \rangle$ underlying-function $\mathscr{E}\ \mathscr{F}\ u_1\ e$ for all e. Similar to the proof that $g\ (\coprod\ \mathscr{D}\ \delta)$ is an upper bound, we first construct f-fixed-e, and then prove the property p that $f\ (\coprod\ \mathscr{D}\ \delta\ ,\ e)$ is the least upper bound in the same way. We then apply the definition of least upper bound to show that $f\ (\coprod\ \mathscr{D}\ \delta\ ,\ e) \sqsubseteq\langle\ \mathscr{F}\ \rangle$ underlying-function $\mathscr{E}\ \mathscr{F}\ u_1\ e$ as desired.

```
v : (u₁ : ⟨ 𝓔 ⟹^dcpo 𝓕 ⟩) → ((i : I) → g (α i) ⊑⟨ 𝓔 ⟹^dcpo 𝓕 ⟩ u₁) →
      g (∐ 𝒟 δ) ⊑⟨ 𝓔 ⟹^dcpo 𝓕 ⟩ u₁
v u₁ upper e = is-sup-gives-is-lowerbound-of-upperbounds (underlying-order 𝓕) p
                 (underlying-function 𝓔 𝓕 u₁ e) (λ i → upper i e)
  where
    f-fixed-e : DCPO[ 𝒟 , 𝓕 ]
    f-fixed-e = continuous→continuous-in-pr₁ 𝒟 𝓔 𝓕 (f , f-is-continuous) e
    p : is-sup (underlying-order 𝓕) (f (∐ 𝒟 δ , e)) (λ i → f (α i , e))
    p = continuity-of-function 𝒟 𝓕 f-fixed-e I α δ                          □
```

We trivially extend this to a proof on DCPO⊥.

```
curry^dcpo⊥ : DCPO⊥[ 𝒟 ×^dcpo⊥ 𝓔 , 𝓕 ] → DCPO⊥[ 𝒟 , 𝓔 ⟹^dcpo⊥ 𝓕 ]
curry^dcpo⊥ f = curry^dcpo ⟪ 𝒟 ⟫ ⟪ 𝓔 ⟫ ⟪ 𝓕 ⟫ f
```

Theorem 5.16 (Uncurry)**.** *From a continuous function of the form $g : \langle\ \mathscr{D}\ \rangle \to \langle\ \mathscr{E} \Longrightarrow^{dcpo} \mathscr{F}\ \rangle$, we can construct a continuous function of the form $f : \langle\ \mathscr{E} \times^{dcpo} \mathscr{E}\ \rangle \to \langle\ \mathscr{F}\ \rangle$, such that for all d, and for all e, $g\ d\ e \equiv f\ (d\ ,\ e)$.*

Proof. We begin similarly to the curry proof, by pattern matching on the continuous function we assume. We then provide a function f, which is defined as taking a pair $(d\ ,\ e) : \langle\ \mathscr{D} \times^{dcpo} \mathscr{E}\ \rangle$ and applying each argument in turn to g. We also provide a proof of continuity by Lemma 5.14.

```
uncurry^dcpo : DCPO[ 𝒟 , 𝓔 ⟹^dcpo 𝓕 ] → DCPO[ (𝒟 ×^dcpo 𝓔) , 𝓕 ]
uncurry^dcpo (g , g-is-continuous) = f , c
  where
    f : ⟨ 𝒟 ×^dcpo 𝓔 ⟩ → ⟨ 𝓕 ⟩
    f (d , e) = underlying-function 𝓔 𝓕 (g d) e
    c : is-continuous (𝒟 ×^dcpo 𝓔) 𝓕 f
    c = continuous-in-arguments→continuous 𝒟 𝓔 𝓕 f continuous-in-pr₁ continuous-in-pr₂
      where
```

Since all continuous functions are monotone, it follows that g is monotone.

```
g-is-monotone : is-monotone 𝒟 (𝓔 ⟹^dcpo 𝓕) g
g-is-monotone = continuous-functions-are-monotone 𝒟 (𝓔 ⟹^dcpo 𝓕) (g , g-is-continuous)
```

We then provide a proof that f is continuous in its first argument. We define this in terms of a

proof u that underlying-function $\mathscr{E}$ $\mathscr{F}$ (g ($\coprod$ $\mathscr{D}$ δ)) e is the upper bound of the pointwise family formed by $g \circ \alpha$ and e, and a proof v that underlying-function $\mathscr{E}$ $\mathscr{F}$ (g ($\coprod$ $\mathscr{D}$ δ)) e is less than any other upper bounds of the pointwise family formed by $g \circ \alpha$ and e.

```
continuous-in-pr₁ : (e : ⟨ ℰ ⟩)
                  → is-continuous 𝒟 ℱ (λ d → underlying-function ℰ ℱ (g d) e)
continuous-in-pr₁ e I α δ = u , v
  where
```

The proof u follows easily from the fact that g is monotone. We use the definition of the least upper bound to give us a proof that $\alpha\ i$ is less than $\coprod$ $\mathscr{D}$ δ for all i. From the monotonicity of g, we show that g ($\alpha\ i$) is less than g ($\coprod$ $\mathscr{D}$ δ). By the definition of the pointwise order, we apply e to the proof we have constructed so far to achieve g ($\alpha\ i$) applied to e is less than g ($\coprod$ $\mathscr{D}$ δ) applied to e.

```
    u : is-upperbound (underlying-order ℱ) (underlying-function ℰ ℱ (g (∐ 𝒟 δ)) e)
                                           (pointwise-family ℰ ℱ (g ∘ α) e)
    u i = g-is-monotone (α i) (∐ 𝒟 δ) (∐-is-upperbound 𝒟 δ i) e
```

We next construct our proof v. We assume u_1, and a proof p that it is an upper bound. We begin our proof in the where clause, where we first construct a proof that $g \circ \alpha$ is a directed family, and that the pointwise family of $g \circ \alpha$ and e is directed. We use proofs that Tom de Jong has previously constructed to show these.

```
    v : (u₁ : ⟨ ℱ ⟩) → ((i : I) → (underlying-function ℰ ℱ ((g ∘ α) i) e) ⊑⟨ ℱ ⟩ u₁)
      → (underlying-function ℰ ℱ (g (∐ 𝒟 δ)) e) ⊑⟨ ℱ ⟩ u₁
    v u₁ p = γ
      where
        ⟨g∘α⟩-is-directed : is-Directed (ℰ ⟹^dcpo ℱ) (g ∘ α)
        ⟨g∘α⟩-is-directed = image-is-directed 𝒟 (ℰ ⟹^dcpo ℱ) (g , g-is-continuous) δ
        ⟨g∘α⟩e-is-directed : is-Directed ℱ (pointwise-family ℰ ℱ (g ∘ α) e)
        ⟨g∘α⟩e-is-directed = pointwise-family-is-directed ℰ ℱ (g ∘ α) ⟨g∘α⟩-is-directed e
```

We next show a proof i, that $\coprod$ $\mathscr{F}$ ⟨g∘α⟩e-is-directed is less than u_1. This follows from the definition of the least upper bound.

```
        i : (∐ ℱ ⟨g∘α⟩e-is-directed) ⊑⟨ ℱ ⟩ u₁
        i = ∐-is-lowerbound-of-upperbounds ℱ ⟨g∘α⟩e-is-directed u₁ p
```

The continuity of g produces the proof ii. We then use the congruence of equality to show underlying-function $\mathscr{E}$ $\mathscr{F}$ (g ($\coprod$ $\mathscr{D}$ δ)) e is equal to underlying-function $\mathscr{E}$ $\mathscr{F}$ ($\coprod$ ($\mathscr{E}$ $\Longrightarrow^{dcpo}$ $\mathscr{F}$) ⟨g∘α⟩-is-directed) e. However, underlying-function $\mathscr{E}$ $\mathscr{F}$ ($\coprod$ ($\mathscr{E}$ $\Longrightarrow^{dcpo}$ $\mathscr{F}$) ⟨g∘α⟩-is-directed) e is the same as $\coprod$ $\mathscr{F}$ ⟨g∘α⟩e-is-directed, since the least upper bound is defined pointwise. From congruence, and Agda applying the definition that the least upper bound is constructed pointwise, we achieve iii.

```
        ii : g (∐ 𝒟 δ) ≡ ∐ (ℰ ⟹^dcpo ℱ) ⟨g∘α⟩-is-directed
```

ii = continuous-function-$\coprod$-≡ $\mathscr{D}$ ($\mathscr{E}$ $\Longrightarrow^{dcpo}$ $\mathscr{F}$) (g , *g-is-continuous*) δ
iii : underlying-function $\mathscr{E}$ $\mathscr{F}$ (g ($\coprod$ $\mathscr{D}$ δ)) e ≡ $\coprod$ $\mathscr{F}$ ⟨g∘α⟩e-is-directed
iii = ap (λ - → underlying-function $\mathscr{E}$ $\mathscr{F}$ - e) ii

Since we have the proof i, we can transport the equality iii to produce the proof we desire. We use back-transport as we are required to show the property for the first operand of the equality, from the second operand possessing it.

γ : underlying-function $\mathscr{E}$ $\mathscr{F}$ (g ($\coprod$ $\mathscr{D}$ δ)) e ⊑⟨ $\mathscr{F}$ ⟩ u_1
γ = back-transport (λ - → - ⊑⟨ $\mathscr{F}$ ⟩ u_1) iii i

continuity-in-pr$_2$ is just the fact that g d is continuous.

continuous-in-pr$_2$: (d : ⟨ $\mathscr{D}$ ⟩) → is-continuous $\mathscr{E}$ $\mathscr{F}$ (underlying-function $\mathscr{E}$ $\mathscr{F}$ (g d))
continuous-in-pr$_2$ d = continuity-of-function $\mathscr{E}$ $\mathscr{F}$ (g d) □

Again, we trivially extend this to DCPO⊥.
uncurry^{dcpo}⊥ : DCPO⊥[$\mathscr{D}$, $\mathscr{E}$ $\Longrightarrow^{dcpo}$⊥ $\mathscr{F}$] → DCPO⊥[$\mathscr{D}$ ×^{dcpo}⊥ $\mathscr{E}$, $\mathscr{F}$]
uncurry^{dcpo}⊥ f = uncurry^{dcpo} ⟪ $\mathscr{D}$ ⟫ ⟪ $\mathscr{E}$ ⟫ ⟪ $\mathscr{F}$ ⟫ f

6 Scott Model of PCF

With our constructions in domain theory, we now have enough to define the Scott model of PCF [Sco93].

6.1 Types

We define the following function from PCF types to their denotational interpretation:
⟦_⟧ : type → DCPO⊥ {$\mathcal{U}_1$} {$\mathcal{U}_1$}
We have two cases we need to provide an interpretation for. The first is the base type, and the second is the function type.

The base type

We first construct the representation of our base type ι. We follow our intuitions from Section 5 that we represent ι as a DCPO, where the underlying set is $\mathbb{N} \cup \bot$, with the information ordering.

In our construction, to embed the notion of an undefined element into our set, we use the lifting monad as Tom de Jong does [Jon19]. The lifting monad, from a type such as $\mathbb{N}$, construct a new type $\mathscr{L}$ $\mathbb{N}$, called the "lifting" of $\mathbb{N}$. The lifting of $\mathbb{N}$ gives us a way to express the definedness of elements, allowing us to say that all elements from the original type $\mathbb{N}$ are "total" (defined), and there is an undefined element $\bot$.

The importance of using the lifting monad is it gives us a constructive way to represent the definedness of elements, and from this we can proceed to define an order which is compatible with

our developments:

$$x \sqsubseteq y = \text{is-defined}\, x \to x \equiv y$$

This order follows our intuition that for flat domains, $\bot$ is less than all other elements, and total elements are incomparable.

