Complete Partial Orders, PCF, and Control

Andrew R. Plummer

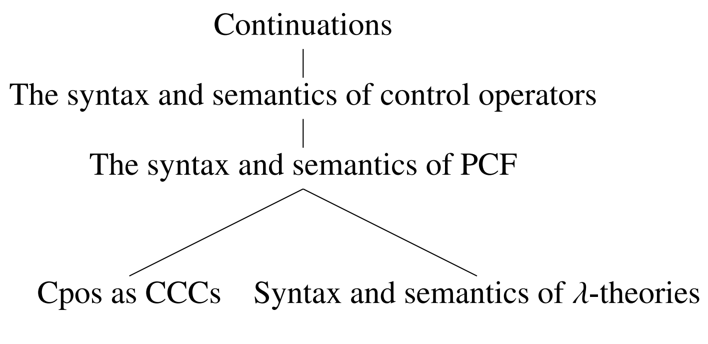
TIE Report Draft – February 2010

Abstract

We develop the theory of directed complete partial orders and complete partial orders. We review the syntax of the abstract programming language PCF, and present its interpretation into the cartesian closed category of complete partial orders. We then present category theoretic interpretations for control operators. This material is mostly drawn from Chapters 1, 6, and 8 in Amadio & Curien (1998).

Introduction

Just a few words about where this is going. We want to understand continuations, so we have determined the following path to understanding them:


Continuations
The syntax and semantics of control operators
The syntax and semantics of PCF
Cpos as CCCs
Syntax and semantics of λ-theories

We assume familiarity with the syntax and semantics of λ-theories, thus it will be covered only in passing. We assume the reader has some exposure to category theory, though we will present enough of it for readers without extensive experience. Thus, we begin with the category theory of cpos (Section 1), and work toward the syntax and semantics of PCF (Section 3). We then present Scott topologies and the category of dcpos with partial continuous functions as arrows (Section 2). We then present the syntax and semantics of control operators (Section 5).

1 Directed Complete Partial Orders

1.1 Definitions and Examples

Definition 1. Given a partial order $(D, \leq)$, a non-empty subset $\Delta \subseteq D$ is called *directed* if, for all $x, y \in \Delta$, there is a $z \in \Delta$ such that $x \leq z$ and $y \leq z$.

We write $\Delta \subseteq_{dir} D$ if Δ is a directed subset of D.

Example 1. The natural numbers $\mathbb{N}$ with the usual $\leq$ relation is a partial order $(\mathbb{N}, \leq)$. Let Δ be any subset of $\mathbb{N}$, and consider $x, y \in \Delta$. It is easy to see that $z = \max(x, y) \in \Delta$ is such that $x \leq z$ and $y \leq z$. That is, $\Delta \subseteq_{dir} \mathbb{N}$.

Definition 2. A partial order $(D, \leq)$ is called a *directed complete partial order* (dcpo) if each $\Delta \subseteq_{dir} D$ has a least upper bound (lub), denoted $\bigvee \Delta$. A directed complete partial order that has a least element, denoted $\bot$, is called a *complete partial order* (cpo).

Example 2. Let $\mathbb{N}_n$ be the set of the first n natural numbers. Then $(\mathbb{N}_n, \leq)$ is a dcpo. Moreover, since $0 \leq k$ for all $k \in \mathbb{N}_n$, $(\mathbb{N}_n, \leq)$ is a cpo with least element 0. Notice that $(\mathbb{N}, \leq)$ is not a dcpo, since the set of odd natural numbers has no least upper bound.

Example 3. Let $\mathbb{Z}_n$ be the set of integers k such that $k \leq n$. Then $(\mathbb{Z}_n, \leq)$ is a dcpo. Notice that $(\mathbb{Z}_n, \leq)$ is not a cpo, since it has no least element.

Let $(D, \leq)$ be a partial order. An *infinite ascending chain* is a sequence $x_0, x_1, \ldots, x_n, \ldots$ of distinct elements in D such that $x_0 \leq x_1 \leq \cdots \leq x_n \cdots$. The unbounded sets $\mathbb{N}$ and $\mathbb{Z}$ are not suitable for forming dcpos, as they contain infinite ascending chains (e.g. the set of odd natural numbers). The upper bounds on their counterparts $\mathbb{N}_n$ and $\mathbb{Z}_n$ disallow infinite ascending chains, making both $\mathbb{N}_n$ and $\mathbb{Z}_n$ suitable for dcpos. We can generalize this observation.

Example 4. All partial orders without infinite ascending chains are dcpos.

The following example provides a method for constructing a cpo out of any given set.

Example 5. Let X be any set. Define $X_\perp = X \cup \{\perp\}$ where $\perp \notin X$, and for $x, y \in X_\perp$, define $x \leq y$ iff $x = \perp$ or $x = y$. Then $(X_\perp, \leq)$ is a cpo.

A cpo $X_\perp$ constructed this way is called *flat*, due to the appearance of its hasse diagram. The following examples of flat cpos will be important in later developments.

Example 6. Let $\mathbf{B} = \{tt, ff\}$, where tt and ff are called *truth values*. Let ω be the smallest infinite limit ordinal. Then both $\mathbf{B}_\perp$ and $\omega_\perp$ are flat cpos.

Definition 3. Let $(D, \leq)$ and $(D', \leq')$ be partial orders. A function $f : D \to D'$ is called *monotonic* if, for all $x, y \in D$, if $x \leq y$, then $f(x) \leq' f(y)$.

Example 7. Let $f : \mathbb{N} \to \mathbb{N}$ be the function $f(n) = n + 1$. Then f is monotonic, given the partial order $(\mathbb{N}, \leq)$.

Example 8. Let $f : \mathbb{N}_n \to \mathbb{N}_{n+1}$ be defined as above. Then f is monotonic, given the partial orders $(\mathbb{N}_n, \leq)$ and $(\mathbb{N}_{n+1}, \leq)$.

Definition 4. Let $(D, \leq)$ and $(D', \leq')$ be dcpos. A function $f : D \to D'$ is called *continuous* if f is monotonic and, for all $\Delta \subseteq_{dir} D$, $f(\bigvee \Delta) = \bigvee f(\Delta)$[1].

When the second condition in the definition is satisfied, we say that f preserves directed lubs.

Example 9. Consider $(\mathbb{N}_n, \leq)$ and $(\mathbb{N}_{n+1}, \leq)$, and $f : \mathbb{N}_n \to \mathbb{N}_{n+1}$ defined in Example 7. We want to show that f is continuous. We have already established that f is monotonic. We just need to show that f preserves directed lubs. Let $\Delta \subseteq_{dir} \mathbb{N}_n$. Then $\bigvee \Delta$ is simply the largest natural number

[1] $f(\Delta) = \{f(\delta) \mid \delta \in \Delta\}$.

in Δ, call it δ. Since $k \leq \delta$ for all $k \in \Delta$, it follows from the monotonicity of f that $f(k) \leq f(\delta)$ for all $k \in \Delta$. That is, $f(\delta)$ is the largest natural number in $f(\Delta)$, hence $\bigvee f(\Delta) = f(\delta) = f(\bigvee \Delta)$. Since Δ was arbitrary, f is continuous.

Proposition 1. *Let $(D, \leq)$ be a dcpo, and let id_D be the identity function on D. Then id_D is a continuous function.*

Proof. For $x, y \in D$ such that $x \leq y$, clearly $id_D(x) = x \leq y = id_D(y)$. Let $\Delta \subseteq_{dir} D$. We have $id_D(\bigvee \Delta) = \bigvee \Delta$, and $\bigvee id_D(\Delta) = \bigvee \Delta$. □

Proposition 2. *Let $(D, \leq)$, $(D', \leq')$, and $(D'', \leq'')$ be dcpos, and let $f : D \to D'$ and $g : D' \to D''$ be continous functions. Then $g \circ f : D \to D''$ is continuous.*

Proof. Let $x, y \in D$ such that $x \leq y$. Then $f(x) \leq' f(y)$, and so $g(f(x)) \leq'' g(f(y))$. Thus, $g \circ f$ is monotonic. Let $\Delta \subseteq_{dir} D$. We need to show that $g(f(\bigvee \Delta)) = \bigvee g(f(\Delta))$. Since f is continuous, $g(f(\bigvee \Delta)) = g(\bigvee f(\Delta))$, and since g is continuous, $g(\bigvee f(\Delta)) = \bigvee g(f(\Delta))$. □

1.2 The Categories Dcpo and Cpo

Propositions 1 and 2 allow us to define the following categories.

Definition 5. The category **Dcpo** has dcpos as objects and continuous functions as arrows. The category **Cpo** is the full subcategory of **Dcpo** with cpos as objects.

Checking in detail that **Dcpo** and **Cpo** are categories is left as an exercise. We now want to show that **Dcpo** and **Cpo** are cartesian closed categories (ccc), and thus induce λ-theories. We need to show the following:

1. There is a cpo $(T, \leq)$ such that for every dcpo $(D, \leq')$, there is exactly one continuous function $1_D : D \to T$. A cpo $(T, \leq)$ satisfying this condition is called a *terminal object*.
2. Given dcpos $(D, \leq)$ and $(D', \leq')$, there is a dcpo $(D \times D', \leq_\times)$ equipped with continuous functions π_1 and π_2 such that for any dcpo $(E, \leq''$ $)$ with continuous functions $f : E \to D$ and $g : E \to D'$, there

is a unique continuous function $\langle f, g\rangle : E \to D \times D'$, defined as $\langle f, g\rangle(x) = \langle f(x), g(x)\rangle$, such that $f = \pi_1 \circ \langle f, g\rangle$ and $g = \pi_2 \circ \langle f, g\rangle$. A dcpo $(D \times D', \leq_\times)$ equipped with continuous functions π_1 and π_2 is called a *product* of D and D'.

