Sequentiality

Dan Suciu
University of Pennsylvania

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation


This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/261
For more information, please contact libraryrepository@pobox.upenn.edu.
Sequentiability

Abstract
This report contains an overview of concrete data structures, sequential functions and sequential algorithms. We define the notions of concrete data structures and prove the representation theorem. Next, we present the notions of sequential functions and of sequential algorithms, and show how the latter can serve as a model for PCF. Finally, we present the language CDS0, designed as a syntax for concrete data structures and sequential algorithms.

Comments
Sequentiality

MS-CIS-93-44
LOGIC & COMPUTATION 63

Dan Suciu

University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

April 1993
Sequentiality

Dan Suciu\textsuperscript{1}
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
Email: suciu@saul.cis.upenn.edu

April 21, 1993

\textsuperscript{1}This report was part of the authors area exam
Abstract

This report contains an overview of concrete data structures, sequential functions and sequential algorithms. We define the notions of concrete data structures and prove the representation theorem. Next, we present the notions of sequential functions and of sequential algorithms, and show how the latter can serve as a model for PCF. Finally, we present the language CDS0, designed as a syntax for concrete data structures and sequential algorithms.
Contents

1 Introduction 2

2 Concrete Data Structures and Concrete Domains 6
  2.1 Definitions ............................................. 6
  2.2 Examples ............................................... 10
  2.3 Concrete Domains .................................... 12

3 Sequential Functions and Sequential Algorithms 18
  3.1 Sequential Functions ................................. 18
  3.2 Concrete Sequential Algorithms .................... 24
  3.3 Abstract Sequential Algorithms .................... 30
  3.4 The Category of Sequential Algorithms ............ 32
  3.5 DcDs as models for PCF .............................. 33

4 Deterministic Concrete Data Structures as a Programming Language 37
  4.1 Specifying Concrete Data Structures in CDS0 ....... 37
  4.2 Sequential Algorithms ............................... 41
  4.3 Operational Semantics ............................... 44

5 Conclusions 50
Chapter 1

Introduction

A measure for a good match between the denotational and the operational semantics of a programming language like PCF, is the relation between the two equivalence relations generated by them. Two program phrases $M$ and $N$ are said to be denotational equivalent, $M \equiv N$ iff $\mathbb{[}M\mathbb{]} = \mathbb{[}N\mathbb{]}$, i.e. iff the two have the same interpretation. $M$ and $N$ are said to be observational equivalent, $M \equiv_{op} N$ iff for any context $C[\ ]$ of a ground type (int or bool), $C[M]$ and $C[N]$ either both diverge, or both converge to the same value. A minimal requirement for a “good match” is the adequacy property:

**Adequacy** $M \equiv N \rightarrow M \equiv_{op} N$.

This requirement is “minimal” in the sense that a semantics which identifies programs with different behavior is general considered to be of no use: the trivial semantics, which assigns to all program phrases the same value, is an example of such an useless semantics.

The converse implication, when it holds, is called full abstraction:

**Full Abstraction** $M \equiv_{op} N \rightarrow M \equiv N$.

Failure of this property is not generally considered a catastrophe. It just means that we might consider different two program phrases $M$ and $N$, based on the denotational semantics, although they happen to behave the same, in the operational semantics.

In the classical paper [11], Plotkin shows that the semantics of bounded complete, $\omega$-algebraic cpos, for the programming language PCF, is adequate
but not fully abstract. The failure of full abstraction is shown to be connected to the \textit{parallel-or} function, which evaluates to \textit{true} if any of its two arguments evaluates to \textit{true}, even if the other argument diverges. Once a parallel-or construction is added to PCF, the same semantics becomes fully abstract. The problem is that PCF has a \textit{sequential} evaluator, while the model contains functions, like the parallel-or, which are \textit{not} sequential: however, we may write in PCF program phrases of a higher type, which can be distinguished only when applied to the parallel-or function. So they \textit{can} be distinguished in the model - where the parallel-or is present - but they cannot be distinguished by the evaluator, because parallel-or is not expressible in PCF.

Although [10] proves that a full abstract model of PCF without parallel-or exists and that it is unique, the construction is purely syntactic. The challenge of a natural description of that model remains an open problem, despite serious efforts made over the years. The right direction to search seems to consist in describing a \textit{submodel} of the bounded complete, $\omega$-algebraic cpo frame that eliminates all “non sequential” function, as the parallel-or, although no generally accepted definition exists for sequentiality for continuous functions between such domains. Efforts in this direction lead Berry to introduce the \textit{stable} functions, which indeed exclude parallel-or, but which turned out to allow for other, more complicated, “non parallel” functions (see section 3.1). However, stable functions were perceived as a step in the right direction, towards a definition of sequentiality.