It also follows that a lifted type $\mathscr{L}\, Y$ forms a DCPO under the above ordering if the type Y is a set, as shown by Knapp and Escardó [EK17]. This construction of a DCPO from a lifted type has been developed in Agda by Tom de Jong. We shall be making use of $\mathscr{L}$-DCPO$\bot$, which, given a set Y, will produce a DCPO with least element $\bot$ under the mentioned ordering, where the underlying set is the lifting of Y. We can produce the total elements of the lifting of Y in Agda from an element n of the type Y as $\eta\ n$, and there is always the undefined element $\bot$ as previously discussed.

We can now construct the interpretation of the base type ι. We use the fact that $\mathbb{N}$ is a set from Martín Escardó's developments. For shorthand, we will use $\mathscr{L}^d\mathbb{N}$ to refer to the DCPO of the lifted natural numbers.

$[\![\ \iota\ ]\!] = \mathscr{L}^d\mathbb{N}$

Function types

We then represent function type $\sigma \Rightarrow \tau$ as a DCPO with $\bot$ where the underlying set is continuous functions from the interpretation of σ and the interpretation of τ.

$[\![\ \sigma \Rightarrow \tau\ ]\!] = [\![\ \sigma\ ]\!] \Longrightarrow^{dcpo}\bot\ [\![\ \tau\ ]\!]$

6.2 Contexts

We interpret contexts simply as a product between the interpretation of each type within the context.

The first consideration is to develop a representation for the empty context. We form a DCPO where the underlying set is $\mathbb{1}$ - the type with one element, namely $*$. We associate an order which states for any x and y of type $\mathbb{1}$, we have $x \sqsubseteq\langle \top \rangle\ y$. The set $\mathbb{1}$ and this order form a DCPO, which we name in Agda as $\top^{dcpo} : \text{DCPO}\ \{\mathcal{U}_1\}\ \{\mathcal{U}_1\}$. We omit the trivial definition and proof of DCPO axioms for brevity. It is also trivial to show this DCPO contains a least element, namely $*$, which we show in Agda in the proof $\top^{dcpo}\bot : \text{DCPO}\bot\ \{\mathcal{U}_1\}\ \{\mathcal{U}_1\}$.

We now can inductively define our interpretation of a context. For our inductive case where we have a context Γ ' x, we make use of $\times^{dcpo}\bot$ to form the product between the interpretation of Γ and the interpretation of the type x.

$【\_】 : \{n : \mathbb{N}\}\ (\Gamma : \text{Context}\ n) \to \text{DCPO}\bot\ \{\mathcal{U}_1\}\ \{\mathcal{U}_1\}$

$【\ \langle\rangle\ 】 = \top^{dcpo}\bot$

$【\ \Gamma\ '\ x\ 】 = 【\ \Gamma\ 】\ \times^{dcpo}\bot\ [\![\ x\ ]\!]$

From this definition, we can think of $d : \langle \ll \llbracket \Gamma \rrbracket \gg \rangle$ as representing a list of values for our variables in the context Γ to be substituted with in our denotational interpretation.

6.3 Terms

We next inductively define the interpretation of terms. Since terms may contain free variables, we interpret a term of type σ under a given context Γ as a continuous function from the interpretation of the context to the interpretation of σ. Intuitively, this means that given a list of values of the types listed in Γ, we can produce a closed term belonging to the type σ. We begin by first developing some constructs which we will use in our interpretation function.

Interpreting variables

When interpreting the term v i which has the type σ under a context Γ, it makes sense that this is essentially the i-th projection of an inhabitant of $\langle \ll \llbracket \Gamma \rrbracket \gg \rangle$. As such, we can see this as a continuous function between $\llbracket \Gamma \rrbracket$ and $[\![\sigma]\!]$.

We implement this i-th projection in Agda by induction on the lookup judgement. We take the second projection after taking i first projections, to extract the desired value.

```
extract : {n : ℕ} {σ : type} {Γ : Context n} → (x : Γ ∋ σ) → ⟨ ⟪ ⟦ Γ ⟧ ⟫ ⟩ → ⟨ ⟪ ⟦ σ ⟧ ⟫ ⟩
extract Z (_ , a) = a
extract (S x) (d , _) = extract x d
```

Since in our setting we require that functions are continuous, we need a proof of continuity. For any variable v x, we will want our underlying function to be extract x. As a result we need to show that for any lookup judgement x, extract x is a continuous function. We show this is the case by induction on the lookup judgement itself, as this determines the structure of extract x. The first case is when the lookup judgement is Z, and we can provide our value. In this case, our function is just pr_2, so we provide a proof that pr_2 is continuous.

```
extract-is-continuous : {n : ℕ} {Γ : Context n} {σ : type} (x : Γ ∋ σ)
                      → is-continuous ⟪ ⟦ Γ ⟧ ⟫ ⟪ ⟦ σ ⟧ ⟫ (extract x)
extract-is-continuous {_} {Γ , σ} {σ} Z = continuity-of-function ⟪ ⟦ Γ , σ ⟧ ⟫ ⟪ ⟦ σ ⟧ ⟫
                                          (pr₂-is-continuous ⟪ ⟦ Γ ⟧ ⟫ ⟪ ⟦ σ ⟧ ⟫)
```

In the inductive case, we have a context Γ , τ, and a proof $x : \Gamma \ni \sigma$. extract (S x) is the same as extract $x \circ \text{pr}_1$. Since we have a proof $\circ^{dcpo}$ that the composition of continuous functions produces a continuous function, we just need to prove that extract x is a continuous function, and that pr_1 is continuous. The fact that extract x is continuous is provided by our inductive hypothesis, and pr_1 is known to be continuous. As such, the continuity property of the composition of these continuous functions is enough to complete the proof.

```
extract-is-continuous {_} {Γ , τ} {σ} (S x)
  = continuity-of-function ⟪ ⟦ Γ , τ ⟧ ⟫ ⟪ ⟦ σ ⟧ ⟫ ( [ ⟪ ⟦ Γ , τ ⟧ ⟫ , ⟪ ⟦ Γ ⟧ ⟫ , ⟪ ⟦ σ ⟧ ⟫ ]
      extract x , (extract-is-continuous x) ∘^dcpo pr₁-is-continuous ⟪ ⟦ Γ ⟧ ⟫ ⟪ ⟦ τ ⟧ ⟫)
```

We then provide a function which, for any lookup judgement, provides a continuous function which extracts the desired value from a context.

```
var-extract : {n : ℕ} {σ : type} {Γ : Context n} → (x : Γ ∋ σ) → DCPO[ ⟪ 【 Γ 】 ⟫ , ⟪ ⟦ σ ⟧ ⟫ ]
var-extract x = extract x , extract-is-continuous x
```

Interpreting IfZero

In his implementation of the Scott model of a combinatorial version of PCF, Tom de Jong has developed an interpretation of IfZero in his setting. He defined the interpretation using the Kleisli extension as $\lambda x, y.(\chi_{x,y})^{\#}$, where:

$$\chi_{x,y}(n) :\equiv \begin{cases} x & \text{if } n = 0; \\ y & \text{else;} \end{cases}$$

However, since our implementation of PCF has contexts, we cannot just use Tom de Jong's interpretation for our IfZero. Instead, we construct a way to first evaluate our terms t, t_1, and t_2 which make up a term IfZero t t_1 t_2 under a context Γ.

We first define a continuous function to apply a list of values for a context to two terms at once. We generalise this to constructing a new continuous function from two continuous functions of the same input domain. We omit the proof of continuity for brevity.

```
to-×-DCPO : DCPO[ 𝒟 , ℰ ] → DCPO[ 𝒟 , ℱ ] → DCPO[ 𝒟 , ℰ ×ᵈᶜᵖᵒ ℱ ]
to-×-DCPO f g = h , c
  where
    h : ⟨ 𝒟 ⟩ → ⟨ ℰ ×ᵈᶜᵖᵒ ℱ ⟩
    h d = (underlying-function 𝒟 ℰ f d) , (underlying-function 𝒟 ℱ g d)
    c : is-continuous 𝒟 (ℰ ×ᵈᶜᵖᵒ ℱ) h
    c I α δ = u , s -- Proofs of u and s omitted for brevity
```

We then, for convenience, define this for continuous functions on DCPOs with least elements.

```
to-×-DCPO⊥ : DCPO⊥[ 𝒟 , ℰ ] → DCPO⊥[ 𝒟 , ℱ ] → DCPO⊥[ 𝒟 , ℰ ×ᵈᶜᵖᵒ⊥ ℱ ]
to-×-DCPO⊥ f g = to-×-DCPO ⟪ 𝒟 ⟫ ⟪ ℰ ⟫ ⟪ ℱ ⟫ f g
```

Since we have three terms which we want to evaluate under the same context, we can then use to-×-DCPO⊥ to construct a continuous function that can perform the denotational substitution of a list of values for the variables in the context to take on. This is just two applications of to-×-DCPO⊥. The first to form a continuous function that evaluates two terms under the same context, and the second application to form a continuous function which evaluates the original two terms and a third term under the same context. We construct this in such a manner that we end up with the result being a pair of the form $((a, b), c)$ as opposed to $(a, (b, c))$. This will prevent us from having to use a proof of associativity of the product of DCPOs, which will become apparent later.

```
(ifZero)-arguments : DCPO⊥[ 【 Γ 】 , 𝒟ᵈ ℕ ] → DCPO⊥[ 【 Γ 】 , 𝒟ᵈ ℕ ]
  → DCPO⊥[ 【 Γ 】 , 𝒟ᵈ ℕ ] → DCPO⊥[ 【 Γ 】 , (𝒟ᵈ ℕ ×ᵈᶜᵖᵒ⊥ 𝒟ᵈ ℕ) ×ᵈᶜᵖᵒ⊥ 𝒟ᵈ ℕ ]
```

```
⦇ifZero⦈-arguments a b c = to-×-DCPO⊥ 【 Γ 】 (𝒟ᵈℕ ×ᵈᶜᵖᵒ⊥ 𝒟ᵈℕ) 𝒟ᵈℕ f c
  where
    f : DCPO⊥[ 【 Γ 】 , 𝒟ᵈℕ ×ᵈᶜᵖᵒ⊥ 𝒟ᵈℕ ]
    f = to-×-DCPO⊥ 【 Γ 】 𝒟ᵈℕ 𝒟ᵈℕ a b
```

Now we have a way to evaluate all three of our terms under the same context, we next look at how we can make use of Tom de Jong's interpretation of IfZero from his combinatorial setting. The type of this interpretation is DCPO⊥[$\mathscr{D}^d\mathbb{N}$, $\mathscr{D}^d\mathbb{N} \Longrightarrow^{dcpo}_{\perp} \mathscr{D}^d\mathbb{N} \Longrightarrow^{dcpo}_{\perp} \mathscr{D}^d\mathbb{N}$], and is named ⦇ifZero⦈. We can not compose this directly with our function that evaluates our subterms of IfZero, as the input of ⦇ifZero⦈ does not match the type of the output of ⦇ifZero⦈-arguments. However, through the use of uncurry, we can construct a function from ⦇ifZero⦈ that takes the arguments as a product, as opposed to one at a time. We therefore construct ⦇ifZero⦈-uncurried as follows:

```
⦇ifZero⦈-uncurried' : DCPO⊥[ 𝒟ᵈℕ ×ᵈᶜᵖᵒ⊥ 𝒟ᵈℕ , 𝒟ᵈℕ ⟹ᵈᶜᵖᵒ⊥ 𝒟ᵈℕ ]
⦇ifZero⦈-uncurried' = uncurryᵈᶜᵖᵒ⊥ 𝒟ᵈℕ 𝒟ᵈℕ (𝒟ᵈℕ ⟹ᵈᶜᵖᵒ⊥ 𝒟ᵈℕ) ⦇ifZero⦈

⦇ifZero⦈-uncurried : DCPO⊥[ (𝒟ᵈℕ ×ᵈᶜᵖᵒ⊥ 𝒟ᵈℕ) ×ᵈᶜᵖᵒ⊥ 𝒟ᵈℕ , 𝒟ᵈℕ ]
⦇ifZero⦈-uncurried = uncurryᵈᶜᵖᵒ⊥ (𝒟ᵈℕ ×ᵈᶜᵖᵒ⊥ 𝒟ᵈℕ) 𝒟ᵈℕ 𝒟ᵈℕ ⦇ifZero⦈-uncurried'
```

Now we can see why the associativity mattered when we were constructing a pair, as since applying uncurry once formed a pair of the outermost arguments, then again formed a pair where the first projection is the pair of the outermost arguments, and the second projection is the innermost argument of ⦇ifZero⦈. We require the associativity of our argument construction to be the same. The reason is, for types A, B, and C, whilst $(A \times B) \times C$ is isomorphic to $A \times (B \times C)$, they are clearly not the same type, and to access an element of type C in the first product type would require a different chain of projections to the second product type.