3. Given dcpos $(D, \leq)$, $(D', \leq')$, and $(E, \leq'')$, and a continuous function $f : D \times D' \to E$, there is a dcpo $(D' \to_{cont} E, \leq_{ext})$ equipped with continuous functions

$$curry(f) : D \to (D' \to E) \quad \text{and} \quad eval : ((D' \to E) \times D') \to E,$$

where $curry(f)$ is unique, such that $curry(f) \times id_{D'} : (D \times D') \to ((D' \to E) \times D')$ is continuous, and $f = eval \circ curry(f) \times id_{D'}$. A dcpo $(D' \to_{cont} E, \leq_{ext})$ equipped with continuous functions $curry(f)$ and $eval$ is called an *exponent* of D, D', and E.

It is easy to see that the cpo $(\{\bot\}, \leq_\bot)$ is a terminal object. The requirements 2. and 3. are more involved. We begin with 2.

Let $(D, \leq)$ and $(D', \leq')$ be dcpos. Define $(D \times D', \leq_\times)$ as follows: $D \times D'$ is the cartesian product of D and D', and $(x, x') \leq_\times (y, y')$ iff $x \leq y$ and $x' \leq' y'$. Moreover, define $\pi_1 : D \times D' \to D$ as $\pi_1(\langle x, x'\rangle) = x$, and $\pi_2 : D \times D' \to D'$ as $\pi_2(\langle x, x'\rangle) = x'$.

Proposition 3 (Products in **Dcpo**). *Let $(D, \leq)$ and $(D', \leq')$ be dcpos. Then $(D \times D', \leq_\times)$ with π_1 and π_2 is a product of D and D'.*

Proof. We first need to show that $(D \times D', \leq_\times)$ is a dcpo. Let $\Delta \subseteq_{dir} D \times D'$. Define $\Delta_D = \{x \mid \langle x, x'\rangle \in \Delta\}$ and $\Delta_{D'} = \{x' \mid \langle x, x'\rangle \in \Delta\}$. Then $\langle \bigvee \Delta_D, \bigvee \Delta_{D'}\rangle$ is the lub of Δ. To see why, let $\langle z, z'\rangle$ be an upper bound for Δ. Then, z is an upper bound for Δ_D and z' is an upper bound for $\Delta_{D'}$. Since $\bigvee \Delta_D$ is a lub for Δ_D, it follows that $\bigvee \Delta_D \leq z$. Similarly, $\bigvee \Delta_{D'} \leq' z'$. Thus, $\langle \bigvee \Delta_D, \bigvee \Delta_{D'}\rangle \leq_\times \langle z, z'\rangle$.

We also need to show that π_1 and π_2 are continuous. By symmetry, we only need consider π_1. Assume $\langle x, x'\rangle \leq_\times \langle y, y'\rangle$. Then $\pi_1(\langle x, x'\rangle) = x \leq y = \pi_1(\langle y, y'\rangle)$. Thus, π_1 is monotonic. Let $\Delta \subseteq_{dir} D \times D'$. Then $\pi_1(\bigvee \Delta) = \bigvee \Delta_D$. Moreover, since $\pi_1(\Delta) = \Delta_D$, it follows that $\bigvee \pi_1(\Delta) = \bigvee \Delta_D$. Thus π_1 is continuous.

Finally, let $(E, \leq'')$ be a dcpo with continuous functions $f : E \to D$ and $g : E \to D'$. We need to show that there is a unique function $\langle f, g\rangle : E \to$

$D \times D'$ such that $f = \pi_1 \circ \langle f, g \rangle$ and $g = \pi_2 \circ \langle f, g \rangle$. First, we show that $\langle f, g \rangle$ is continuous.

Let $x \leq'' y$. Since f and g are continuous, it follows that $f(x) \leq f(y)$ and $g(x) \leq' g(y)$. Thus, $\langle f(x), g(x) \rangle \leq_\times \langle f(y), g(y) \rangle$. Hence $\langle f, g \rangle$ is monotonic. Let $\Delta \subseteq_{dir} E$. Since $\langle f, g \rangle$ is monotonic, we have $\bigvee \langle f, g \rangle(\Delta) \leq_\times \langle f, g \rangle(\bigvee \Delta)$. Since f and g are continuous, we have the following:

$$\begin{aligned} \bigvee\{f(x) \mid x \in \Delta\} &= f(\bigvee \Delta) = f(\bigvee \Delta) \\ \bigvee\{g(x) \mid x \in \Delta\} &= g(\bigvee \Delta) = g(\bigvee \Delta). \end{aligned}$$

Let $\bigvee \langle f, g \rangle(\Delta) = \langle z, z' \rangle$. Then z is an upper bound for $\{f(x) \mid x \in \Delta\}$ and z' is an upper bound for $\{g(x) \mid x \in \Delta\}$. Thus $\bigvee\{f(x) \mid x \in \Delta\} \leq z$ and $\bigvee\{g(x) \mid x \in \Delta\} \leq' z'$. Hence $\langle f, g \rangle(\bigvee \Delta) \leq_\times \bigvee \langle f, g \rangle(\Delta)$. Extensionality ensures the uniqueness of $\langle f, g \rangle$. □

We have shown that the category **Dcpo** is closed under products. A simple addition to the argument shows that **Cpo** is also closed under products. We now move on to 3.

Let $(D, \leq)$ and $(D', \leq')$ be dcpos. Define $(D \rightarrow_{cont} D', \leq_{ext})$ as follows: $D \rightarrow_{cont} D'$ is the set of continuous functions from D to D', and $f \leq_{ext} g$ iff for all $x \in D$, $f(x) \leq' g(x)$.

We need the following lemmas. The proofs are left as an exercise.

Lemma 1. *Let $(D, \leq)$ and $(D', \leq')$ be dcpos. Then $(D \rightarrow_{cont} D', \leq_{ext})$ is a dcpo. If $(D, \leq)$ and $(D', \leq')$ are cpos, then $(D \rightarrow_{cont} D', \leq_{ext})$ is a cpo.*

Lemma 2. *Let $(D, \leq)$, $(D', \leq')$, and $(E, \leq'')$ be dcpos. A function $f : D \times D' \rightarrow E$ is continuous iff for all $x \in D$ ($y \in D'$) the functions $f_x : D' \rightarrow E$ ($f_y(x) : D \rightarrow E$) defined by $f_x(y) = f(\langle x, y \rangle)$ ($f_y(x) = f(\langle x, y \rangle)$) are continuous.*

Let $(D, \leq)$, $(D', \leq')$, $(E, \leq)$, and $(E', \leq')$ be dcpos, and let $f : D \rightarrow D'$ and $g : E \rightarrow E'$ be continuous functions. Define $f \times g : D(\times E) \rightarrow (D' \times E')$ as $f \times g(\langle x, x' \rangle) = \langle f(x), g(x') \rangle$.

Proposition 4. *The function $f \times g(\langle x, x' \rangle) = \langle f(x), g(x') \rangle$ is continuous.*

Proof. The proof is similar to that of $\langle f, g \rangle$. □

Proposition 5 (Exponents in **Dcpo**). *Let $(D, \leq)$, $(D', \leq')$ and $(E, \leq'')$ be dcpos, and let $f : D \times D' \to E$ be a continuous function. Then the dcpo $(D' \to_{cont} E, \leq_{ext})$ equipped with continuous functions*

$$curry(f) : D \to (D' \to E) \quad \text{and} \quad eval : ((D' \to E) \times D') \to E,$$

defined as $curry(f)(x)(y) = f(\langle x, y\rangle)$ and $eval(\langle f, x\rangle) = f(x)$, respectively, is an exponent of D, D', and E.

Proof. We need to show that $curry(f)$ is continuous. Let $x \leq y$. Notice that $curry(f)(x) = f_x(x')$ and $curry(f)(y) = f_y(x')$ for $x' \in D'$. Let $x' \in D'$. Then $f_x(x') = f(\langle x, x'\rangle)$ and $f_y(x') = f(\langle y, x'\rangle)$. Since $\langle x, x'\rangle \leq_\times \langle y, x'\rangle$ and f is continuous, it follows that $f(\langle x, x'\rangle) \leq'' f(\langle y, x'\rangle)$. Since x' was arbitrary, $curry(f)(x) \leq_{ext} curry(f)(y)$.

To see that $curry(f)(\bigvee \Delta) \leq_{ext} \bigvee curry(f)(\Delta)$, let $x \in D'$. Then

$$curry(f)(\bigvee \Delta)(x') = f(\langle \bigvee \Delta, x'\rangle) = \bigvee f(\langle \Delta, x'\rangle) = \bigvee\{f(\langle \delta, x'\rangle) \mid \delta \in \Delta\}.$$

Let $g = \bigvee curry(f)(\Delta) = \bigvee\{curry(f)(\delta) \mid \delta \in \Delta\}$. Then for all $\delta \in \Delta$, $f(\langle \delta, x'\rangle) = curry(f)(\delta)(x') \leq'' g(x')$. Since x' was arbitrary, g is an upper bound for $\{f(\langle \delta, x'\rangle) \mid \delta \in \Delta\}$. Hence $\bigvee\{f(\langle \delta, x'\rangle) \mid \delta \in \Delta\} \leq_{ext} g$.