Efforts to find a good definition of sequentiality were made also independently of a search for a full abstract model for PCF. [12] gives a good explanation of the difficulties for such a definition. The simple case is that of a first order function, between flat domains, ex. $f(x_1, \ldots, x_n)$, $f : N_1 \times \ldots \times N_1 \rightarrow N_1$; $f$ is called \textit{sequential} iff $f$ is either a constant, or there exists an index $i$ such that $f$ is strict in $x_i$, and for any $x_i \in N_1$, the function $f_i(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = f(x_1, \ldots, x_n)$ is sequential. The problem arises at higher types, when a natural definition is not so obvious: [12] shows how sequentiality can be defined at higher types such that sequential functions are closed under composition. However, for this construction to work, it is necessary for the base domains to be flat, and the definition itself works only for functions with several arguments. A good definition of sequentiality for a function $f : D \rightarrow D'$ doesn’t seem possible, with this technique.

The search for a good definition of sequentiality was also inspired by the study of the syntactic properties of the evaluator for PCF, which is a
sequential process; see [5] and [3] for a review of these properties. As a consequence of the sequential nature of the evaluator, the following property was identified. Consider PCF extended with a new constant $\Omega$ at each type, for the “undefined” term, and define $M \leq_\Omega N$ iff $N$ can be obtained from $M$ by substituting some of the $\Omega$’s with arbitrary terms. Then it is the case that, if $M$ is a program which diverges, then either $\forall N. M \leq_\Omega N \Rightarrow N \uparrow$ ($N$ diverges), or, there exists an “occurrence” $w$ of $\Omega$ in the syntactic tree of $M$ such that $\forall N. M \leq_\Omega N, N \downarrow \Rightarrow N$ has at the position $w$ something different from $\Omega$.

The notion of sequentiality for functions $f : D \rightarrow D'$ and the definition of concrete domains, for which sequentiality can be defined, and the connection between concrete domains and concrete data structures is attributed to Kahn and Plotkin. Curien and Berry defined the notion of “sequential algorithm”, showed that they form the right construction for exponentiation, and defined a programming language for concrete data structures. They tried unsuccessfully to derive a fully abstract model for PCF based on sequential algorithms. Recently, Cartwright and Felleisen in [4] discovered that a natural extension of PCF with a construction called *catch* allows for a naturally defined fully abstract model, which seems to be strongly related to concrete data structures and sequential algorithms; indeed, Curien shows in [6] that concrete data structures and sequential algorithms are a fully abstract model of PCF+catch.

This report is structured in three chapters. In chapter 2, we define the notions of concrete data structures and concrete domains, and state the relation between them. In chapter 3, we define sequential functions and the two notions of sequential algorithms, and prove that the concrete data structures and sequential algorithms form a cartesian closed category. In chapter 4, we present the programming language DCDS0 and its operational semantics.

We shall use the following domain-theoretic notions, which can be found, for example, in [7]. Here is a very short review:

A dcpo (directed complete partial order) is a partial order $(D, \leq)$ such that, any directed set $X \subseteq D$ (i.e. for which $x, y \in X \Rightarrow \exists z \in X$ such that $x \leq z$ and $y \leq z$) has a least upper bound, $\forall X$. For the rest of this short review, let $(D, \leq)$ be some dcpo.

$x_0 \in D$ is called a finite (or compact) element, if $x \leq \forall X$ with $X \subseteq D$ directed, implies $\exists y \in X$ s.t. $x \leq y$. 

4
\((D, \leq)\) is called \textbf{algebraic}, if \(\forall y \in D, \{x \mid x \leq y, x \text{ finite}\}\) is directed and has \(y\) as lub.

\((D, \leq)\) is \textbf{\(\omega\)-algebraic}, if it is algebraic and the set of its finite elements is at most denumerable.

\((D, \leq)\) is \textbf{consistently complete (or bounded complete)}, if any \textbf{bounded} set \(X \subseteq D\) (i.e. for which there is some \(z \in D\) s.t. \(\forall x \in X, x \leq z\)) has a lub, \(\bigvee X\). Equivalently, \((D, \leq)\) is consistently complete, if any nonempty set \(X\) has a glb, \(\bigwedge X\).