We now can produce our interpretation of IfZero in our setting. We compose the function which evaluates each of the subterms of IfZero a b c under a context Γ with the uncurried ⦇ifZero⦈ construction. This provides us with a continuous function that takes a list of values to substitute for the variables, and produces the desired result of the IfZero evaluation.

```
⦇ifZero⦈Γ : DCPO⊥[ 【 Γ 】 , 𝒟ᵈℕ ] → DCPO⊥[ 【 Γ 】 , 𝒟ᵈℕ ] → DCPO⊥[ 【 Γ 】 , 𝒟ᵈℕ ]
          → DCPO⊥[ 【 Γ 】 , 𝒟ᵈℕ ]
⦇ifZero⦈Γ a b c = [ 【 Γ 】 , (𝒟ᵈℕ ×ᵈᶜᵖᵒ⊥ 𝒟ᵈℕ) ×ᵈᶜᵖᵒ⊥ 𝒟ᵈℕ , 𝒟ᵈℕ ]
                    ⦇ifZero⦈-uncurried ∘ᵈᶜᵖᵒ⊥ (⦇ifZero⦈-arguments Γ a b c)
```

Interpreting application

For an interpretation of $M \cdot N$, we apply the interpretation of M to the interpretation of N. We begin by constructing a continuous function called eval which performs this application given a continuous function from a DCPO $\mathscr{D}$ to $\mathscr{E}$, and an element of the DCPO $\mathscr{D}$, corresponding to the evaluation map mentioned by Streicher [Str06, Section 4].

```
eval : DCPO[ (𝒟 ⟹ᵈᶜᵖᵒ ℰ) ×ᵈᶜᵖᵒ 𝒟 , ℰ ]
eval = f , c
```

where

f : ⟨ (𝒟 ⟹^{dcpo} ℰ) ×^{dcpo} 𝒟 ⟩ → ⟨ ℰ ⟩

f (g , d) = underlying-function 𝒟 ℰ g d

We next provide a proof that f is continuous. Similarly to our uncurry proof, since our function input is a product, we can prove that f is continuous by showing it is continuous in each of its arguments. Since the proofs of continuous$_1$ and continuous$_2$ are relatively straightforward, we omit them for brevity.

c : is-continuous ((𝒟 ⟹^{dcpo} ℰ) ×^{dcpo} 𝒟) ℰ f

c = continuous-in-arguments→continuous (𝒟 ⟹^{dcpo} ℰ) 𝒟 ℰ f continuous$_1$ continuous$_2$

where -- Proofs of continuity in arguments omitted.

We can then construct eval⊥ for convenience, similar to as we have previously, to work on elements from DCPO⊥ rather than DCPO.

With the above construction, we can now form our interpretation of application as the composition of our to-×-DCPO⊥ construction which we use to evaluate two terms under a context simultaneously, and eval⊥ which performs the application as we would expect.

Interpretation function of terms

We begin with our interpretation function definition.

⟦_⟧$_e$: {n : ℕ} {Γ : Context n} {σ : type} (t : PCF Γ σ) → DCPO⊥[【 Γ 】 , ⟦ σ ⟧]

We first consider the interpretation for the PCF term Zero under the context Γ. This term has no free variables, and we always want it to correspond to the natural number zero in ℕ. So, we define a constant function where the output is always the total element zero of the lifted set 𝒟 ℕ. Since any constant function is continuous, then the function we have defined must be continuous.

⟦ Zero {_} {Γ} ⟧$_e$ = (λ _ → η zero) , const-functions-are-continuous ⟪ 【 Γ 】 ⟫ ⟪ ⟦ ι ⟧ ⟫ (η zero)

We next define the interpretation of Succ t, under a context Γ. Here we make use of 𝒟, which is responsible for taking a function between two types, and producing a function between the lifting of the types. It maps elements as the original function would, just with the extra mapping of ⊥ to ⊥. The lifted function is also shown to be continuous. We compose this function with the interpretation of the term t. We can view this first determining the value of t under a context Γ, and then taking the successor of the result.

⟦ Succ {_} {Γ} t ⟧$_e$ = [【 Γ 】 , ⟦ ι ⟧ , ⟦ ι ⟧]
(𝒟 succ , 𝒟-continuous ℕ-is-set ℕ-is-set succ) ∘^{dcpo}⊥ ⟦ t ⟧$_e$

The interpretation of Pred t under a context Γ is then similar to that of Succ t, except we use the function pred instead, which maps all natural numbers n to n − 1, apart from 0 which maps to 0.

⟦ Pred {_} {Γ} t ⟧$_e$ = [【 Γ 】 , ⟦ ι ⟧ , ⟦ ι ⟧]
(𝒟 pred , 𝒟-continuous ℕ-is-set ℕ-is-set pred) ∘^{dcpo}⊥ ⟦ t ⟧$_e$

For IfZero t t_1 t_2 under a context Γ, we interpret this using the DCPO with ⊥ we constructed previously. The order of the arguments is intentionally switched, as we define our arguments to

IfZero in the same order as Streicher [Str06, Section 2], however we use the interpretation from Tom de Jong, who defines his in a slightly different order [Jon19].

$$[\![\text{IfZero } \{\_\} \{\Gamma\} \ t \ t_1 \ t_2 \ ]\!]_e = (\!|\text{ifZero}|\!)\Gamma \ \Gamma \ [\![\ t_1 \ ]\!]_e \ [\![\ t_2 \ ]\!]_e \ [\![\ t \ ]\!]_e$$

Now we consider the interpretation of ƛ t. Since ƛ t has a type of $\sigma \Rightarrow \tau$, we need to produce a continuous function from $\llbracket \ \Gamma \ \rrbracket$ to the continuous function space $[\![\ \sigma \ ]\!] \Longrightarrow^{dcpo}_{\perp} [\![\ \tau \ ]\!]$. However, from the interpretation of t, we have a continuous function from $\llbracket \ \Gamma \ , \ \sigma \ \rrbracket$ to $[\![\ \tau \ ]\!]$. We can therefore just apply the currying operation to $[\![\ t \ ]\!]_e$.

$$[\![\text{ƛ } \{\_\} \{\Gamma\} \{\sigma\} \{\tau\} \ t \ ]\!]_e = \text{curry}^{dcpo}_{\perp} \ \llbracket \ \Gamma \ \rrbracket \ [\![\ \sigma \ ]\!] \ [\![\ \tau \ ]\!] \ [\![\ t \ ]\!]_e$$

As we previously explored, we represent the application of a term t to a term t_1 as first evaluating both terms under a list of values to substitute for their free variables, and then performing the evaluation as specified by eval⊥.

$$[\![\ \_\cdot\_ \ \{\_\} \{\Gamma\} \{\sigma\} \{\tau\} \ t \ t_1 \ ]\!]_e = [\ \llbracket \ \Gamma \ \rrbracket \ , \ (\ [\![\ \sigma \Rightarrow \tau \ ]\!] \times^{dcpo}_{\perp} [\![\ \sigma \ ]\!] \ ) \ , \ [\![\ \tau \ ]\!] \ ]$$
$$(\text{eval}_{\perp} \ [\![\ \sigma \ ]\!] \ [\![\ \tau \ ]\!]) \circ^{dcpo}_{\perp} (\text{to-}\times\text{-DCPO}_{\perp} \ \llbracket \ \Gamma \ \rrbracket \ [\![\ \sigma \Rightarrow \tau \ ]\!] \ [\![\ \sigma \ ]\!] \ [\![\ t \ ]\!]_e \ [\![\ t_1 \ ]\!]_e)$$

For a variable v x with type σ under the context Γ, we use our var-extract function we previously defined applied to x. This provides us with a continuous function from the interpretation of the context Γ to the interpretation of the type σ.

$$[\![\text{v } x \ ]\!]_e = \text{var-extract } x$$

When interpreting Fix t, we make use of Tom de Jong's interpretation of Fix in his setting. The function μ provides us an interpretation of the fixed-point combinator for a type σ. Therefore, this makes μ $[\![\ \sigma \ ]\!]$ a continuous function between the DCPOs $[\![\ \sigma \Rightarrow \sigma \ ]\!]$ and $[\![\ \sigma \ ]\!]$. Since we want to apply this fixed-point combinator to the interpretation of t, we compose μ $[\![\ \sigma \ ]\!]$ with the interpretation of t. This allows us to form a new continuous function which takes a list of values to substitute for the free variables in t, and applies the fixed-point combinator to return a result of the type $[\![\ \sigma \ ]\!]$, which is the fixed point of $[\![\ t \ ]\!]_e$ after substituting the list of values for free variables.

$$[\![\text{Fix } \{\_\} \{\Gamma\} \{\sigma\} \ t \ ]\!]_e = [\ \llbracket \ \Gamma \ \rrbracket \ , \ [\![\ \sigma \Rightarrow \sigma \ ]\!] \ , \ [\![\ \sigma \ ]\!] \ ] \ (\mu \ [\![\ \sigma \ ]\!]) \circ^{dcpo}_{\perp} [\![\ t \ ]\!]_e$$

7 Correctness

We next move to our first proof that relates the operational semantics and denotational semantics. We show that the operational semantics are correct with respect to the Scott model. This means that for any terms P and V, if $P \Downarrow V$, then it must follow that $[\![\ P \ ]\!]_e \equiv [\![\ V \ ]\!]_e$.

7.1 Substitution lemma

The substitution lemma is said to be proved by straightforward induction on the typing judgement [Str06, Lemma 5.1]. During our development, we experience some differences which Agda requires us to formalise, making the proof less straightforward. We first begin with a lemma, which is not formalised in the proof by Streicher. It only becomes evident that we require it when we try to prove the ƛ case whilst proving the substitution lemma by induction on the typing judgement. We will see later exactly where the following lemma becomes useful.