Since $curry(f)$ and $id_{D'}$ are continuous, it follows that $curry(f) \times id_{D'}$ is continuous.

We need to show that $eval$ is continuous. Let $\langle f, x\rangle \leq_\times \langle g, y\rangle$. Then, since $f \leq_{ext} g$, $eval(\langle f, x\rangle) = f(x) \leq'' g(x) \leq'' g(y) = eval(\langle g, y\rangle)$. Thus $eval$ is monotonic.

To show that $eval$ is continuous it is enough to show that $eval(\langle \_, x'\rangle)$ and $eval(\langle f, \_\rangle)$ are continuous for all $f \in D' \to E$ and $x' \in D$, respectively. By extensionality, $eval(\langle f, \_\rangle) = f(\_)$ for all $f \in (D' \to E)$, and is by assumption continuous.

Let $x' \in D'$. The monotonicity of $eval(\langle \_, x'\rangle)$ is trivial since $f \leq_{ext} g$ implies that $eval(\langle f, x'\rangle) = f(x') \leq'' g(x') = eval(\langle g, x'\rangle)$. Let $\Delta \subseteq_{dir} (D' \to E)$. We need to show that $eval(\langle \bigvee \Delta, x'\rangle) = \bigvee eval(\langle \Delta, x'\rangle)$. That is, $(\bigvee \Delta)(x') = \bigvee\{\delta(x') \mid \delta \in \Delta\}$. This follows from $f(\_) = \bigvee \Delta(\_)$ being the lub of Δ.

To see that $f = eval \circ curry(f) \times id_{D'}$, let $\langle x, x'\rangle \in D \times D'$. Then $curry(f) \times id_{D'}(\langle x, x'\rangle) = \langle curry(f)(x), x'\rangle$, and $eval(\langle curry(f)(x), x'\rangle) = f(\langle x, x'\rangle)$. $\square$

We have shown that **Dcpo** satisfies 1., 2. and 3. and so is a bona fide ccc. It is easy to augment the proofs above to directly show that **Cpo** is also a ccc. In Section 3, we show that the abstract programming language PCF is naturally interpreted in **Cpo**.

2 Scott Topologies

2.1 Topologies and Bases

Before proceeding, we need a few definitions and concepts from basic topology.

Definition 6. Let X be a set. A *topology* on X is a set $T \subseteq \wp(X)$ such that $\emptyset$ and X are in T, and T is closed under infinite unions and finite intersections. A *topological space* is a set X together with a topology T on X, typically written as the ordered pair (X, T).

Let (X, T) be a topological space. The sets in T are called *open*, and the complements of open sets are called *closed*. For example, both X and $\emptyset$ are in T, and thus open. Moreover, $X^c = \emptyset$ and $\emptyset^c = X$, thus both X and $\emptyset$ are also closed. Sets that are both open and closed are called *clopen*.

Example 10. Let W be any set, and let $P = \wp(X)$. Then (W, P) is a topological space. Clearly, $\emptyset, W \in P$, by definition of powerset, and it is easy to verify that P is closed under both union and intersection.[2]

Definition 7. A *basis* over X is a collection $\mathfrak{B}$ of subsets of X such that

1. For each $x \in X$, there is at least one basis element B containing x.
2. If x belongs to the intersection of two basis elements B_1 and B_2, then there is a basis element B_3 containing x such that $B_3 \subseteq B_1 \cap B_2$.

Definition 8. If $\mathfrak{B}$ satisfies the two conditions in Definition 7, then we define the *topology generated by* $\mathfrak{B}$ as follows: A subset U of X is open if for each $x \in U$, there is a basis element $B \in \mathfrak{B}$ such that $x \in B$ and $B \subseteq U$. If T is the topology generated by $\mathfrak{B}$, then $\mathfrak{B}$ is called *a basis for* T.

[2]Formal semanticists can think of W as the set of possible worlds and P as the set of all propositions.

Example 11. Let $Q = \{(q_1, q_2) \mid q_1, q_2 \in \mathbb{Q}, q_1 < q_2\}$. Then Q is a basis for $\mathbb{R}$. Indeed, let $r \in \mathbb{R}$. Then the open interval $(int(r) - 1, int(r) + 1) \in Q$ contains r[3]. Let $(q_1, q_2), (p_1, p_2) \in Q$ such that both contain r. If $(q_1, q_2) \subseteq (p_1, p_2)$, then (q_1, q_2) serves as the set we need. A similar argument holds if $(p_1, p_2) \subseteq (q_1, q_2)$, Assume that $(q_1, q_2) \nsubseteq (p_1, p_2)$, and without loss of generality, assume that $q_2 < p_2$. Then $r \in (p_1, q_2) \subseteq (q_1, q_2) \cap (p_1, p_2)$.

Proposition 6. *Let X be a set, and let $\mathfrak{B}$ be a basis for a topology T on X. Then T is the collection of all unions of elements of $\mathfrak{B}$.*

Example 12. Given Example 11, we have that $\mathfrak{Q} = \{\bigcup Q' \mid Q' \subseteq Q\}$ is the topology generated by Q. Moreover, $(\mathbb{R}, \mathfrak{Q})$ is a topological space.

Definition 9. Let (X, T) and (Y, S) be topological spaces. A function $f : X \to Y$ is called *continuous* if, for all open sets $U \in S$, $f^{-1}(U) \in T$.

Example 13. Let $(\mathbb{R}, \mathfrak{Q})$ be the topological space in Example 12, and let $f : \mathbb{R} \to \mathbb{R}$ be the function $f(x) = x^2$. Then f is continuous (in the topological sense). To see why this is, consider an open set $U = (4, 25)$. Then $f^{-1}(U) = (-5, -2) \cup (2, 5)$. Since $(-5, -2)$ and $(2, 5)$ are open, so is $f^{-1}(U)$. Thus the inverse image of U is open. This is true in general, thus f is continuous.

2.2 Scott Topologies

We now look at topologies based on partial orders. Let $(D, \leq)$ be a partial order. A subset $A \subseteq D$ is *upper closed under* $\leq$. if, for $x \in A$ and $x \leq y$, it follows that $y \in A$. The *Alexandrov topology on D*, denoted $\mathfrak{A}$, is the set of upper closed subsets of D. That is, $(D, \mathfrak{A})$ is a topological space.

We can also recover partial orders from (T_0) topologies. Let (X, T) be a (T_0) topological space. Define the *specialization order on X* as $x \leq y$ iff for all $U \in T$, if $x \in U$, then $y \in U$. It is easy to verify that $\leq$ is a partial order (assuming T is T_0).

We now define topologies that are based on dcpos.

Definition 10. Let $(D, \leq)$ be a dcpo. A subset $A \subseteq D$ is called *Scott open* if the following hold:

[3] $int(r)$ is the integer portion of r.

- A is *upper closed under* $\leq$, i.e. if $x \in A$ and $x \leq y$, then $y \in A$.
- If $\Delta \subseteq$ is directed and $\bigvee \Delta \in A$, then there is an $x \in \Delta$ such that $x \in A$.

Proposition 7. *Let $(D, \leq)$ be a dcpo and let Ω_D be the set of Scott open subsets of D. Then (D, Ω_D) is a topological space.*

Proof. Clearly, $\emptyset, D \in \Omega_D$. Assume $S \subseteq \Omega_D$ and let $x \in \bigcup S$. Then $x \in A$ for some $A \in S$. Let $y \in D$ such that $x \leq y$. Since A is upper closed under $\leq$, it follows that $y \in A$. Thus $y \in \bigcup S$. Let $\Delta \subseteq_{dir} D$ such that $\bigvee \Delta \in \bigcup S$. Then $\bigvee \Delta \in A$ for some $A \in S$. Since A is Scott open, there is a $z \in \Delta$ such that $z \in A$. Thus $z \in \bigcup S$.

Assume S is of finite cardinality. Let $x \in \bigcap S$. Then $x \in A$ for each $A \in S$. Let $y \in D$ such that $x \leq y$. Since each A is upper closed under $\leq$, it follows that $y \in A$, for each A. Thus $y \in \bigcap S$. Let $\Delta \subseteq_{dir} D$ such that $\bigvee \Delta \in \bigcap S$. Then $\bigvee \Delta \in A$ for each $A \in S$. Since A is Scott open, for each A, there is a $z_A \in \Delta$ such that $z_A \in A$. Since Δ is directed, there is a $z \in \Delta$ such that $z_A \leq z$ for each z_A. Thus $z \in A$ for each A. Hence $z \in \bigcap S$. □

The topological space (D, Ω_D) is called the *Scott topology* over D.

Lemma 3. *Let $(D, \leq)$ be a dcpo. The specialization order $\leq'$ on (D, Ω_D) is the partial order $\leq$.*

Proof. We need to show set equality of the relations $\leq'$ and $\leq$. Let $(x, y) \in \leq$, and let $U \in \Omega_D$. Suppose $x \in U$. Since U is upper closed under $\leq$, and $x \leq y$, it follows that $y \in U$. That is, $(x, y) \in \leq'$. Thus, $\leq \subseteq \leq'$. Now, let $(x', y') \in \leq'$. Notice that $U_{y'} = \{a \in D \mid a \nleq y'\}$ is Scott open, and hence in Ω_D. Suppose $x' \nleq y'$. Then $x' \in U_{y'}$, and since $(x', y') \in \leq'$, it follows that $y' \in U_{y'}$. Yet, this is a contradiction. Thus $x' \leq y'$, and so $\leq' \subseteq \leq$. □

Lemma 4. *Let $(D, \leq)$ and $(D', \leq')$ be dcpos. The (topologically) continuous functions from (D, Ω_D) to $(D', \Omega_{D'})$ are the (order theoretic) continuous functions from $(D, \leq)$ to $(D', \leq')$.*

Proof. Let $f : D \to D'$ be (topologically) continuous. We need to show that f is monotonic. Assume $x \leq y$. We need to show that $f(x) \leq' f(y)$. Let $U' \in \Omega_{D'}$ be any open set containing $f(x)$. Now, $f^{-1}(U')$ is an open set containing x, and thus contains y. Hence, $f(y) \in U'$. Since U' was arbitrary, it follows that $x \leq' y$.