\(x, y \in D\) are called \textbf{consistent}, and we write \(x \uparrow y\), if \(\exists z \in D, x \leq z\) and \(y \leq z\).

\((D, \leq)\) is \textbf{coherent}, if \(\forall X \subseteq D\) s.t. \(\forall x, y \in X, x \uparrow y, X\) has a lub, \(\bigvee X\). Equivalently, \((D, \leq)\) is coherent if \(\forall x, y \in D, x \uparrow y \Rightarrow \exists x \sqcup y\) (the fact that \((D, \leq)\) is a dcpo is used here).

Any coherent dcpo is obviously also consistently complete. The converse is not true. Eg., \(\langle\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}\}, \subseteq\rangle\) is a consistently complete cpo, but not coherent.
Chapter 2

Concrete Data Structures and Concrete Domains

2.1 Definitions

Concrete data structures were introduced first by Kahn and Plotkin, as an application of domain theory for some simple data structures. The concepts are rather simple and self explanatory, so we postpone giving examples until the next section.

Definition 1 A concrete data structure, or cds is \( M = (C, V, E, \triangleright) \)
where:

a \( C \) is a denumerable set of cells

b \( V \) is a denumerable set of values

c \( E \subseteq C \times V \) is a set of events

d \( \triangleright \subseteq P_{\text{fin}}(E) \times C \) is an enabling relation between finite sets of events and cells. We write \( \triangleright c \) for \((\phi, c) \in \triangleright\), and call \( c \) an initial state.

Definition 2 Let \( x \subseteq E \) be a set of events. Define the set of cells enabled in \( x \), \( E(x) \), to be the smallest set closed under the rule:

\[
\begin{align*}
&c_1, \ldots, c_n \in E(x) \\
&(c_1, v_1), \ldots, (c_n, v_n) \in x \\
&\{(c_1, v_1), \ldots, (c_n, v_n)\} \triangleright c \\
&c \in E(x)
\end{align*}
\]
We also define $F(x)$ to be the set of filled cells in $x$, $F(x) = \{c \mid \exists v \in V, (c, v) \in x\}$, and $A(x)$ to be the set of accessible cells, $A(x) = E(x) - F(x)$.

**Remark** Clearly $x \subseteq x' \Rightarrow E(x) \subseteq E(x')$ and $F(x) \subseteq F(x')$. $\square$

**Definition 3** A state $x$ of $M$ is a set of events ($x \subseteq E$), such that:

**consistence** Any cell holds at most one value, i.e. $(c, v) \in x, (c, v') \in x \Rightarrow v = v'$.

**safety** Any cell $c$ filled in $x$ is enabled in $x$, i.e. $F(x) \subseteq E(x)$

**Example.** The cds $LIST = (C, V, E, \rightarrow)$, where $C = (c_0, c_1, c_2, \ldots)$ is a countable collection of cells, $V = N \times \{\cdot, \rightarrow\}$ (i.e. each cell may be filled with an integer and with a pointer to the next cell or nil), $E = C \times V$ (any cell may be filled with any value), $\rightarrow c_0, \{(c_n, (k, \rightarrow))) \vdash c_{n+1}$ (i.e. a cell may be filled only when the previous one is filled with a pointer). Then a state $x$ is a set of cells filled with values and pointers, such that whenever a cell $c_{i+1}$ is filled in $x$, the previous cell $c_i$ is also filled and with a non null pointer. An infinite list is also a valid state, if all pointers are non null. It is not necessary for a finite list to end with nil.

Let $(D(M), \leq)$ be the set of states of the cds $M$, ordered by set inclusion. We say that a cds $M$ generates the order structure $(D(M), \leq)$.

For the beginning, we are primarily interested in the structure of $(D(M), \leq)$.

**Proposition 1** $(D(M), \leq)$ is a $\omega$-algebraic coherent (and hence, consistently complete) dcpo, in which suprema of directed or pairwise compatible sets are unions.

**Proof** Clearly, for any $X \subseteq D(M)$, $F(\bigcup X) = \bigcup F(X)$. Then:

1. $(D(M), \leq)$ is a dcpo. Indeed, let $X \subseteq D(M)$ be a directed set of states. Then $\bigcup X$ clearly satisfies the consistence condition. More, $F(\bigcup X) = \bigcup F(X) \subseteq \bigcup E(X) \subseteq E(\bigcup X)$, so $\bigcup X$ is safe.

2. $(D(M), \leq)$ is $\omega$-algebraic and has the finite states as compact elements. This statement is obvious.
3. $(D(M), \leq)$ is coherent, i.e. if $X \subseteq D(M)$ of pairwise consistent states, then $\bigcup X \in D(M)$. The proof is as in 1.