Lemma 7.1. *We assume a term* M : PCF $\Gamma\ \sigma$*, and a map* ρ *from variables in the context* Γ *to variables under a new context* Δ*. Given a set of values* e *to substitute for variables in* Γ*, and a set of values* d *to substitute for variables in* Δ*, if it is true that for all variables* x *in the context* Γ*,* extract $x\ e \equiv$ extract $(\rho\ x)\ d$*, then it follows that the result of applying* e *to the interpretation of* M *is equal to the result of applying* d *to the interpretation of* M *with variables renamed according to* ρ*.*

Proof. We begin by formalising the type which we are aiming to prove. We use $\mathrm{pr}_1\ [\![\ M\ ]\!]_e$ as opposed to underlying-function $\langle\!\langle\ 【\ \Gamma\ 】\ \rangle\!\rangle\ \langle\!\langle\ [\![\ \sigma\ ]\!]\ \rangle\!\rangle\ [\![\ M\ ]\!]_e$ to access the underlying function from interpretation of a term M. Whilst we prefer the latter as it is more readable, some lines in our proof would become excessively long otherwise.

```
rename-lemma : {n m : ℕ} {Γ : Context n} {Δ : Context m} {σ : type}
  → (M : PCF Γ σ) → (ρ : ∀ {A} → Γ ∋ A → Δ ∋ A)
  → (d : ⟨ ⟪ 【 Δ 】 ⟫ ⟩) → (e : ⟨ ⟪ 【 Γ 】 ⟫ ⟩)
  → (∀ {A} → (x : Γ ∋ A) → extract x e ≡ extract (ρ x) d)
  → pr₁ [[ M ]]e e ≡ pr₁ [[ rename ρ M ]]e d
```

We case split on the term M. The Zero case is proved by definition, since the term Zero does not change under renaming, and the interpretation of Zero is a constant function to η zero, any two arguments applied to the interpretation of Zero always give the same result.

```
rename-lemma Zero ρ d e eq = refl
```

The case of Succ M makes use of ap as we have before, which is applying the fact that for any two types $\boldsymbol{A}, \boldsymbol{B}$, given $f : \boldsymbol{A} \to \boldsymbol{B}$, and $x, y : \boldsymbol{A}$, $x \equiv y \to f\,x \equiv f\,y$. This is useful as our goal is to prove that $\mathscr{S}$ succ $(\mathrm{pr}_1\ [\![\ M\ ]\!]_e\ e) \equiv \mathscr{S}$ succ $(\mathrm{pr}_1\ [\![$ rename $\rho\ M\ ]\!]_e\ d)$. Since our inductive hypothesis gives $\mathrm{pr}_1\ [\![\ M\ ]\!]_e\ e \equiv \mathrm{pr}_1\ [\![$ rename $\rho\ M\ ]\!]_e\ d$, the proof of this case is trivial. Similarly for the Pred M case, apart from we use pred instead of succ.

```
rename-lemma (Succ M) ρ d e eq = ap (𝒮 succ) (rename-lemma M ρ d e eq)
rename-lemma (Pred M) ρ d e eq = ap (𝒮 pred) (rename-lemma M ρ d e eq)
```

For the IfZero $M\ M_1\ M_2$ case, we use ap_3 This follows the same concept as ap, apart from it works for functions of three arguments, and thus requires three proofs of equality - one for each argument. All three equalities are given from our inductive hypothesis.

```
rename-lemma (IfZero M M₁ M₂) ρ d e eq = ap₃ (λ x₁ x₂ x₃ → pr₁ ((ifZero)₁ x₂ x₃) x₁)
                                          (rename-lemma M ρ d e eq) (rename-lemma M₁ ρ d e eq) (rename-lemma M₂ ρ d e eq)
```

The next case we show is for ƛ M, which is the most interesting case.

```
rename-lemma (ƛ {n} {Γ} {σ} {τ} M) ρ d e eq = γ
  where
```

We want to show that $\mathrm{pr}_1\ [\![\ ƛ\ M\ ]\!]_e\ e \equiv \mathrm{pr}_1\ [\![$ rename ρ (ƛ M) $]\!]_e\ d$. Our first step is the application of the definition of rename, which states that rename ρ (ƛ M) simplifies to ƛ (rename (ext ρ) M), where ext, given a map from variables in one context to variables in another, provides a map between the

extended contexts as defined in [KSW20]. Applying this definition means we now want to show that $\mathrm{pr}_1\ [\![\ \lambda\!\!\lambda\ M\ ]\!]_e\ e \equiv \mathrm{pr}_1\ [\![\ \lambda\!\!\lambda\ (\mathrm{rename}\ (\mathrm{ext}\ \rho)\ M)\ ]\!]_e\ d$. Both sides of the equality are continuous functions from the domain $[\![\ \sigma \Rightarrow \tau\ ]\!]$.

Since our representation of a continuous function is a dependent pair, we begin by showing the first projections of both of the continuous functions are equal. The first projection of the pair is the underlying function itself. We say that two functions are equal if they always provide the same output as each other when given the same input. We use $\sim$ to represent this relation between two functions. Since the interpretation of lambda abstraction is the currying of the subterm, this means we want to show that $\mathrm{pr}_1\ [\![\ M\ ]\!]_e\ (e\ ,\ z) \equiv \mathrm{pr}_1\ [\![\ \mathrm{rename}\ (\mathrm{ext}\ \rho)\ M\ ]\!]_e\ (d\ ,\ z)$ for all z. However, we notice that this is just our inductive hypothesis.

```
ih : (λ z → pr₁ [[ M ]]ₑ (e , z)) ∼ (λ z → pr₁ [[ rename (ext ρ) M ]]ₑ (d , z))
ih z = rename-lemma M (ext ρ) (d , z) (e , z) g
  where
```

Our inductive hypothesis requires a new proof that extracting a value for a given variable x from $(e\ ,\ z)$ is the same as renaming x under the extended ρ and then extracting a value from $(d\ ,\ z)$. We prove this by induction on the lookup judgement. Our first case is Z, from which it follows by definition that both sides reduce to z, and thus our proof is just refl.

```
    g : ∀ {A} → (x : (Γ ' σ) ∋ A) → extract x (e , z) ≡ extract (ext ρ x) (d , z)
    g Z = refl
```

Our next case is $\mathrm{S}\ x$. We want to show $\mathrm{extract}\ (\mathrm{S}\ x)\ (e\ ,\ z) \equiv \mathrm{extract}\ (\mathrm{ext}\ \rho\ (\mathrm{S}\ x))\ (d\ ,\ z)$ which reduces to showing $\mathrm{extract}\ x\ e \equiv \mathrm{extract}\ (\rho\ x)\ d$ by definition, since $\mathrm{ext}\ \rho\ (\mathrm{S}\ x)$ reduces to $\mathrm{S}\ (\rho\ x)$, and then $\mathrm{extract}\ (\mathrm{S}\ (\rho\ x))\ (d\ ,\ z)$ reduces to $\mathrm{extract}\ (\rho\ x)\ d$, similarly $\mathrm{extract}\ (\mathrm{S}\ x)\ (e\ ,\ z)$ reduces to $\mathrm{extract}\ x\ e$. We notice this case is trivial, as it is given by applying x to our assumption eq.

```
    g (S x) = eq x
```

Since we have shown that the underlying functions produce equal output for the same input, we can provide a proof that these two functions are equal by nfunext, which applies our axiomatic approach to function extensionality. We also make use of Tom de Jong's proof that a given function being continuous is a proposition, which, as we have previously defined, means that any two proofs of continuity for a given function are equal. Since we have shown our underlying functions to be equal, and being continuous is a proposition, we can then make use of to-subtype-≡ to provide a proof that the two dependent pairs are equal.

```
γ : pr₁ [[ λ̂ M ]]ₑ e ≡ pr₁ [[ rename ρ (λ̂ M) ]]ₑ d
γ = to-subtype-≡ (being-continuous-is-a-prop ⟪ [[ σ ]] ⟫ ⟪ [[ τ ]] ⟫) (nfunext fe ih)
```

Application is then also trivial. We use ap_2, which again is similar to ap except for two argument functions. The two equalities of arguments we provide come from the induction hypothesis.

```
rename-lemma (M · M₁) ρ d e eq = ap₂ pr₁ (rename-lemma M ρ d e eq) (rename-lemma M₁ ρ d e eq)
```

The variable case comes straight from our assumption *eq*.

```
rename-lemma (v x) ρ d e eq = eq x
```

Then our final case for Fix M is shown by a simple application of ap, with the induction hypothesis.

```
rename-lemma (Fix {_} {_} {σ} M) ρ d e eq = ap (pr₁ (μ ⟦ σ ⟧)) (rename-lemma M ρ d e eq)
```
□

With this lemma formalised, we can now move on to prove the lemma we are actually interested in. We formalise it as Streicher does, and then we try to formalise it in Agda.

Lemma 7.2 (Substitution lemma)**.** *Let* $\Gamma = \sigma_1, ..., \sigma_n$ *be a context and* M : PCF $\Gamma\ \tau$ *a term. For all contexts* Δ *and terms* N_i : PCF $\Delta\ \sigma_i$ *with* $i = 1, ..., n$*, it holds that for all* d : $\langle \langle\langle\ \llbracket\ \Delta\ \rrbracket\ \rangle\rangle \rangle$*:*

$$\mathrm{pr}_1\ [\![\ \mathrm{subst}\ f\ M\ ]\!]_e\ d \equiv [\![\ M\ ]\!]_e(\mathrm{pr}_1\ [\![\ N_1\ ]\!]_e\ d,\ldots,\mathrm{pr}_1\ [\![\ N_n\ ]\!]_e\ d)$$

Proof. As stated, this proof should follow by induction on the typing judgement M : PCF $\Gamma\ \tau$. However, let's begin formalising this in Agda.

We first start with a helper function, which we use to construct the list of $[\![\ N_i\ ]\!]_e$ applied to d.

```
applied-all : {m n : ℕ} {Γ : Context n} → {Δ : Context m}
            → (∀ {A} → Γ ∋ A → PCF Δ A) → ⟨ ⟪ ⟦ Δ ⟧ ⟫ ⟩ → ⟨ ⟪ ⟦ Γ ⟧ ⟫ ⟩
applied-all {_} {_} {⟨⟩} f d = *
applied-all {_} {_} {Γ , x} f d = (applied-all (f ∘ S) d) , (pr₁ ⟦ f Z ⟧ₑ d)
```

Next, we begin to formalise the substitution lemma with our new construction. We use a substitution function f to represent the mapping of variables with de Bruijn index i to term N_i.

```
substitution-lemma' : {n m : ℕ}{Γ : Context n}{Δ : Context m}{σ : type}
                    → (M : PCF Γ σ) → (f : ∀ {A} → Γ ∋ A → PCF Δ A) → (d : ⟨ ⟪ ⟦ Δ ⟧ ⟫ ⟩)
                    → pr₁ ⟦ M ⟧ₑ (applied-all f d) ≡ pr₁ ⟦ subst f M ⟧ₑ d
```

This seems to match the lemma we are trying to prove, but whilst attempting to prove this for the lambda abstraction case, we begin to uncover some lemmas which we need to prove regarding applied-all, some of which may be tricky.

This is one of the cases where our proof is made simple by reformulating what we are trying to prove. We make an extra assumption e : $\langle \langle\langle\ \llbracket\ \Gamma\ \rrbracket\ \rangle\rangle \rangle$. We then add the constraint that for all lookup judgements x of the context Γ, we have $\mathrm{pr}_1\ [\![\ \mathrm{v}\ x\ ]\!]_e\ e \equiv \mathrm{pr}_1\ [\![\ f\ x\ ]\!]_e\ d$. We can view this as the proof requiring a list of values e, which take on the form similar to what we were trying to achieve with applied-all. The advantage this time is that we have access to the equality between extracting index x from e and the interpretation of the term $f\,x$ applied to d, which we did not have in our previous formation.

```
substitution-lemma : {n m : ℕ}{Γ : Context n}{Δ : Context m} {σ : type}
  → (M : PCF Γ σ) → (f : ∀ {A} → Γ ∋ A → PCF Δ A)
  → (e : ⟨ ⟪ ⟦ Γ ⟧ ⟫ ⟩) → (d : ⟨ ⟪ ⟦ Δ ⟧ ⟫ ⟩)
```

```
  → (∀ {A} → (x : Γ ∋ A) → pr₁ ⟦ v x ⟧ₑ e ≡ pr₁ ⟦ f x ⟧ₑ d )
  → pr₁ ⟦ M ⟧ₑ e ≡ pr₁ ⟦ subst f M ⟧ₑ d
```

Most of these cases are proved in a similar way to how we proved them for rename-lemma, so we shall only consider the interesting ones.