We need to show that $f(\bigvee \Delta) = \bigvee f(\Delta)$, for directed Δ. By monotonicity, we immediately have that $\bigvee f(\Delta) \leq' f(\bigvee \Delta)$. Suppose $f(\bigvee \Delta) \nleq' \bigvee f(\Delta)$. Then $f(\bigvee \Delta) \in U_{\bigvee f(\Delta)} = \{a \in D' \mid a \nleq' \bigvee f(\Delta)\}$. Thus, $\bigvee \Delta \in f^{-1}(U_{\bigvee f(\Delta)})$, and so $f(\delta) \in U_{\bigvee f(\Delta)}$ for some $\delta \in \Delta$. This is a contradiction since $f(\delta) \leq' \bigvee f(\Delta)$. □

2.3 Algebraicity and Partial Continuity

Definition 11. Let $(D, \leq)$ be a dcpo. an element $d \in D$ is called *compact* if, for each $\Delta \subseteq_{dir} D$, from $d \leq \bigvee Delta$ it follows that there is $x \in \Delta$ such that $d \leq x$.

We denote the set of compact elements of D by $K(D)$. That is, $K(D) = \{d \in D \mid d \text{ is compact }\}$. The set $K(D)$ is called the *basis* of D. We will show that the elements of $K(D)$ indeed yield a basis for the Scott topology on (D, Ω_D), when the following definition holds on $(D, \leq)$.

Definition 12. Let $(D, \leq)$ be a dcpo. Then $(D, \leq)$ is called *algebraic* if for all $x \in D$ the set $K(D)_x = \{d \in K(D) \mid d \leq x\}$ is directed, and $\bigvee K(D)_x = x$.

The elements of $K(D)_x$ are called *approximants* of x. To see why, consider the following example.

Example 14. The dcpo $(\wp(\omega), \subseteq)$ is algebraic. Let Δ_{odd} be the set of all sets of odd numbers. Then $\bigvee \Delta_{odd} = \{a \in \omega \mid a \text{ is odd }\}$. Now, $\bigvee \Delta_{odd}$ is an infinite set, and $K(D)_{\bigvee \Delta_{odd}}$ is the set of all finite sets of odd numbers. That is, $K(D)_{\bigvee \Delta_{odd}} = \{\{1\}, \{1, 3\}, \{1, 3, 5\}, \ldots\}$. Each set in $K(D)_{\bigvee \Delta_{odd}}$ is an approximation of $\bigvee \Delta_{odd}$, and as the sets in $K(D)_{\bigvee \Delta_{odd}}$ get larger, they better approximate $\bigvee \Delta_{odd}$.

Proposition 8. *Let $(D, \leq)$ be and algebraic dcpo. Let $\uparrow d = \{x \in D \mid d \leq x\}$. Then $B = \{\uparrow d \mid d \in K(D)\}$ is a basis for (D, Ω_D).*

Proof. We need to show that B is a basis. Let $x \in D$. Since D is algebraic, $x = \bigvee K(D)_x$, and so $x \in \uparrow d$ for each $d \in K(D)_x$. Let $x \in \uparrow d \cap \uparrow e$. Then

$d, e \in K(D)_x$. Since $K(D)_x$ is directed, there is a $z \in K(D)_x$ such that $d \leq z$ and $e \leq z$. Thus $\uparrow z \subseteq \uparrow d$ and $\uparrow z \subseteq \uparrow e$. Thus $\uparrow z \subseteq (\uparrow d \cap \uparrow e)$.

Finally, notice that each $\uparrow d$ is Scott open. Indeed, let $\bigvee \Delta \in \uparrow d$. Then $d \leq \bigvee \Delta$, and by compactness of d, there is a $z \in \Delta$ such that $d \leq z$. Hence $z \in \uparrow d$. Let $A \in \Omega_D$, and let $x \in A$. Since D is algebraic, and since $\bigvee K(D)_x = x \in A$, there is a compact d such that $d \in A$. Thus, $\uparrow d \subseteq A$. Thus Ω_D is the topology generated by B. □

We conclude this section with a definition of partial continuous function, and its role in the categorical theory of dcpos. Let X and Y be sets. A *partial function* $f : X \to Y$ is a function $f : X' \to Y$ such that $X' \subseteq X$. The set X' is called the *domain* of f, and is denoted $dom(f)$.

Definition 13. Let $(D, \leq)$ and $(D', \leq')$ be dcpos. A partial function $f : D \to D'$ is called *continuous* if $dom(f)$ is Scott open, and f restricted to $dom(f)$ is continuous. That is, $f : dom(f) \to D'$ is continuous.

We will use the following category in our discussion of monads. For now, we simply give a definition.

Definition 14. The category **pDcpo** has dcpos as objects and partial continuous functions as arrows.

3 Abstract Programming

4 Monads

4.1 Functors

In this section, we introduce transformation from one category to another. These transformations, called *functors*, are essential to many theoretical concepts in programming language theory, and practical applications in natural language theory. For example, in natural language theory the mapping between a syntactic logic and a semantic logic is typically a functor, so long as the logics themselves constitute categories. In programming language theory, the concept of functional programming is founded entirely on category theory, and functors are used to handle interactions between programs and the real world. We will come back to these issues later. At present, we formalize functors.

Definition 15. A *functor F* from a category $\mathbf{C}$ to a category $\mathbf{D}$ is a function such that:

1. Each object $a \in Ob(\mathbf{C})$ is mapped to an object $F(a) \in Ob(\mathbf{D})$.
2. Each arrow $f : a \to b$ in $Ar(\mathbf{C})$ is mapped to an arrow $F(f) : F(a) \to F(b)$ in $Ar(\mathbf{D})$ such that:
   (a) For all objects $a \in Ob(\mathbf{C})$, the identity arrow on a is mapped to the identity arrow on $F(a)$. That is,
   $$F(1_a) = 1_{F(a)}.$$
   (b) For all pairs $\langle g, f\rangle$ where $f, g \in Ar(\mathbf{C})$ and cod f = dom g,
   $$F(g \circ f) = F(g) \circ F(f).$$

Condition (b) is better understood visually, using the following commutative diagrams:

$$\begin{tikzcd}
a \arrow[r, "f"] \arrow[dr, "g\circ f"'] & b \arrow[d, "g"] \\
 & c
\end{tikzcd}
\qquad
\begin{tikzcd}
F(a) \arrow[r, "F(f)"] \arrow[dr, "F(g\circ f)"'] & F(b) \arrow[d, "F(g)"] \\
 & F(c)
\end{tikzcd}$$

If F is a functor from $\mathbf{C}$ to$\mathbf{D}$, we write $F : \mathbf{C} \to \mathbf{D}$ or $\mathbf{C} \overset{F}{\to} \mathbf{D}$. Before proceeding, we define the following:

Definition 16 (Identity Functor)**.** The *identity functor* on a category $\mathbf{C}$ is the functor $1_{\mathbf{C}} : \mathbf{C} \to \mathbf{C}$ such that

1. for all $a \in Ob(\mathbf{C})$, $1_{\mathbf{C}}(a) = a$.
2. for all $f \in Ar(\mathbf{C})$, $1_{\mathbf{C}}(f) = f$.

Since functors are functions we immediately have the following:

Definition 17 (Functor Composition)**.** Given functors $F : \mathbf{A} \to \mathbf{B}$ and $G : \mathbf{B} \to \mathbf{C}$, the composite of F and F, denoted $G \circ F$, is the functor defined as $(G \circ F)(a) = G(F(a))$, for all $a \in Ob(\mathbf{A})$; and $(G \circ F)(f) = G(F(f))$, for all $f \in Ar(\mathbf{A})$.

Since functor composition is a special case of function composition, we have that functor composition is associative. Hence, for all functors $F : \mathbf{A} \to \mathbf{B}$, $G : \mathbf{B} \to \mathbf{C}$, and $H : \mathbf{C} \to \mathbf{D}$, we have $H \circ (G \circ F) = (H \circ G) \circ F$. This, together with the identity functor, allows us to think of functors as arrows between categories. Though this approach leads to some complications, it will prove useful later on.

4.2 Natural Transformations

Let us quickly review our theoretical constructs thus far. We defined categories as mathematical constructs that contain objects and arrows. We then defined functors as mappings between categories. At the end of the previous section, we provided the definitions needed to view categories as objects and functors as arrows. That is, categories themselves can be the objects of other categories, and functors the arrows. Now, we take the next step in abstraction, and consider functors as objects of certain categories. To do so, we need a reasonable concept of *arrow between functors*. We thus introduce natural transformations.

Let $\mathbf{C}$ and $\mathbf{D}$ be categories, and let $F : \mathbf{C} \to \mathbf{D}$ and $G : \mathbf{C} \to \mathbf{D}$ be functors from $\mathbf{C}$ to $\mathbf{D}$. The image of F is a representation of the category $\mathbf{C}$ within the category $\mathbf{G}$. The image of G is similarly a representation of $\mathbf{C}$ within $\mathbf{D}$. Suppose we want to translate the representation F yields into the representation that G yields. And we want to do so in a way that preserves the structure of the representation that F yields. We start with an object $a \in Ob(\mathbf{C})$. The object a is mapped to an object $F(a) \in Ob(\mathbf{D})$ and an object $G(a) \in Ob(\mathbf{D})$. We want to translate the object $F(a)$ onto $G(a)$ using an arrow in $\mathbf{D}$. We denote the translation arrow as $\tau_a : F(a) \to G(a)$. Now, F itself preserves the structure of $\mathbf{C}$, that is, given an arrow $f : a \to b$ in $Ar(\mathbf{C})$, we have an arrow $F(f) : F(a) \to F(b)$ in $Ar(\mathbf{D})$. So far, we have the following information:

$$\begin{tikzcd}
a \arrow[d, "f"'] & F(a) \arrow[r, "\tau_a"] \arrow[d, "F(f)"'] & G(a) \\
b & F(b) &
\end{tikzcd}$$

Now, the functor G also acts on f, yielding an arrow $G(f) : G(a) \to G(b)$ in $Ar(\mathbf{D})$. In order to complete the translation, we need to translate $F(f)$ onto $G(f)$. To do this, we first need to translate $F(b)$ onto $G(b)$ using an arrow in $\mathbf{D}$. We denote this translation arrow, as before, as $\tau_b : F(b) \to G(b)$. We now have all the information we need:

$$\begin{tikzcd}
a \arrow[d, "f"'] & F(a) \arrow[r, "\tau_a"] \arrow[d, "F(f)"'] & G(a) \arrow[d, "G(f)"] \\
b & F(b) \arrow[r, "\tau_b"'] & G(b)
\end{tikzcd}$$

We simply check that the above diagram commutes for each $a \in Ob(\mathbf{C})$ and $f : a \to b$ in $Ar(\mathbf{C})$. If so, we have a structure-preserving transformation of the representation of $\mathbf{C}$ in $\mathbf{D}$ given by F into the representation of $\mathbf{C}$ in $\mathbf{D}$ given by G. The cumulative process just described is a *natural transformation* from $\mathbf{C}$ to $\mathbf{D}$. We provide the formal definition below.

Definition 18. Let $\mathbf{C}$ and $\mathbf{D}$ be categories, and let $F : \mathbf{C} \to \mathbf{D}$ and $G : \mathbf{C} \to \mathbf{D}$ be functors from $\mathbf{C}$ to $\mathbf{D}$. A *natural transformation* from F to G is a collection of arrows τ, contained in $Ar(\mathbf{D})$, such that: for each object $a \in Ob(\mathbf{C})$, there is an arrow $\tau_a : F(a) \to G(a)$ in τ, called a *component* of τ, such that for any arrow $f : a \to b$ in $Ar(\mathbf{C})$, given $\tau_b : F(b) \to G(b) \in \tau$, we have $G(f) \circ \tau_a = \tau_b \circ F(f)$. That is, the following diagram commutes:

$$\begin{tikzcd}
F(a) \arrow[r, "\tau_a"] \arrow[d, "F(f)"'] & G(a) \arrow[d, "G(f)"] \\
F(b) \arrow[r, "\tau_b"'] & G(b)
\end{tikzcd}$$

If τ is a natural transformation from **C** to **D**, we write $\tau : \mathbf{C} \to \mathbf{D}$, or $\mathbf{C} \xrightarrow{\tau} \mathbf{D}$,

We conclude this section by providing the definitions needed to form categories of functors. Specifically, given two categories **C** and **D**, we are interested in forming the *functor category* $\mathbf{D}^{\mathbf{C}}$, whose objects are all the functors from **C** to **D**. At this point it should be clear that a category with functors as objects will have natural transformations as arrow. We need to have an identity arrow $1_F : F \to F$ for each functor F.

Definition 19 (Identity Transformation)**.** The *identity transformation* on a functor $F : \mathbf{C} \to \mathbf{D}$ is the natural transformation $1_F : F \to F$ such that for all $a \in Ob(\mathbf{C})$,

$$\tau_a = 1_{F(a)} : F(a) \to F(a).$$

We also need a reasonable definition of natural transformation composition. Let F, G, and H be functors from category **C** to category **D**, and let $\tau : F \to G$ and $\sigma : G \to H$ be natural transformations. Let $f : a \to b$ be an arrow in **C**. Consider the following diagram:

$$\begin{tikzcd}
F(a) \arrow[r, "\tau_a"] \arrow[d, "F(f)"'] & G(a) \arrow[r, "\sigma_a"] \arrow[d, "G(f)"] & H(a) \arrow[d, "H(f)"] \\
F(b) \arrow[r, "\tau_b"'] & G(b) \arrow[r, "\sigma_b"'] & H(b)
\end{tikzcd}$$

To define a natural transformation $(\sigma \circ \tau)$ from F to H, we need to define $(\sigma \circ \tau)_a$ for each $a \in Ob(\mathbf{C})$. Since F, G, and H are functors, the smaller square diagrams commute. Thus,

(1) $H(f) \circ \sigma_a = \sigma_b \circ G(f)$
(2) $G(f) \circ \tau_a = \tau_b \circ F(f)$

which yield:

(3) $H(f) \circ \sigma_a \circ \tau_a = \sigma_b \circ G(f)\tau_a$
(4) $\sigma_b \circ G(f) \circ \tau_a = \sigma_b \circ \tau_b \circ F(f)$

By transitivity, (3) and (4) yield the equality

(5) $H(f) \circ \sigma_a \circ \tau_a = \sigma_b \circ \tau_b \circ F(f).$

We simply let $(\sigma \circ \tau)_a = \sigma_a \circ \tau_a$ for each a. Then (5) satisfies the definition of a component of a natural transformation from F to H.

Definition 20 (Natural Transformation Composition)**.** Let F, G, and H be functors from category $\mathbf{C}$ to category $\mathbf{D}$, and let $\tau : F \to G$ and $\sigma : G \to H$ be natural transformations. The *composite* of τ and σ is the collection of arrows $(\sigma \circ \tau)_a = \sigma_a \circ \tau_a$ for all $a \in Ob(\mathbf{C})$.

Natural transformation composition is, of course, associative. The proof is left to the enthusiastic reader. Thus we have the functor category $\mathbf{D}^{\mathbf{C}}$.

4.3 Introducing Monads

We postpone the mathematical treatment of monads to first develop the intuition behind their usage. We briefly compare two programming paradigms, imperative programming and declarative programming, and show how programs written in the imperative paradigm can be simulated in the declarative paradigm.

In *imperative programming*, a program is a sequence of commands for the computer to perform. For example, a programmer may instruct the computer to take a list of numbers, add one to each number in the list, and then square each number in the list. This sequence of commands in pseudocode is:

Imperative Algorithm

```
Given $A$, a list of length $n$
for $i = 1$ to $n$ do
    $A[i] \leftarrow A[i] + 1$
end for
for $i = 1$ to $n$ do
    $A[i] \leftarrow A[i] * A[i]$
end for
```

The algorithm specifies the commands to be carried out and the sequence in which they are carried out. In contrast, *declarative programming* specifies only the commands, with no explicit sequencing given. To develop the contrast, we focus on functional programming, a kind of declarative programming where all programming constructs are functions. The following functional pseudocode simulates the imperative algorithm above:

Functional Algorithm

```
add1 :: [a] → [a]
add1 [] = []
add1 (n:ns) = [n+1] ++ (add1 ns)

square :: [a] → [a]
square [] = []
square (n:ns) = [n*n] ++ (square ns)

Given a list $A$
square(add1($A$))
```

The functions add1 and square are recursive, and serve to simulate the **for** loops in the imperative algorithm. In general, recursive functions give us a (declarative) construct for simulating (imperative) looping constructs.

Looping constructs are instances of *control constructs* (or *control structures*) – programming constructs that affect the control flow (sequence of command executions) of a program. Command sequencing through use of control constructs is built into the imperative programming paradigm, and more complex control constructs that manipulate command sequencing (e.g. continuations) are immediately available. Since control constructs are at times quite convenient, we want to simulate them in the declarative paradigm. In the remainder of this section, we define declarative constructs, called monads, that provide for the simulation.

Definition 21. A *monad*[4] is a triple $\langle M, \mathsf{return}, \gg\!= \rangle$ where

- M is a function on data types,
- return is a function from data types to data types under M,
- $\gg\!=$ is an operation on functions between data types and data types under M.

Moreover, the following equations are satisfied for all $f : A \to MB$ and $g : B \to MC$.:

1. $(\gg\!= \mathsf{return}) = \mathsf{id}_{MA}$,
2. $((\gg\!= f) \circ \mathsf{return}) = f$,
3. $((\gg\!= g) \circ (\gg\!= f)) = (\gg\!= ((\gg\!= g) \circ f))$.

Haskellers are more familiar with the following equivalent presentation of the monad equations:

H1. $(Ma \gg\!= \mathsf{return}) = Ma$,
H2. $(\mathsf{return}\, a \gg\!= f) = fa$,
H3. $((Ma \gg\!= f) \gg\!= g) = (Ma \gg\!= \lambda x.(fx \gg\!= g))$.