We first begin with the variable case. This is now trivial, since it's given by our assumption.

```
substitution-lemma (v x) f e d eq = eq x
```

Next is the lambda abstraction case. This is similar to the equivalent case for rename-lemma, although we need to construct a proof of $\mathrm{pr}_1\ [\![\ \mathrm{v}\ x\ ]\!]_e\ (e\ ,\ z) \equiv \mathrm{pr}_1\ [\![\ \mathrm{exts}\ f\ x\ ]\!]_e\ (d\ ,\ z)$ for all x.

```
substitution-lemma (ƛ {_} {Γ} {σ} {τ} M) f e d eq = γ
  where
    ih : (λ z → pr₁ ⟦ M ⟧ₑ (e , z)) ∼ (λ z → pr₁ ⟦ subst (exts f) M ⟧ₑ (d , z))
    ih z = substitution-lemma M (exts f) (e , z) (d , z) exts-eq
      where
```

We prove our new equality holds by induction on the lookup judgement. The base case is trivial, as it reduces to showing that $z \equiv z$.

```
        exts-eq : ∀ {A} → (x : (Γ ' σ) ∋ A) → pr₁ ⟦ v x ⟧ₑ (e , z) ≡ pr₁ ⟦ exts f x ⟧ₑ (d , z)
        exts-eq Z = refl
```

The next case is more interesting. By definition, $\mathrm{pr}_1\ [\![\ \mathrm{v}\ (\mathrm{S}\ \{\_\}\ \{\_\}\ \{\_\}\ \{\sigma\}\ x)\ ]\!]_e\ (e\ ,\ z)$ reduces to $\mathrm{pr}_1\ [\![\ \mathrm{v}\ x\ ]\!]_e\ e$. We have provided the implicit arguments to S, as Agda struggles to work one out in this case. We use the underscore for the ones which Agda can work out, and provide the parameter which it can not. We then show $\mathrm{pr}_1\ [\![\ \mathrm{v}\ x\ ]\!]_e\ e \equiv \mathrm{pr}_1\ [\![\ f\ x\ ]\!]_e\ d$ by applying our assumption eq to x. We next consider showing $\mathrm{pr}_1\ [\![\ f\ x\ ]\!]_e\ d \equiv \mathrm{pr}_1\ [\![\ \mathrm{exts}\ f\ (\mathrm{S}\ x)\ ]\!]_e\ (d\ ,\ z)$. However, since exts f (S x) reduces to rename S (f x), showing this equality is just an application of rename-lemma. Constructing the proof that rename-lemma requires is trivial, as for all typing judgements x_1, the equality we need to show reduces to extract x d ≡ extract x d, and we can provide the proof refl.

```
        exts-eq (S x) = pr₁ ⟦ v (S {_} {_} {_} {σ} x) ⟧ₑ (e , z) ≡⟨ eq x ⟩
                        pr₁ ⟦ f x ⟧ₑ d ≡⟨ rename-lemma (f x) S (d , z) d (λ x₁ → refl) ⟩
                        pr₁ ⟦ exts f (S x) ⟧ₑ (d , z) ∎
```

We show our representations of the continuous functions are equal in the same way we did in rename-lemma, using the proof that the underlying functions are equal by function extensionality.

```
  γ : pr₁ ⟦ ƛ M ⟧ₑ e ≡ pr₁ ⟦ subst f (ƛ M) ⟧ₑ d
  γ = to-subtype-≡ (being-continuous-is-a-prop ⟪ ⟦ σ ⟧ ⟫ ⟪ ⟦ τ ⟧ ⟫) (nfunext fe ih)
```

□

We then prove a corollary, which we will need when we prove correctness.

Corollary 7.3. *If* M : PCF (Γ ' σ) τ *and* N : PCF Γ σ, *then* $\mathrm{pr}_1\ [\![\ ƛ\ M \cdot N\ ]\!]_e \sim \mathrm{pr}_1\ [\![\ M\ [\ N\ ]\ ]\!]_e$

Proof. Our proof of β-equality follows easily from substitution-lemma. □

7.2 Proving correctness

Now we have enough to construct our proof of correctness. There is no explicit proof of this theorem in the paper we are following for our proofs, although there is a mention that it is easy to verify by induction on the structure of derivations that correctness of the Scott model with respect to the operational semantics holds [Str06, Section 7]. Since in our setting ⇓ is defined as the propositional truncation of ⇓', we first begin with a lemma for ⇓'. From this lemma, it becomes trivial to then show $[\![M]\!]_e$ and $[\![V]\!]_e$ are equal, and therefore correctness, since a function being continuous is a proposition.

Lemma 7.4. *For any* M : PCF $\Gamma\ \sigma$ *and* V : PCF $\Gamma\ \sigma$, *if* $M \Downarrow' V$, *then* $\mathrm{pr}_1\ [\![M]\!]_e \sim \mathrm{pr}_1\ [\![M]\!]_e$.

Proof. This proof is by induction on the derivations as suggested by Streicher. Some cases are simply applying induction hypothesis, and others make use of equalities already constructed by Tom de Jong. As such, for brevity, we omit most of this proof. We will consider the particular case for application, which makes use of β-equality we previously proved. To keep the lemma we are trying to prove tidy, we made the proof of this lemma for application a separate sublemma. Since the inductive hypothesis in the sublemma would not be strong enough, we take the inductive hypothesis from the main lemma as assumptions, which we can then give to the sublemma when it is applied. The assumptions are that the main lemma holds for $M \Downarrow' ƛ\ E$ and $E\ [\ T\ ] \Downarrow' N$.

```
correctness-· : {n : ℕ} {Γ : Context n} {σ τ : type}
              → (M : PCF Γ (σ ⇒ τ)) → (E : PCF (Γ , σ) τ) → (T : PCF Γ σ) → (N : PCF Γ τ)
              → pr₁ [![ M ]!]e ∼ pr₁ [![ ƛ E ]!]e → pr₁ [![ E [ T ] ]!]e ∼ pr₁ [![ N ]!]e
              → pr₁ [![ M · T ]!]e ∼ pr₁ [![ N ]!]e
```

We assume $d : \langle\langle\ 【\ \Gamma\ 】\ \rangle\rangle$, and show that the output of both functions are equal. We do this by equational reasoning. After applying the definition of $[\![M \cdot T]\!]_e$, our next step is the use of our assumption that $\mathrm{pr}_1\ [\![M]\!]_e \sim \mathrm{pr}_1\ [\![ƛ\ E]\!]_e$. If we apply d to this assumption, we can construct the proof that $\mathrm{pr}_1\ [\![M]\!]_e\ d \equiv \mathrm{pr}_1\ [\![ƛ\ E]\!]_e\ d$. We next make use of our proof of β-equality, to show that applying d to $\mathrm{pr}_1\ [\![ƛ\ E \cdot T]\!]_e$ is the same as applying d to $\mathrm{pr}_1\ [\![E\ [\ T\ ]]\!]_e$. Our final step is the application of the assumption that $\mathrm{pr}_1\ [\![E\ [\ T\ ]]\!]_e \sim \mathrm{pr}_1\ [\![N]\!]_e$, similar to how we did previously.

```
correctness-· M E T N c₁ c₂ d = pr₁ (pr₁ [![ M ]!]e d) (pr₁ [![ T ]!]e d)   ≡⟨ i ⟩
                                pr₁ (pr₁ [![ ƛ E ]!]e d) (pr₁ [![ T ]!]e d) ≡⟨ ii ⟩
                                pr₁ [![ E [ T ] ]!]e d ≡⟨ c₂ d ⟩
                                pr₁ [![ N ]!]e d        ∎
  where
    i = ap (λ - → pr₁ - (pr₁ [![ T ]!]e d)) (c₁ d)
    ii = β-equality E T d
```

As mentioned, since most cases are simple or just make use of equalities already constructed by Tom de Jong, we omit most of the proof. However, we will show how we apply the sublemma above. We see how the sublemma makes use of the inductive hypotheses from correctness'.

```
correctness' : {n : ℕ} {Γ : Context n} {σ : type}
             → (M N : PCF Γ σ) → M ⇓' N → pr₁ ⟦ M ⟧ₑ ∼ pr₁ ⟦ N ⟧ₑ
             -- Most cases omitted for brevity.
correctness' .(_ · _) N (·-step {_} {_} {_} {_} {M} {E} {T} {_} r r₁) =
                 correctness-· M E T N (correctness' M (ƛ E) r) (correctness' (E [ T ]) N r₁)   □
```

We next show correctness, which follows easily from correctness'.

Theorem 7.5 (Correctness)**.** *For any* $M : \mathrm{PCF}\ \Gamma\ \sigma$ *and* $V : \mathrm{PCF}\ \Gamma\ \sigma$*, if* $M \Downarrow V$ *then it follows that* $[\![M]\!]_e \equiv [\![V]\!]_e$*.*

Proof. Our proof involves showing the continuous functions $[\![M]\!]_e$ and $[\![N]\!]_e$ are equal, which we do in a similar fashion to as we did in substitution-lemma. We use a proof i that the two functions produce the same output for the same input, meaning they are equal by function extensionality.

```
correctness : {n : ℕ} {Γ : Context n} {σ : type} (M N : PCF Γ σ) → M ⇓ N → ⟦ M ⟧ₑ ≡ ⟦ N ⟧ₑ
correctness {_} {Γ} {σ} M N step
          = to-subtype-≡ (being-continuous-is-a-prop ⟪ 【 Γ 】 ⟫ ⟪ ⟦ σ ⟧ ⟫) (nfunext fe i)
  where
```

We next look at the proof i that the underlying functions of the interpretations are related by $\sim$. We assume $d : \langle \ll\ 【\ \Gamma\ 】\ \gg \rangle$, and show that $\mathrm{pr}_1\ [\![M]\!]_e\ d \equiv \mathrm{pr}_1\ [\![N]\!]_e\ d$. Since *step* is an element of type $M \Downarrow N$, which is the propositional truncation of the type $M \Downarrow' N$, we use ∥∥-rec to prove what we are trying to show. We are able to do this since the underlying type of $[\![\sigma]\!]$ is a set, therefore $\mathrm{pr}_1\ [\![M]\!]_e\ d \equiv \mathrm{pr}_1\ [\![N]\!]_e\ d$ is a proposition. We then provide a proof that from a given $x_1 : M \Downarrow' N$ we can produce a proof that $\mathrm{pr}_1\ [\![M]\!]_e\ d \equiv \mathrm{pr}_1\ [\![N]\!]_e\ d$ holds. Finally, we provide *step*, which shows there exists a proof such that $M \Downarrow' N$, and therefore conclude from the recursion principle that $\mathrm{pr}_1\ [\![M]\!]_e\ d \equiv \mathrm{pr}_1\ [\![N]\!]_e\ d$ holds.

```
    i : pr₁ ⟦ M ⟧ₑ ∼ pr₁ ⟦ N ⟧ₑ
    i d = ∥∥-rec (sethood ⟪ ⟦ σ ⟧ ⟫) (λ x₁ → correctness' M N x₁ d) step
```
□

8 Computational Adequacy

We next show that the denotational semantics are computationally adequate with respect to the operational semantics, following the proof by Streicher [Str06, Section 7]. We show this in the usual way, with the construction of a logical relation. We first define a syntactic preorder, the applicative approximation relation.

8.1 Applicative approximation

We can view the applicative approximation relation between two closed terms as containing the information that the two terms are in some sense equivalent in terms of their reductions. This becomes clearer in our definition.

Definition 8.1. We define the applicative approximation relation between closed terms M and N of type σ, $M \sqsubseteq_{\sigma} N$, by induction on σ as follows:

- For the base type ι, $M \sqsubseteq_{\iota} N$ iff $\forall(n : \mathbb{N}) \to M \Downarrow \underline{n} \to N \Downarrow \underline{n}$.
- For function types $\sigma \Rightarrow \tau$, $M \sqsubseteq_{\sigma\Rightarrow\tau} N$ iff $\forall(P : \mathrm{PCF}\ \langle\rangle\ \sigma) \to (M \cdot P) \sqsubseteq_{\tau} (N \cdot P)$.

In Agda, this translates almost directly, apart from the type is not a subscript of the relation symbol. Instead, it is an implicit parameter, as it can be inferred from the type of M and N.

```
_⊑̃_ : {σ : type} → PCF ⟨⟩ σ → PCF ⟨⟩ σ → 𝒰₀ ̇
_⊑̃_ {ι} M N = ∀ (n : ℕ) → M ⇓ ℕ-to-ι n → N ⇓ ℕ-to-ι n
_⊑̃_ {σ ⇒ τ} M N = ∀ (P : PCF ⟨⟩ σ) → (M · P) ⊑̃ (N · P)
```

During our attempts at following Streicher's proof of adequacy, we come across two lemmas regarding this relation which we depend upon. The first is the proof that for every PCF type σ and $M : \mathrm{PCF}\ \langle\rangle\ (\sigma \Rightarrow \sigma)$, it holds that $M \cdot \mathrm{Fix}\ M \sqsubseteq \mathrm{Fix}\ M$. Streicher proves this by inspection of the inductive definition of $\Downarrow$, so we first attempt to prove this by induction on σ. This makes sense, since $\sqsubseteq$ is defined by induction on the type, but we seem to become stuck in the inductive case.

Later in the proof of adequacy, we come across another proof on $\sqsubseteq$. Streicher notes that it generally holds that given $M : \mathrm{PCF}\ (\langle\rangle\ '\ \sigma)\ \tau$ and $N : \mathrm{PCF}\ \langle\rangle\ \sigma$, that $M\ [\ N\ ] \sqsubseteq ƛ\ M \cdot N$. However, this is just a footnote and no proof is given. Agda will not accept that a statement is true unless we prove it. Again, we try to prove this by induction on τ, but we become stuck.

One thing that seems common in both of our attempts at proving the lemmas is that the inductive hypothesis does not seem to be strong enough. However, we can notice some commonality in the lemmas we are trying to prove. From applying the big-step semantics, if $M \cdot \mathrm{Fix}\ M \Downarrow' V$, then it follows that $\mathrm{Fix}\ M \Downarrow' V$ for any V. Similarly, we can also say if $M\ [\ N\ ] \Downarrow' V$ then $ƛ\ M \cdot N \Downarrow' V$ for any V. So, it seems a more general form of what we are trying to prove is the following lemma:

Lemma 8.2. *Given $M : \mathrm{PCF}\ \langle\rangle\ \sigma$ and $N : \mathrm{PCF}\ \langle\rangle\ \sigma$, if for all $V : \mathrm{PCF}\ \langle\rangle\ \sigma$, $M \Downarrow' V$ implies that $N \Downarrow' V$, then $M \sqsubseteq N$.*

Proof. We prove this by induction on σ.

```
⊑̃-lemma : {σ : type} → (M N : PCF ⟨⟩ σ) → ((V : PCF ⟨⟩ σ) → M ⇓' V → N ⇓' V) → M ⊑̃ N
```

The base case is simple. From the definition of $\sqsubseteq$ for the base type, we want to show that for an $n : \mathbb{N}$, that if $M \Downarrow \mathbb{N}\text{-to-}\iota\ n$, then $N \Downarrow \mathbb{N}\text{-to-}\iota\ n$. We prove this using $\|\|$-functor as we have done previously. This requires that we show, given $M \Downarrow' \mathbb{N}\text{-to-}\iota\ n$, then $N \Downarrow' \mathbb{N}\text{-to-}\iota\ n$, but this follows

straight from applying our assumption f.

```
⊑̃-lemma {ι} M N f n step = ∥∥-functor γ step
  where
    γ : M ⇓' ℕ-to-ι n → N ⇓' ℕ-to-ι n
    γ step₁ = f (ℕ-to-ι n) step₁
```

Our inductive case follows straight from the inductive hypothesis, however we rely on a proof γ.

```
⊑̃-lemma {σ ⇒ τ} M N f P = ⊑̃-lemma (M · P) (N · P) γ
  where
```

Our proof γ assumes a V : PCF $\langle\rangle$ τ, and $step$: $(M \cdot P) \Downarrow' V$. Our goal is to show $(N \cdot P) \Downarrow' V$. There is only one possible case for $step$, and that's ·-step. So, we case split on $step$, and end up with this single case. From this, we get a proof $step_1$ that, for some E, $M \Downarrow'$ ƛ E, and a proof $step_2$ that E [P] $\Downarrow' V$. Now since we want to show $(N \cdot P) \Downarrow' V$, the only way we can show this is from ·-step which requires a proof that for some E, $N \Downarrow'$ ƛ E and a proof that E [P] $\Downarrow' V$.

We have that $N \Downarrow'$ ƛ E by applying our assumption f to $step_1$. Since we already have $step_2$, we can now apply ·-step to conclude this proof.

```
    γ : (V : PCF ⟨⟩ τ) → (M · P) ⇓' V → (N · P) ⇓' V
    γ V (·-step {_} {_} {_} {_} {_} {E} step₁ step₂) = ·-step N-step step₂
      where
        N-step : N ⇓' ƛ E
        N-step = f (ƛ E) step₁
```

□

We now can show the main properties we are interested in, which follow from the above lemma easily.

Corollary 8.3. *For each type* σ *and term* M : PCF $\langle\rangle$ $(\sigma \Rightarrow \sigma)$*, it holds that* $(M \cdot \text{Fix } M) \sqsubseteq$̃ $(\text{Fix } M)$.

Proof. Our proof follows immediately from ⊑̃-lemma. We provide p, which is a proof of the property we require the two terms of our applicative approximation to have in order to apply our lemma. To show p, we assume V : PCF $\langle\rangle$ σ, and given $(M \cdot \text{Fix } M) \Downarrow' V$, we show Fix $M \Downarrow' V$ by the big-step rule Fix-step.

```
fix-⊑̃ : {σ : type} → {M : PCF ⟨⟩ (σ ⇒ σ)} → (M · (Fix M)) ⊑̃ (Fix M)
fix-⊑̃ {σ} {M} = ⊑̃-lemma (M · Fix M) (Fix M) p
  where
    p : (V : PCF ⟨⟩ σ) → (M · Fix M) ⇓' V → Fix M ⇓' V
    p _ step = Fix-step step
```

□

Corollary 8.4. *Given* M : PCF $(\langle\rangle$ ' $\sigma)$ τ *and* N : PCF $\langle\rangle$ σ*, it follows that* $(M$ [N]$) \sqsubseteq$̃ $(ƛ\ M \cdot N)$.

Proof. Our proof again follows immediately from ⊑̃-lemma. We provide p, which is a proof of the

property we require the two terms of our applicative approximation to have in order to apply our lemma. To show p, we assume V : PCF ⟨⟩ τ, and given (M [N]) ⇓' V, we show (ƛ M · N) ⇓' V by the big-step rule ·-step.

```
β-⊑̃ : {σ τ : type} {M : PCF (⟨⟩ ' σ) τ} {N : PCF ⟨⟩ σ} → (M [ N ]) ⊑̃ (ƛ M · N)
β-⊑̃ {_} {τ} {M} {N} = ⊑̃-lemma (M [ N ]) (ƛ M · N) p
  where
    p : (V : PCF ⟨⟩ τ) → (M [ N ]) ⇓' V → (ƛ M · N) ⇓' V
    p _ step = ·-step ƛ-id step
```

□

8.2 Showing adequacy

We begin by defining the adequacy relation. For the base case, we take the product with the singleton type 𝟙, which is a slight hack to ensure our resulting type is in the universe $\mathcal{U}_1$, matching the σ ⇒ σ_1 case. We could have formulated this differently, but in order to stay as close to Streicher's proof as possible and avoid any difficulties, we opt for this solution.

```
adequacy-relation : (σ : type) (d : ⟨ ⟪ ⟦ σ ⟧ ⟫ ⟩) (M : PCF ⟨⟩ σ) → 𝒰₁̇
adequacy-relation ι d M = 𝟙 × ∀ (p : is-defined d) → M ⇓ ℕ-to-ι (value d p)
adequacy-relation (σ ⇒ σ₁) d M = ∀ (e : ⟨ ⟪ ⟦ σ ⟧ ⟫ ⟩) (N : PCF ⟨⟩ σ)
                                 → adequacy-relation σ e N → adequacy-relation σ₁ (pr₁ d e) (M · N)
```

We first state some lemmas, which we rely on in our proof. The proofs are shown by Streicher [Str06, Section 7], and they translate into Agda without much difficulty. As such, we omit the proofs for brevity.

Lemma 8.5. *Given a context Γ, M : PCF Γ σ, d d' : ⟨ ⟪ ⟦ σ ⟧ ⟫ ⟩, if d ⊑⟨ ⟪ ⟦ σ ⟧ ⟫ ⟩ d' and* adequacy-relation σ *d M, then* adequacy-relation σ *d' M.*

Lemma 8.6. *Given a directed family* α : I → ⟨ ⟪ ⟦ σ ⟧ ⟫ ⟩, *a proof* δ *that* α *is directed, and a term M* : PCF ⟨⟩ σ, *if for all i : I it holds that* adequacy-relation σ (α i) M, *then it follows that* adequacy-relation σ (⨆ ⟪ ⟦ σ ⟧ ⟫ δ) M.

Lemma 8.7. *For every M* : PCF ⟨⟩ σ, adequacy-relation σ (the-least ⟪ ⟦ σ ⟧ ⟫) M.

Lemma 8.8. *For M* : PCF ⟨⟩ σ, *M'* : PCF ⟨⟩ σ, *d* : ⟨ ⟪ ⟦ σ ⟧ ⟫ ⟩, *if* adequacy-relation σ *d M and M ⊑̃ M', then* adequacy-relation σ *d M'.*

Lemma 8.9. *For a given M* : PCF ⟨⟩ (σ ⇒ σ), *and f* : ⟨ ⟪ ⟦ σ ⇒ σ ⟧ ⟫ ⟩, *if* adequacy-relation (σ ⇒ σ) *f M, then* adequacy-relation σ (pr₁ (μ ⟦ σ ⟧) *f*) (Fix M).

We can now begin to look at the main lemma we need for showing computational adequacy.

Lemma 8.10. *Given M* : PCF Γ τ, *a list of values d* : ⟨ ⟪ 【 Γ 】 ⟫ ⟩, *and a mapping f from variables in Γ to closed terms, if for each variable i in Γ it is true that* extract *i d and f i are related*

by the adequacy relation, then $\mathrm{pr}_1\ [\![\ M\ ]\!]_e\ d$ *and* subst $f\ M$ *are related by the adequacy relation.*

Proof. As in Streicher's proof, we will only show the most interesting cases. We begin by formulating the type of what we are trying to prove. This closely resembles the statement of the lemma.

```
main-lemma : {n : ℕ} {Γ : Context n} {τ : type} (M : PCF Γ τ) (d : ⟨ ⟪ ⟦ Γ ⟧ ⟫ ⟩)
           → (f : ∀ {A} → (i : Γ ∋ A) → PCF ⟨⟩ A)
           → (g : ∀ {A} → (i : Γ ∋ A) → adequacy-relation A (extract i d) (f i))
           → adequacy-relation τ (pr₁ [[ M ]]e d) (subst f M)
```

We show this by induction on M. However, we will only explore the lambda abstraction, variable, and fixed-point combinator cases, as the rest, as Streicher suggests, go through without pain.