Equations 1. and 2. are visualized as

$$\begin{tikzcd}
A \arrow[r, "\mathsf{return}"] \arrow[dr, "\mathsf{return}"'] & MA \arrow[d, "\gg=\mathsf{return}"] \\
 & MA
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "\mathsf{return}"] \arrow[dr, "f"'] & MA \arrow[d, "\gg= f"] \\
 & MB
\end{tikzcd}$$

[4]Or *kleisli triple* to mathematicians.

Equation 3. is visualized as

$$\begin{tikzcd}
A \arrow[r, "\mathsf{return}"] \arrow[dr, "f"] \arrow[ddr, "(\gg= g)\circ f"'] & MA \arrow[d, "\gg= f"] & \\
& MB \arrow[d, "\gg= g"] & B \arrow[l, "\mathsf{return}"'] \arrow[dl, "g"] \\
& MC &
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "\mathsf{return}"] \arrow[ddr, "(\gg= g)\circ f"'] & MA \arrow[d, "\gg= f"] \\
& MB \arrow[d, "\gg= g"] \\
& MC
\end{tikzcd}$$

Now that we have our definition of monad, we can show how they are used to simulate control constructs. In the next section we will show how to simulate complex control constructs such as continuations. At present, we begin with a very simple example, simulating a **for** loop. Consider the following imperative algorithm that takes a list and produces a list of doubles:

Imperative Algorithm

```
  Given $A$, a list of length $n$
  **for** $i = 1$ to $n$ **do**
      $B[i] \leftarrow (A[i], A[i])$
  **end for**
```

Thus, on input [1,2,3] the algorithm outputs the list of doubles [(1,1),(2,2),(3,3)]. Of course, we can simulate this program using a recursive function, but we would like to use a monad. Fortunately, we know that lists are monadic. We write $[B]$, instead of the usual MB, to indicate that we are in the list monad. Similarly, we write $[b]$ instead of Mb. In the list monad, we have $\mathsf{return}\, a = [a]$. We need to parse the **for** loop and simulate each of its components.

First, we need a function to do the work done inside the **for** loop. Once an element has been taken from the list A it is assigned to the double (a, a). Thus, let f' be a function that maps each a to its double (a, a).

Next, the double (a, a) is placed inside a list. We already have a function that does this, namely return. Let f be the composition $\mathsf{return} \circ f'$, that is f

maps a to $[(a,a)]$. Thus we now have the following:

$$\begin{tikzcd}
Int \arrow[r, "\mathsf{return}"] \arrow[dr, "f"'] & {[Int]} \\
 & {[(Int, Int)]}
\end{tikzcd}$$

Since we are in the list monad, we are guaranteed the existance of a function $(\gg= f)$ such that $(\gg= f) \circ \mathsf{return} = f$. Since $fa = [(a,a)]$, and $\mathsf{return}\,a = [a]$, it must be the case that $(\gg= f)\ [a] = [(a,a)]$.

Finally, we need to simulate the iteration of the **for** loop over the entire list A, and $(\gg= f)$ is just the function we need to do so. We need a way to concatenate the lists $[(a,a)]$ that are yielded by $(\gg= f)$ for each $a \in A$. Fortunately, the monad has already given us this by definition:

$$\begin{array}{rcl}
(\gg= f)\ [a_1, \ldots, a_n] & =_{def} & (\gg= f)\ [a_1] + + \cdots + + (\gg= f)\ [a_n] \\
 & = & [(a_1, a_1), \ldots, (a_n, a_n)].
\end{array}$$

That is, $(\gg= f)$ just is the **for** loop, and so we have simulated the imperative **for** loop with a monad. This kind of simulation holds in general for control constructs, and other aspects of imperative programming:

- **Exceptions**: $MA = (A + E)$ $\quad$ (E a set of exceptions).
- **Side effects**: $MA = (A \times S)^S$ $\quad$ (S a set of states).
- **Continuations**: $MA = R^{(R^A)}$ $\quad$ (R a set of results).

In the next section we will show in detail how to simulate continuations. Before doing so, we present the mathematics of monads.

4.4 Monads Mathematically

5 Continuations and Control

5.1 Call-by-name and Call-by-value

We begin with a λ-theory (see Appendix A) restricted as follows:

- All variables and contants are terms,
- If f and a are terms, then fa is a term,
- If f is a term and x a variable, then $\lambda_x f$ is a term.

All relevant equational rules and equivalences hold over this restricted set of terms. We designate a subset of terms as *values* as follows:

- All variables and constants are values,
- All abstractions $\lambda_x f$ are values.

Suppose we use our typing rules to contruct a term $(\lambda_x M)N$. We can define different ways to reduce such a term to some other term. For example, we might use *beta reduction* to substitute the term N for occurences of the variable x in the term M. We might want to carry out this substitution only under certain circumstances, or we might want to reduce the term N first, or not at all. The different ways we might carry this process out are called *reduction strategies*. Similarly, we might want to carry out this process only if the terms invloved reduce to values. Reduction processes that have such restrictions are called *evaluation strategies*. We will present two approaches to reduction and evaluation called *call-by-name* (CBN) and *call-by-value* (CBV).

For the remainder of this section, we let M and N be metavariables over terms, and V be a metavariable over values. The one-step reduction rules for reduction strategies for CBN and CBV are given in the top portion of Table 1. Each reduction strategy induces a relation between terms. The CBN reduction relation, denoted $\rightarrow_n$, is the smallest set of terms closed under the CBN reduction strategy rules. Similarly, the CBV reduction relation, denoted $\rightarrow_v$, is the smallest set of terms closed under the CBV reduction strategy rules. Notice that $\rightarrow_n$ is more inclusive with respect to beta re-

Reduction Strategies

Call-by-name	Call-by-value
$(\lambda_x M)N \to_n M[N/x]$	$(\lambda_x M)V \to_v M[V/x]$
$\dfrac{M \to_n N}{ML \to_n NL}$	$\dfrac{M \to_v N}{ML \to_v NL}$
	$\dfrac{M \to_v N}{LM \to_v LN}$

Evaluation Strategies

Call-by-name	Call-by-value
$V \mapsto_n V$	$V \mapsto_v V$
$\dfrac{M \mapsto_n \lambda_x M' \quad M'[N/x] \mapsto_n V}{MN \mapsto_n V}$	$\dfrac{M \mapsto_v \lambda_x M' \quad N \mapsto_v V' \quad M'[V'/x] \mapsto_n V}{MN \mapsto_n V}$

Table 1: CBN and CBV reduction and evaluation strategies

dexes, since the argument term in the reduction rule need not be a value. Also note that $\to_v$ is closed under the application of values to terms related by $\to_v$. We denote the reflexive-transitive closure of $\to_n$ by $\to_n^*$, and of $\to_v$ by $\to_v^*$.

The CBN and CBV evaluation strategies are given in lower portion of Table 1. Each reduction strategy also induces a relation between terms. The CBN evaluation relation, denoted $\mapsto_n$, is the smallest set of terms closed under the CBN evaluation strategy rules. Similarly, the CBV evaluation relation, denoted $\mapsto_v$, is the smallest set of terms closed under the CBV evaluation strategy rules. It is somewhat instructive to compare $\to_n^*$ and $\to_v^*$ to $\mapsto_n^*$ and $\mapsto_v^*$, respectively. The comparison tells us the following:

Proposition 9. *Given $\to_v$ and $\to_n$, we have that $\mapsto_v \subseteq \to_v^*$ and $\mapsto_n \subseteq \to_n^*$.*

Hereafter, we will focus primarily on evaluation strategies. In Table 1, we presented rule-based strategies that yielded our evaluation relations. There is an equivalent presentation that uses "holes" in terms. A *context* is a term that is missing a subterm. For example, $\lambda_x x[]$ is a context. A context is *filled* if a term is placed in the context's hole, e.g. the context $\lambda_x x[]$ can be filled by the term $(\lambda_x y)$ to yield the term $\lambda_x x(\lambda_y x)$.

We want to conceptualize a step in computation using contexts. A *program* is a term with no free variables. We need to parse a program into a context E and a leftmost-outermost redex R, and fill E with the contractum of R. Contexts E that provide for this are called *evaluation contexts*. We present a grammar for evaluation contexts for both CBN and CBV evaluation:

- $E_v ::= [\,] \quad | \quad E_v[(V\,[\,])] \quad | \quad E_v[([\,]\,M)]$
- $E_n ::= [\,] \quad | \quad E_n[([\,]\,M)]$

with the evaluation relation defined as:

- $E_v[(\lambda_x MV)] \mapsto_v E_v[(M[V/x])]$
- $E_n[(\lambda_x MN)] \mapsto_n E_n[(M[N/x])]$

Given a program M, we can parse M into an evaluation context E and redex (VN). Then M parsed as $E[(VN)]$ means that (VN) is the current instruction, and E is the rest of the program (e.g. the continuation). An *operational semantics* of a λ-theory is defined as the reflexive-transitive closure of an evaluation relation, and is denoted *Eval*. For example,

- $Eval_v(M) = V$ iff $M \mapsto_v^* V$
- $Eval_n(M) = V$ iff $M \mapsto_n^* V$.

We conclude this discussion with a demonstration of the operational equivalence of CBN and CBV, using a (cps) transformation.

Definition 22. The Fischer CPS transform F is defined as follows:

- $F(x) = \lambda_k kx$
- $F(\lambda_x M) = \lambda_k k(\lambda_a(\lambda_x F(M)a))$
- $F(MN) = \lambda_k F(M)(\lambda_m F(N)\lambda_n(mk)n)$

where a, k, m, n are variables that do not occur in the argument of F.