Our variable case follows immediately from our assumption g.

```
main-lemma (v x) d f g = g x
```

Next, we consider the Fix case. From our inductive hypothesis, we have that $\mathrm{pr}_1\ [\![\ M\ ]\!]_e\ d$ and subst $f\ M$ are related by the adequacy relation. Therefore, we can apply Lemma 8.9 to conclude our proof.

```
main-lemma {_} {_} {σ} (Fix M) d f g = lemma8-9 (subst f M) (pr₁ [[ M ]]e d) ih
  where
    ih : adequacy-relation (σ ⇒ σ) (pr₁ [[ M ]]e d) (subst f M)
    ih = main-lemma M d f g
```

We now consider the lambda abstraction case. Since the type of $ƛ\ M$ is $\sigma \Rightarrow \tau$, by the definition of adequacy-relation we further assume $d_1 : \langle\ \langle\!\langle\ [\![\ \sigma\ ]\!]_e\ \rangle\!\rangle\ \rangle$, $M_1 : \mathrm{PCF}\ \langle\rangle\ \sigma$, and *rel* : adequacy-relation $\sigma\ d_1\ M_1$ on top of our previous assumptions. Our goal is to show that $\mathrm{pr}_1\ (\mathrm{pr}_1\ [\![\ ƛ\ M\ ]\!]_e\ d)\ d_1$ and subst $f\ (ƛ\ M) \cdot M_1$ are related by the adequacy relation, which we give the name γ.

```
main-lemma (ƛ {_} {Γ} {σ} {τ} M) d f g d₁ M₁ rel = γ
  where
```

Our first step in the proof is to apply our inductive hypothesis. Since M is under the context Γ ' σ, we need to supply a mapping from each variable in this extended context to a closed term. We do this via mapping Z to M_1, and S x to $f\ x$, which is the function of extend-with $M_1\ f$. We also need to provide a proof that each of these mappings of variables is related to their respective values under the adequacy relation, which we call ext-g.

```
    ih : adequacy-relation τ (pr₁ (pr₁ [[ ƛ M ]]e d) d₁) (subst (extend-with M₁ f) M)
    ih = main-lemma M (d , d₁) (extend-with M₁ f) ext-g
      where
```

To construct ext-g, we perform induction on the lookup judgement. The base case is trivial, it is provided by our assumption *rel*. The inductive case is also trivial, as these are just the variables which were in the context Γ, and our assumption g states the property holds for each of these.

```
ext-g : ∀ {A} → (x : (Γ ' σ) ∋ A)
              → adequacy-relation A (extract x (d , d₁)) (extend-with M₁ f x)
ext-g Z = rel
ext-g (S x) = g x
```

We next show i, an equality subst (extend-with M_1 f) $M \equiv$ (subst (exts f) M) [M_1] which is mentioned simply without proof by Streicher [Str06, Lemma 7.4]. This is another example of the differences when proving properties with a proof assistant. Whilst mentioned trivially on paper, constructing subst-ext in Agda relied on many lemmas. The resulting lines of code were in the hundreds, as we were forced to convince Agda that the equality is true.

```
i : subst (extend-with M₁ f) M ≡ (subst (exts f) M) [ M₁ ]
i = subst-ext M M₁ f
```

From β-⊑̃, we have that (subst (exts f) M) [M_1] ⊑̃ ƛ (subst (exts f) M) · M_1. We are then able to prove subst (extend-with M_1 f) M ⊑̃ ƛ (subst (exts f) M) · M_1 by transporting the equality i with the property shown by β-⊑̃. By definition, subst f (ƛ M) is the same as ƛ (subst (exts f) M), so we have also shown subst (extend-with M_1 f) M ⊑̃ ((subst f (ƛ M)) · M_1), which is our proof ii.

```
ii : subst (extend-with M₁ f) M ⊑̃ ((subst f (ƛ M)) · M₁)
ii = back-transport (λ - → - ⊑̃ ((subst f (ƛ M)) · M₁)) i β-⊑̃
```

Finally, we apply Lemma 8.8 to conclude our proof.

```
γ : adequacy-relation τ (pr₁ (pr₁ ⟦ ƛ M ⟧ₑ d) d₁) ((subst f (ƛ M)) · M₁)
γ = adequacy-step (subst (extend-with M₁ f) M) ((subst f (ƛ M)) · M₁) ii (pr₁ ⟦ M ⟧ₑ (d , d₁)) ih □
```

Adequacy now follows from main-lemma simply.

Theorem 8.11 (Adequacy)**.** *If the interpretation of a term M : PCF ⟨⟩ ι is equal to η n, for some natural number n, then there exists a reduction such that $M \Downarrow$ ℕ-to-ι n.*

Proof. We develop a proof γ that the adequacy relation holds between η n and M. From this, it becomes trivial to show $M \Downarrow$ ℕ-to-ι n, as a proof that η n is defined is the unique element * of the type 𝟙, as the total elements of the lifted set are always defined.

```
adequacy : (M : PCF ⟨⟩ ι) (n : ℕ) → pr₁ ⟦ M ⟧ₑ * ≡ η n → M ⇓ ℕ-to-ι n
adequacy M n eq = pr₂ γ *
  where
```

We first apply main-lemma. We provide the identity substitution ids, which is the unique substitution we can provide for a term with no free variables. A proof that each term in the empty context is related to its extraction from the empty context is trivial, as there is no case for ⟨⟩ ∋ A. In Agda, () represents the absurd case. We conclude i, that the adequacy relation holds between the interpretation of M and the identity substitution applied to M.

```
i : adequacy-relation ι (pr₁ ⟦ M ⟧ₑ *) (subst ids M)
i = main-lemma M * ids f
  where
    f : ∀ {A} → (x : ⟨⟩ ∋ A) → adequacy-relation A (extract x *) (v x)
    f ()
```

We next apply the proof that the identity substitution is equal to the original term, specifically that subst ids $M \equiv M$. Like we have previously, we transport this equality to achieve a proof ii that the adequacy relation must hold between the interpretation of M and the term M.

```
ii : adequacy-relation ι (pr₁ ⟦ M ⟧ₑ *) M
ii = transport (adequacy-relation ι (pr₁ ⟦ M ⟧ₑ *)) (sub-id M) i
```

Finally, we can then prove the adequacy relation holds between $\eta\ n$ and M. We use our assumption eq that $\mathrm{pr}_1\ [\![M]\!]_e\ * \equiv \eta\ n$.

```
γ : adequacy-relation ι (η n) M
γ = transport (λ - → adequacy-relation ι - M) eq ii
```

□

9 Evaluation

During this investigation, we have had to learn many new skills and concepts. The obvious skill we have had to develop throughout this project is our proficiency with developing our proofs in Agda. In the beginning, our Agda knowledge was minimal. There are many Agda features we have had to explore during this project. We experienced times where we used anonymous modules to make assumptions across many proofs without having to explicitly list the assumptions in each individual proof. We also had to become familiar with using Agda's universes, which we renamed to terminology closer to that used in type theory like in Martín Escardó's developments. We gained a deeper understanding of how we can transfer proofs from paper to Agda using these concepts, and as a result our Agda skills vastly improved towards the end of our investigations, learning where Agda differs from paper proofs. A particular example of where we noticed this difference was when constructing the substitution lemma in Section 7.1, where we saw that it was not as straightforward to translate to Agda in our setting as it sounds by Streicher.

The learning curve. When we first began this project, the proofs were suboptimal in the sense that many could have been shortened considerably, and many steps were unnecessary. As our understanding of Agda and the environment we were working in grew, such as developing a deeper understanding of domain theory and the current developments by Tom de Jong, if we were to come back to the proofs, it would be possible to write them in a more succinct and elegant manner. A good example of a proof which was rewritten at the end of the project was the proof that eval was continuous in its first argument, which is shown in Appendix C. Our goal was to show that given a directed family $\alpha : I \to \langle\ \mathscr{D} \Longrightarrow^{dcpo} \mathscr{E}\ \rangle$, and $d : \langle\ \mathscr{D}\ \rangle$, then f ($\coprod$ ($\mathscr{D} \Longrightarrow^{dcpo} \mathscr{E}$) δ , d) is the least

upper bound of the pointwise family formed by α and d. Our original proof spanned many lines. We first showed that f ($\bigsqcup$ ($\mathscr{D} \Longrightarrow^{dcpo} \mathscr{E}$) δ , d) is the upper bound, and then that it was the least of the upper bounds, and as such was the least upper bound. However, at the end of the project, when reviewing this proof we notice that this was unnecessary. With our greater understanding, we learn that the least upper bound of α applied to an element d, is by definition the least upper bound of the pointwise family formed by α and d. Therefore, we rewrote this proof simply to be a single line by knowledge of this fact. If time allowed, we would go through all proofs and attempt to make them as readable and elegant as possible.

As mentioned, we also have had to develop our understanding of domain theory. We began from basic knowledge and understanding, where we learned the definitions of a directed family, an upper bound, and more. We then had to build on this to learn more interesting properties, such as proving that the products of two DCPOs form a DCPO. After this, we developed constructions involving continuous functions and our product of DCPOs, along with Tom de Jong's existing developments, to form the Scott model of PCF. This learning was the key focus for someone who comes from a software engineering background, as it required mathematical concepts which we were not familiar with at the beginning. Relating to our previous point, as our understanding in this area grew, as did our proficiency in proving properties surrounding domain theory. We compare this to defining the PCF types and terms, and the operational semantics of PCF. These concepts are familiar to someone with our background. We can easily relate them to what one may see when programming in a language such as Java, and how the operational semantics relates to compilers and interpreters. The concepts we learn and develop in domain theory were perhaps more difficult to relate to, and hence required the most time and effort to form an understanding.

As a result of this lack of initial understanding, our first contribution to the DCPO developments took a considerable amount of time longer than the later proofs. This contribution was that the product of two DCPOs also forms a DCPO under the component-wise ordering. This proof, whilst trivial to somebody with experience in domain theory, was difficult for us due to our unfamiliarity. However, once we overcome this initial learning curve, we possessed enough knowledge to complete proofs surrounding DCPOs with considerably more pace and efficiency.

Challenges in formalisation. Another area that was difficult at times, whilst as the project has progressed we have become better at it, was translating the paper proofs we were following into Agda. There were a couple of times throughout the project where we experienced theorems or lemmas that were stated to be "straightforward induction", with no detailed proof given. For these, we sometimes had to develop our own proofs. A particularly good example of this was the proofs regarding the applicative approximation relation when we were proving computational adequacy. The first was that $M \cdot \mathrm{Fix}\ M \sqsubseteq \mathrm{Fix}\ M$. Although a proof was given for this, it did not directly translate into Agda. The other proof was a footnote in Streicher's paper which states it generally holds that $M\ [\ N\ ] \sqsubseteq \lambda\ M \cdot N$. Both of these proofs we tried to prove by induction on types,

however both became stuck. It seemed our inductive hypothesis was not strong enough. After some assistance and time, we managed to work out that we can prove a more general lemma which both of these proofs follow trivially. This shows that some proofs which may be left out of papers require explicit formalisation in Agda, increasing confidence in the correctness of our proofs.

There was also the proof subst (extend-with M_1 f) $M \equiv$ (subst (exts f) M) [M_1] required in Lemma 8.10. This property was mentioned as being true in [Str06, Lemma 7.4], but without proof. We first turned to [KSW20], as there are similar proofs surrounding substitution, but for the untyped lambda calculus. Whilst these proofs gave some direction as to how we should attempt to solve the proof, they did not translate directly for PCF. We instead were required to prove three fusion lemmas for the rename and subst operations. This "simple" proof became hundreds of lines of code, and is another example of how, when formalising proofs in Agda, we are forced to show every detail.

During this project, we have had to learn a lot, and many times there were satisfying moments when we finally managed to complete proofs. However, the one that particularly stands out was the completion of the Scott model. After spending a considerable amount of time and effort learning domain theory, it gave great satisfaction to finally see the knowledge come together to provide us with the denotational semantics of PCF. The statistics of the volume of work are provided in Appendix B. It also required some creativity at times to reuse Tom de Jong's developments, as his interpretations for IfZero and Fix were not under contexts. This was a milestone in our developments in that we were sure we had learned enough domain theory to be able to construct the model and now could move on to showing computational adequacy.

10 Conclusion

Our investigations have shown the differences at times between paper proofs and proofs in a proof assistant. We have seen times where we have had to focus on formalising proofs which are not always detailed in texts, which our proof assistant required us to formalise.

We began by formalising the PCF types and terms in Section 3. We decided upon our choice of variables, considering the difference between de Bruijn indices and named variables in Section 3.2. We next moved on to define substitution and the operational semantics in Section 4. Following this, we looked at PCF through a more mathematical lens. In Section 5, we built upon Tom de Jong's existing formalisations of domain theory to provide us with a framework to define the Scott model of PCF in Section 6. We then showed that the operational semantics are correct with respect to the Scott model in Section 7, and finally that the Scott model is computationally adequate with respect to the operational semantics in Section 8.

The difference between paper proofs and using a proof assistant was particularly apparent in Section 7.1 when formalising the substitution lemma and Section 8.1 when formalising lemmas surrounding the applicative approximation relation. There were either no proofs for these lemmas, or

the proof did not directly translate into Agda. As a result, we had to spend extra time developing these proofs, whereas if we weren't using a proof assistant we would probably have relied upon these statements as true. Whilst we managed to prove them, it is not impossible that we could come across situations similar to as Gouëzel did [GS19] where we come across a statement which does not hold. As a result, we have highlighted that a proof assistant requires explicit proofs of all properties and steps involved, whereas a paper proof may be more likely to leave out proofs or rely on assumptions which could turn out to be false.