Theorem 1. *Let $M \in \Lambda$. Then $Eval_v(F(M)\lambda_x x) = Eval_n(F(M)\lambda_x x)$*

V-Projection	$\Gamma, A, \Gamma' \rhd A$	Pairing	$\dfrac{\Gamma \rhd A \qquad \Gamma \rhd B}{\Gamma \rhd A \wedge B}$
Null Product	$\Gamma \rhd \top$	First	$\dfrac{\Gamma \rhd A \wedge B}{\Gamma \rhd A}$
Abstraction	$\dfrac{\Gamma, A \rhd B}{\Gamma \rhd A \to B}$	Second	$\dfrac{\Gamma \rhd A \wedge B}{\Gamma \rhd B}$
Application	$\dfrac{\Gamma \rhd A \to B \qquad \Gamma \rhd A}{\Gamma \rhd B}$	Permute	$\dfrac{\Gamma, B, C, \Gamma' \rhd A}{\Gamma, C, B, \Gamma' \rhd A}$
Contraction	$\dfrac{\Gamma, B, B \rhd A}{\Gamma, B \rhd A}$	Weakening	$\dfrac{\Gamma \rhd A}{\Gamma, B \rhd A}$

Table 2: Positive Intuitionistic Propositional Logic

5.2 The Curry-Howard Isomorphism for CPL

We begin with positive intuitionistic propositional logic (PIPL), presented in Table 2. The (well-formed) formulas of PIPL are generated in the usual way (i.e. using set of atomic symbols and the set of connectives $\{\wedge, \to\}$). Each Γ in Table 2 is a *list* of formulas. We write a list of one element $[A]$ as just A, and denote list concatenation using commas. Equivalently, each we could take each Γ to be a multiset, in which case Permutation can be omitted.

Consider the following two proofs:

$$(1) \qquad \dfrac{A, A \to B \rhd A \to B \qquad A, A \to B \rhd A}{A, A \to B \rhd B}\,\text{App}$$

$$(2) \qquad \dfrac{A \wedge B \rhd A \wedge B}{A \wedge B \rhd B}\,\text{Sec}$$

We have two different proofs (among many) of the formula B. Ultimately, we will identify the formula B with the set of proofs of B. A convenient way

V-Projection	$\Gamma, x : A, \Gamma' \rhd x : A$	Pairing	$\dfrac{\Gamma \rhd a : A \qquad \Gamma \rhd b : B}{\Gamma \rhd (a,b) : A \wedge B}$
Null Product	$\Gamma \rhd * : 1$	First	$\dfrac{\Gamma \rhd h : A \wedge B}{\Gamma \rhd \pi(h) : A}$
Abstraction	$\dfrac{\Gamma, x : A \rhd b : B}{\Gamma \rhd \lambda_x b : A \to B}$	Second	$\dfrac{\Gamma \rhd h : A \wedge B}{\Gamma \rhd \pi'(h) : B}$
Application	$\dfrac{\Gamma \rhd f : A \to B \qquad \Gamma \rhd a : A}{\Gamma \rhd f(a) : B}$	Permute	$\dfrac{\Gamma, x : B, y : C, \Gamma' \rhd a : A}{\Gamma, y : C, x : B, \Gamma' \rhd a : A}$
Contraction	$\dfrac{\Gamma, x : B, x : B \rhd a : A}{\Gamma, x : B \rhd a : A}$	Weakening	$\dfrac{\Gamma \rhd a : A}{\Gamma, x : B \rhd a : A}$

Table 3: Anonymous Typing rules

of keeping track of the steps of the proofs is to adorn each formula with a *term*, and have each rule in Table 2 act as an operation on the terms. For example, we adorn $A \to B$ with a term f and A with a term a. We indicate the adornment as $f : A \to B$ and $a : A$. We modify Application to act on terms as follows:

$$\frac{\Gamma \rhd f : A \to B \qquad \Gamma \rhd a : A}{\Gamma \rhd fa : B}.$$

Our proof (1) now looks like:

$$\frac{a : A, f : A \to B \rhd f : A \to B \qquad a : A, f : A \to B \rhd a : A}{a : A, f : A \to B \rhd fa : B}\ \textsc{App}$$

The term fa is identified with the proof (1) of the formula B.

We specify a language of terms with the following grammar:

$$\mathcal{T} ::= * \mid x \mid \lambda_x \mathcal{T} \mid \mathcal{T}\mathcal{T} \mid (\mathcal{T}, \mathcal{T}) \mid \pi(\mathcal{T}) \mid \pi'(\mathcal{T})$$

where x is a (meta-)variable.

Each formula A has infinitely many variables x such that $x : A$. Given this assumption, we adorn types with terms according to Table 3.

Thus our proof of (2) now looks like:

$$(2) \qquad \frac{(a,b) : A \wedge B \rhd (a,b) : A \wedge B}{(a,b) : A \wedge B \rhd \pi'(a,b) : B}\ \textsc{Sec}$$

and so the term $\pi'(a,b)$ is identified with the proof (2) of the formula B. Of course, the term language we have described just is that of the (pure) typed λ-calculus (see Appendix A). The identification of formulas of PIPL with types of TLC, and of intuitionistic proofs with terms of TLC, is called the Curry-Howard Isomorphism.

We wish to extend the Curry-Howard Isomorphism to classical propositional logic (CPL). Thus, we need to introduce negation into our proof system. We introduce negation by designating a single *result formula* (later *result type*). We denote the result formula by R. We now write $\neg A$ for formulas $A \to R$. Abstraction allows us to introduce negations into proofs:

$$\frac{\Gamma, x : A \rhd r : R}{\Gamma \rhd \lambda_x r : \neg A}$$

The addition of negation gives us full intuitionistic propositional logic (IPL). To achieve CPL, we need to include double negation elimination. That is, we need the rule

$$\frac{\Gamma \rhd \neg\neg A}{\Gamma \rhd A}.$$

Moreover, we need a term operation corresponding to this rule. We introduce a *control operator* C to our langauge of terms. Thus our term language is now:

$$\mathcal{T} ::= * \mid x \mid \lambda_x\mathcal{T} \mid C\mathcal{T} \mid \mathcal{T}\mathcal{T} \mid (\mathcal{T},\mathcal{T}) \mid \pi(\mathcal{T}) \mid \pi'(\mathcal{T})$$

and we have the following rule:

$$(C) \quad \frac{\Gamma \rhd a : \neg\neg A}{\Gamma \rhd Ca : A}$$

Since we have added another syntactic combination rule, we need to introduce an *evaluation strategy* for reducing terms of the form Ca. To do so properly, we need another operator $\mathcal{A}$. Thus our term language is now:

$$\mathcal{T} ::= * \mid x \mid \lambda_x\mathcal{T} \mid C\mathcal{T} \mid \mathcal{A}\mathcal{T} \mid \mathcal{T}\mathcal{T} \mid (\mathcal{T},\mathcal{T}) \mid \pi(\mathcal{T}) \mid \pi'(\mathcal{T})$$

We postpone discussion of evaluation strategies and simply present the corresponding rule:

$$(\mathcal{A}) \quad \frac{\Gamma \rhd r : R}{\Gamma \rhd \mathcal{A}r : R}$$

Using $\mathcal{A}$ and C, we have a proof system for CPL wherein formulas are identified with types and proofs are identified with terms. That is, we have extended the Curry-Howard Isomorphism to CPL. The term language is a TLC with control operators.

5.3 The Continuation Monad and CPS Transformations

A The λ-Calculus

Much of the following material is drawn from Gunter (1992). Let Σ_1 be a collection of type constants, or *basic types*. We form the *types over* Σ_1 with the context-free grammar:

$$\mathcal{S} ::= 1 \mid A \mid \mathcal{S} \to \mathcal{S} \mid \mathcal{S} \times \mathcal{S}$$

where $A \in \Sigma_1$, and 1 is the null product type. That is, the types over Σ_1 are the trees generated by this grammar.

Let Σ_0 be a function from term constants to types over Σ_1. The pair $\Sigma = (\Sigma_0, \Sigma_1)$ is called a *signature*. We form the *terms over* Σ_0 with the context-free grammar:

$$\mathcal{T} ::= * \mid a \mid x \mid \lambda_x \mathcal{T} \mid \mathcal{T}\mathcal{T} \mid (\mathcal{T}, \mathcal{T}) \mid \pi(\mathcal{T}) \mid \pi'(\mathcal{T})$$

where x is a variable and $(a, A) \in \Sigma_0$. The $*$ is a special constant of type 1. The terms over Σ_0 are the trees generated by this grammar, called *term trees*. Equivalence classes of term trees modulo $\equiv_\alpha$ are called *λ-terms*, or simply *terms*. Free variables and substitution are defined in the usual way.

Herein we use Latin minuscules as metavariables ranging over λ-terms. Let $\Sigma = (\Sigma_0, \Sigma_1)$ be a signature. A *type assignment* is a (possibly empty) list Γ of pairs $x : A$, where x is a variable and A a type, such that the variables in Γ are distinct.

A *typing judgment* is a triple consisting of a type assignment Γ, a term a, and a type A such that all of the free variables of a occur in Γ. Let $\mathcal{A}$ be the collection of all type assignments, and let Λ_0 be the collection of all λ-terms and Λ_1 be the collection of all types. Let

$$\mathcal{J} = \{(\Gamma, a, A) \in \mathcal{A} \times \Lambda_0 \times \Lambda_1 \mid \text{ all the free variables in } a \text{ occur in } \Gamma\}.$$

That is, $\mathcal{J}$ is the collection of all typing judgments. We define $\rhd : \subseteq \mathcal{J}$ to be the least relation closed under the axioms and rules in Table 4. We write $\Gamma \rhd a : A$ to indicate that $(\Gamma, a, A) \in \rhd :$. If Γ is empty, we write $\rhd a : A$. The typing judgments that appear above the line in each rule are called *premises*, and those below the line are called *conclusions*.

A *typing derivation* is a labelled tree where the labels are typing judgements, the leaves are axioms and each non-leaf is labelled by the conclusion of a

V-Projection	$\Gamma, x : A, \Gamma' \rhd x : A$	Pairing	$\dfrac{\Gamma \rhd a : A \qquad \Gamma \rhd b : B}{\Gamma \rhd (a, b) : A \times B}$
Constant	$\Gamma \rhd c : \Sigma_0(c)$	First	$\dfrac{\Gamma \rhd h : A \times B}{\Gamma \rhd \pi(h) : A}$
Null Product	$\Gamma \rhd * : 1$	Second	$\dfrac{\Gamma \rhd h : A \times B}{\Gamma \rhd \pi'(h) : B}$
Abstraction	$\dfrac{\Gamma, x : A \rhd b : B}{\Gamma \rhd \lambda_x b : A \to B}$	Permute	$\dfrac{\Gamma, x : B, y : C, \Gamma' \rhd a : A}{\Gamma, y : C, x : B, \Gamma' \rhd a : A}$
Application	$\dfrac{\Gamma \rhd f : A \to B \qquad \Gamma \rhd a : A}{\Gamma \rhd f(a) : B}$	Weakening	$\dfrac{\Gamma \rhd a : A \quad (x \notin \Gamma)}{\Gamma, x : B \rhd a : A}$

Table 4: Typing rules

rule whose premises are the labels of that non-leaf's daughters. A term a is of type A iff $\Gamma \rhd a : A$ is the conclusion of some typing derivation. If $\Gamma \rhd a : A$ and $\Gamma \rhd a : A'$, then $A = A'$.

An *equation* is a four-tuple (Γ, a, b, A) where Γ is a type assignment, $\Gamma \rhd a : A$ and $\Gamma \rhd b : A$. We typically write equations as $(\Gamma \rhd a = b : A)$. Let T be a set of equations. We write $T \vdash (\Gamma \rhd a = b : A)$ if $(\Gamma \rhd a = b : A) \in T$. The statement $T \vdash (\Gamma \rhd a = b : A)$ is called an *equational judgment*. We write $\vdash (\Gamma \rhd a = b : A)$ to indicate that $(\Gamma \rhd a = b : A)$. is to be included in every theory.

An *equational theory* is a set of equations closed under the axioms and rules in Table 5. The equational judgments that appear above the line in each rule are called *premises*, and those below the line are called *conclusions*. An *equational derivation* is defined exactly as a typing derivation, with equational judgements in place of typing judgements.

Definition 23. An equational theory that satisfies the rules in Table 6 is called a *λ-theory*.

Add	$\dfrac{T \vdash (\Gamma \rhd a = b : A) \quad (x : B \notin \Gamma)}{T \vdash (\Gamma, x : B \rhd a = b : A)}$
Drop	$\dfrac{T \vdash (\Gamma, x : B \rhd a = b : A) \quad (x \notin \mathrm{Fv}(a) \cup \mathrm{Fv}(b))}{T \vdash (\Gamma \rhd a = b : A)}$
Permute	$\dfrac{T \vdash (\Gamma, x : B, y : C, \Gamma' \rhd a = b : A)}{T \vdash (\Gamma, y : C, x : B, \Gamma' \rhd a = b : A)}$
Reflexivity	$\vdash (\Gamma \rhd a = a : A)$
Symmetry	$\dfrac{T \vdash (\Gamma \rhd a = b : A)}{T \vdash (\Gamma \rhd b = a : A)}$
Transitivity	$\dfrac{T \vdash (\Gamma \rhd a = b : A) \qquad T \vdash (\Gamma \rhd b = c : A)}{T \vdash (\Gamma \rhd a = c : A)}$
μ	$\dfrac{T \vdash (\Gamma \rhd a = b : A \to B) \qquad T \vdash (\Gamma \rhd c = d : A)}{T \vdash (\Gamma \rhd a(c) = b(d) : B)}$
ξ	$\dfrac{T \vdash (\Gamma, x : A \rhd a = b : B)}{T \vdash (\Gamma \rhd \lambda_x a = \lambda_x b : A \to B)}$

Table 5: Equational Rules

Null Product	$\vdash (\Gamma \rhd a = * : 1)$
β	$\vdash (\Gamma \rhd (\lambda_x b)(a) = [a/x]b : B)$
η	$\vdash (\Gamma \rhd \lambda_x f(x) = f : A \to B) \quad (x \notin \mathrm{Fv}(f))$
F-projection	$\vdash (\Gamma \rhd \pi(a, b) = a : A)$
S-projection	$\vdash (\Gamma \rhd \pi'(a, b) = b : B)$
Pair-identity	$\vdash (\Gamma \rhd (\pi(a, b), \pi'(a, b)) = (a, b) : A \times B)$

Table 6: λ-rules

B Category Theory

Definition 24 (Category Axioms)**.** A *category* **C** consists of the following:

1. A collection of *objects*, denoted by $Ob(\mathbf{C})$;
2. A collection of *arrows*, denoted by $Ar(\mathbf{C})$;
3. Operations dom and cod from the collection of arrows to the collection of objects:

   $$\mathrm{dom} : Ar(\mathbf{C}) \to Ob(\mathbf{C}) \quad \text{and} \quad \mathrm{cod} : Ar(\mathbf{C}) \to Ob(\mathbf{C}).$$

   For any arrow $f \in Ar(\mathbf{C})$, dom f is called the *domain* of f, and cod f is called the *codomain* of f. If dom $f = a$ and cod $f = b$, we represent this as

   $$f : a \to b \quad \text{or} \quad a \xrightarrow{f} b;$$

4. An operation $\circ$ that assigns to each pair $\langle g, f\rangle$ of arrows $f, g \in Ar(\mathbf{C})$ that satisfies cod f = dom g, an arrow $g \circ f$, called the *composite* of f and g, such that

   $$\mathrm{dom}\, g \circ f = \mathrm{dom}\, f, \quad \text{and} \quad \mathrm{cod}\, g \circ f = \mathrm{cod}\, g;$$

5. For all objects $b \in Ob(\mathbf{C})$, there is an arrow $\mathsf{id}_b : b \to b$ in $Ar(\mathbf{C})$, called the *identity arrow on b*, such that the following identity holds for all arrows $f : a \to b$ and $g : b \to c$ in $Ar(\mathbf{C})$:

   $$\mathsf{id}_b \circ f = f \quad \text{and} \quad g \circ \mathsf{id}_b = g;$$

6. For all arrows $f : a \to b$, $g : b \to c$, and $h : c \to d$ in $Ar(\mathbf{C})$ the following identity holds:

   $$h \circ (g \circ f) = (h \circ g) \circ f.$$

Note that $Ob(\mathbf{C})$ and $Ar(\mathbf{C})$ are not necessarily sets. In our preliminary examples, however, they will be sets in order to facilitate understanding.

We will use directed graphs to represent arrow compositions just as we A *diagram* over **C** is a directed graph with edges labelled by arrows of **C**, and nodes by objects of **C**. We say that a diagram *commutes* if for every pair of vertices A and B, and paths $f_1, \ldots, f_n$ and $g_1, \ldots, g_m$ between A and B, we have that $f_n \circ \cdots \circ f_1 = g_m \circ \cdots \circ g_1$.

To illustrate, we restate the categorical properties for identity arrows (Definition 24 (5)) and associativity of composition (Definition 24 (6)).

Property 1 (Identity Law for Arrow Composition)**.** *Let* $\mathbf{C}$ *be a category, and let* $b \in Ob(\mathbf{C})$ *with identity arrow* id_b*. For all arrows* $f : a \to b$ *and* $g : b \to c$ *in* $Ar(\mathbf{C})$*, the following diagram commutes.*

$$\begin{tikzcd}
 & b \arrow[d, "\mathsf{id}_b"] \arrow[dr, "g"] & \\
a \arrow[ur, "f"] \arrow[r, "f"] & b \arrow[r, "g"] & c
\end{tikzcd}$$

Property 2 (Associative Law of Arrow Composition)**.** *Let* $\mathbf{C}$ *be a category. For all arrows* $f : a \to b$*,* $g : b \to c$*, and* $h : c \to d$ *in* $Ar(\mathbf{C})$*, the following diagram commutes.*

$$\begin{tikzcd}
a \arrow[r, "f"] \arrow[d, "h\circ(g\circ f)=(h\circ g)\circ f"'] \arrow[dr, "g\circ f"] & b \arrow[d, "g"] \arrow[dl, "h\circ g"'] \\
d & c \arrow[l, "h"]
\end{tikzcd}$$

We can encompass all set theoretic properties of functions within category theory. We define the category **Set** as follows:

- The objects are sets.
- The arrows are functions.
- The composition operator is function composition.
- The identity arrow id_A is the identity function id_A on A.

It is easy to check that **Set** satisfies Definition 24, due to the Identity and Associative Laws for Functional Composition.

References

Amadio, R. M., & Curien, P. L. (1998). *Domains and Lambda-Calculi*. Cambridge University Press.

Gunter, C. (1992). *Semantics of Programming Languages*. The MIT Press.