References

[Agd] The Agda Team. "Universe Levels - Agda 2.6.0 documentation". URL: https://agda.readthedocs.io/en/v2.6.0/language/universe-levels.html (cit. on p. 4).

[Bru72] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0 (cit. on p. 7).

[EK17] Martín H. Escardó and Cory M. Knapp. "Partial Elements and Recursion via Dominances in Univalent Type Theory". In: *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)*. Ed. by Valentin Goranko and Mads Dam. Vol. 82. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 21:1–21:16. ISBN: 978-3-95977-045-3. DOI: 10.4230/LIPIcs.CSL.2017.21 (cit. on pp. 2, 13, 25).

[Esc+] Martín Hötzel Escardó et al. "TypeTopology — Various new theorems in constructive univalent mathematics written in Agda". URL: https://www.cs.bham.ac.uk/~mhe/agda-new/ (cit. on p. 2).

[Esc19] Martín Hötzel Escardó. *Introduction to Univalent Foundations of Mathematics with Agda*. 2019. arXiv: 1911.00580 [cs.LO] (cit. on p. 5).

[GS19] Sébastien Gouëzel and Vladimir Shchur. "A corrected quantitative version of the Morse lemma". In: *Journal of Functional Analysis* 277.4 (2019), pp. 1258–1268. ISSN: 0022-1236. DOI: https://doi.org/10.1016/j.jfa.2019.02.021 (cit. on pp. 2, 45).

[How69] William A. Howard. "The Formulae-as-types notion of construction". In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. 1980 (originally circulated 1969), pp. 479–490. ISBN: 978-0-12-349050-6 (cit. on p. 2).

[Jon19] Tom de Jong. *The Scott model of PCF in univalent type theory*. 2019. arXiv: 1904.09810 [math.LO] (cit. on pp. 2, 12, 13, 24, 30).

[KSW20] Wen Kokke, Jeremy G. Siek, and Philip Wadler. "Programming language foundations in Agda". In: *Science of Computer Programming* 194 (2020), p. 102440. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2020.102440 (cit. on pp. 8, 9, 32, 44).

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091 (cit. on p. 8).

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

[Plo83] Gordon Plotkin. "Domains". Lecture notes on domain theory, known as the *Pisa Notes*. 1983. URL: https://homepages.inf.ed.ac.uk/gdp/publications/Domains_a4.ps (cit. on p. 14).

[Sco93] Dana S. Scott. "A type-theoretical alternative to ISWIM, CUCH, OWHY". In: *Theoretical Computer Science* 121.1 (1993), pp. 411–440. DOI: 10.1016/0304-3975(93)90095-B (cit. on pp. 2, 24).

[Str06] Thomas Streicher. *Domain-Theoretic Foundations of Functional Programming*. USA: World Scientific Publishing Co., Inc., 2006. ISBN: 9812701427 (cit. on pp. 2, 9, 14, 28, 30, 35, 36, 39, 41, 44).

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013 (cit. on p. 2).

[Wei95] Klaus Weihrauch. *A Simple Introduction to Computable Analysis*. Tech. rep. 171. Informatik Berichte. FernUniversität Hagen, 1995 (cit. on p. 14).

Appendices

Appendix A Type Checking

The source code is verified to type check with Agda 2.6.0.1. It can not be guaranteed that the source code for this project will not encounter errors, such as unknown flags or syntax errors, with other versions. The full source code for this project can be found at https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2019/bxh538. The source files can be found in the `src` folder, with the `.lagda` file extension.

Upon retrieval of the source code, individual files can be type checked by running:

```
$ agda <path-to-file>
```

This will also ensure any imported files are type checked.

However, when exploring Agda code, we tend to use Emacs. There is the Agda mode for Emacs which adds functionality to Emacs for interacting with Agda files, such as loading an opened Agda file. Instructions for setting this up can be found at the Agda documentation.

After opening an Agda file in Emacs, it can then be type checked with `Ctrl-c Ctrl-l`. A more detailed list of the Agda mode for Emacs can be found at https://agda.readthedocs.io/en/v2.6.0.1/tools/emacs-mode.html.

Appendix B Codebase Statistics

We make use of Tom de Jong's foundations in domain theory, as well as formalisations of homotopy type theory by Martín Escardó et al. The existing codebase is currently hosted at https://www.cs.bham.ac.uk/~mhe/agda-new/. The contributors are found at the beginning of each file as a comment.

We have added the following to the existing codebase:

- 4338 lines, approx. 11% increase.
- 375631 characters, approx. 14% increase.
- 18 new files, approx. 12% increase.

Appendix C Shortening of Proofs

Below is an example of the simplification of the proof of eval, where we simplify the proof of continuity in the first argument to a single line.

First, our original eval proof:

```
eval-original : DCPO[ (𝒟 ⟹^dcpo ℰ) ×^dcpo 𝒟 , ℰ ]
eval-original = f , c
  where
    f : ⟨ (𝒟 ⟹^dcpo ℰ) ×^dcpo 𝒟 ⟩ → ⟨ ℰ ⟩
    f (g , d) = underlying-function 𝒟 ℰ g d
    f-is-monotone : is-monotone ((𝒟 ⟹^dcpo ℰ) ×^dcpo 𝒟) ℰ f
    f-is-monotone (g₁ , d₁) (g₂ , d₂) (g₁⊑g₂ , d₁⊑d₂) =
                    f (g₁ , d₁) ⊑⟨ ℰ ⟩[ continuous-functions-are-monotone 𝒟 ℰ g₁ d₁ d₂ d₁⊑d₂ ]
                    f (g₁ , d₂) ⊑⟨ ℰ ⟩[ g₁⊑g₂ d₂ ]
                    f (g₂ , d₂) ■⟨ ℰ ⟩

    c : is-continuous ((𝒟 ⟹^dcpo ℰ) ×^dcpo 𝒟) ℰ f
    c = continuous-in-arguments→continuous (𝒟 ⟹^dcpo ℰ) 𝒟 ℰ f continuous₁ continuous₂
      where
        continuous₁ : (e : ⟨ 𝒟 ⟩) → is-continuous (𝒟 ⟹^dcpo ℰ) ℰ (λ d → f (d , e))
        continuous₁ d I α δ = u , v
          where
            u : is-upperbound (underlying-order ℰ) (f (⨆ (𝒟 ⟹^dcpo ℰ) {I} {α} δ , d))
                                                   (λ z → f (α z , d))
            u i = f-is-monotone (α i , d) (⨆ (𝒟 ⟹^dcpo ℰ) {I} {α} δ , d)
                                (⨆-is-upperbound (𝒟 ⟹^dcpo ℰ) {I} {α} δ i , reflexivity 𝒟 d)
            v : (u₁ : ⟨ ℰ ⟩) →
                  ((i : I) → f (α i , d) ⊑⟨ ℰ ⟩ u₁) →
                  f (⨆ (𝒟 ⟹^dcpo ℰ) {I} {α} δ , d) ⊑⟨ ℰ ⟩ u₁
            v u₁ p = ⨆-is-lowerbound-of-upperbounds ℰ isdir₁ u₁ p
              where
                isdir₁ : is-Directed ℰ (pointwise-family 𝒟 ℰ α d)
                isdir₁ = pointwise-family-is-directed 𝒟 ℰ α δ d

        continuous₂ : (d : ⟨ 𝒟 ⟹^dcpo ℰ ⟩) → is-continuous 𝒟 ℰ (λ e → f (d , e))
        continuous₂ g I α δ = u , v
          where
            u : is-upperbound (underlying-order ℰ) (f (g , ⨆ 𝒟 δ)) (λ z → f (g , α z))
            u i = f-is-monotone (g , α i) (g , ⨆ 𝒟 δ)
                                ((reflexivity (𝒟 ⟹^dcpo ℰ) g) , (⨆-is-upperbound 𝒟 δ i))
            v : (u₁ : ⟨ ℰ ⟩) →
                  ((i : I) → f (g , α i) ⊑⟨ ℰ ⟩ u₁) → f (g , ⨆ 𝒟 δ) ⊑⟨ ℰ ⟩ u₁
            v u₁ p = transport (λ - → - ⊑⟨ ℰ ⟩ u₁) (e₁ ⁻¹) p₁
```

```
      where
        e₁ : f (g , ∐ 𝒟 δ) ≡ ∐ ℰ (image-is-directed 𝒟 ℰ g δ)
        e₁ = continuous-function-∐-≡ 𝒟 ℰ g δ
        p₁ : (∐ ℰ (image-is-directed 𝒟 ℰ g δ)) ⊑⟨ ℰ ⟩ u₁
        p₁ = ∐-is-lowerbound-of-upperbounds ℰ (image-is-directed 𝒟 ℰ g δ) u₁ p
```

Next, we present the shorter proof:

```
eval : DCPO[ (𝒟 ⟹^dcpo ℰ) ×^dcpo 𝒟 , ℰ ]
eval = f , c
  where
    f : ⟨ (𝒟 ⟹^dcpo ℰ) ×^dcpo 𝒟 ⟩ → ⟨ ℰ ⟩
    f (g , d) = underlying-function 𝒟 ℰ g d
    f-is-monotone : is-monotone ((𝒟 ⟹^dcpo ℰ) ×^dcpo 𝒟) ℰ f
    f-is-monotone (g₁ , d₁) (g₂ , d₂) (g₁⊑g₂ , d₁⊑d₂) =
                  f (g₁ , d₁) ⊑⟨ ℰ ⟩[ continuous-functions-are-monotone 𝒟 ℰ g₁ d₁ d₂ d₁⊑d₂ ]
                  f (g₁ , d₂) ⊑⟨ ℰ ⟩[ g₁⊑g₂ d₂ ]
                  f (g₂ , d₂) ■⟨ ℰ ⟩
    c : is-continuous ((𝒟 ⟹^dcpo ℰ) ×^dcpo 𝒟) ℰ f
    c = continuous-in-arguments→continuous (𝒟 ⟹^dcpo ℰ) 𝒟 ℰ f continuous₁ continuous₂
      where
        continuous₁ : (d : ⟨ 𝒟 ⟩) → is-continuous (𝒟 ⟹^dcpo ℰ) ℰ (λ g → f (g , d))
        continuous₁ d I α δ = ∐-is-sup ℰ (pointwise-family-is-directed 𝒟 ℰ α δ d)
        continuous₂ : (g : ⟨ 𝒟 ⟹^dcpo ℰ ⟩) → is-continuous 𝒟 ℰ (λ d → f (g , d))
        continuous₂ g I α δ = u , l
          where
            u : is-upperbound (underlying-order ℰ) (f (g , ∐ 𝒟 δ)) (λ z → f (g , α z))
            u i = f-is-monotone (g , α i) (g , ∐ 𝒟 δ)
                                ((reflexivity (𝒟 ⟹^dcpo ℰ) g) , (∐-is-upperbound 𝒟 δ i))
            l : (u₁ : ⟨ ℰ ⟩) → ((i : I) → f (g , α i) ⊑⟨ ℰ ⟩ u₁) → f (g , ∐ 𝒟 δ) ⊑⟨ ℰ ⟩ u₁
            l u₁ p = transport (λ - → - ⊑⟨ ℰ ⟩ u₁) (i ⁻¹) ii
              where
                i : f (g , ∐ 𝒟 δ) ≡ ∐ ℰ (image-is-directed 𝒟 ℰ g δ)
                i = continuous-function-∐-≡ 𝒟 ℰ g δ
                ii : (∐ ℰ (image-is-directed 𝒟 ℰ g δ)) ⊑⟨ ℰ ⟩ u₁
                ii = ∐-is-lowerbound-of-upperbounds ℰ (image-is-directed 𝒟 ℰ g δ) u₁ p
```