However, not any $\omega$-algebraic coherent dcpo is the set of states of some cds $M$. A complete characterization, due to Kahn and Plotkin, of the domains associated with cdss's, is given in section 2.3.

**Definition 4** A cds $x$ is well founded if the order relation $c < c'$ generated on the set of cells by: $\{(c_1, v_1), (c_2, v_2), \ldots, (c_{n-1}, v_{n-1}), (c, v)\} \vdash c' \Rightarrow c < c'$ is well founded.

The reason for considering well founded cdss's is that the safety condition can be stated in a simpler form:

**Proposition 2** In a well founded cds, the safety condition for a set of events $x$ is equivalent to the following condition:

- Any cell $c$ filled in $x$ has an enabling in $x$, i.e. there are events $(c_1, v_1), \ldots, (c_n, v_n) \in x$ such that $\{(c_1, v_1), \ldots, (c_n, v_n)\} \vdash c$.

**Proof** $\Longrightarrow$. If $c$ is filled in $x$, then $c \in E(x)$, so, by the definition of $E(x)$, there are $(c_1, v_1), \ldots, (c_n, v_n) \in x$ such that $\{(c_1, v_1), \ldots, (c_n, v_n)\} \vdash c$.

$\Longleftarrow$. This is done by the same argument as in Konig's lemma: suppose there is a cell $c$, filled in $x$, $c \notin E(x)$. $c$ has an enabling $\{(c_1, v_1), \ldots, (c_n, v_n)\} \vdash c$ in $x$ (i.e. $(c_1, v_1), \ldots, (c_n, v_n) \in x$). There must be at least one cell $c_i$ not included in $E(x)$ (or else we would have $c \in E(x)$), and $c_i < c$. Continuing, we find an infinite descending chain of cells.

In this report, we have chosen to state all properties and make all proofs for arbitrary cdss, without relying on well-foundness. That this is indeed possible is stated in [5] and [2].

**Definition 5** A cds $M$ is called stable or deterministic, if for any state $x$ and any cell $0 \vdash c$, $x'_0 \vdash c$ and $x'_0 \subseteq x$, then $x_0 = x'_0$. A deterministic cds is abbreviated dcds.
So any state of a dcqs contains at most one enabling for any cell c. It
seems that considering a stronger version of cds’s M, in which any cell c has
at most one enabling (i.e. \(x_0 \vdash c, x'_0 \vdash c \Rightarrow x_0 = x'_0\)) is sufficient, but too
restrictive for practical purposes. In the next subsection we shall give an
example of a dcqs for which this condition fails.

The following proposition, and especially its corollary, gives the main
motivation for considering deterministic cds’s.

**Proposition 3** In a dcqs M, if \(X \subseteq D(M)\) is a nonempty, upperbounded
collection of states (i.e. \(\exists z \in D(M)\) s.t. \(\forall x \in X, x \leq z\)), then \(\bigcap X\) is also
a state (\(\bigcap X \in D(M)\)).

**Proof** The consistency condition \(((c, v), (c, v')) \in \bigcap X \Rightarrow v = v')\) is triv-
ially satisfied by \(\bigcap X\), even for nondeterministic cds’s. For the safety con-
dition, we have to show that \(c \in F(\bigcap X), \Rightarrow c \in E(\bigcap X)\). But \(F(\bigcap X) \subseteq
\bigcap_{x \in X} F(x)\) (because \(X\) is upperbounded, they are even equal), so we may
suppose \(\forall x \in X, c \in F(x)\). Then \(\forall x \in X, c \in E(x)\), so \(c \in \bigcap_{x \in X} E(x)\), and
the proof is concluded by the following lemma. □

**Lemma 1** If M is a dcqs and X is a nonempty, upperbounded set of states,
then \(E(\bigcap X) = \bigcap_{x \in X} E(x)\).

**Proof** For the nontrivial inclusion, \(\bigcap_{x \in X} E(x) \subseteq E(\bigcap X)\), suppose \(c \in
\bigcap_{x \in X} E(x)\). Let \(x \in X\); as \(c \in E(x)\), there are \((c_1, v_1), \ldots, (c_n, v_n) \in
x\) such that \(\{(c_1, v_1), \ldots, (c_n, v_n)\} \vdash c\). But X is upperbounded, say by
the state z, so, for any other \(x' \in X, c\) is enabled in \(x'\) by the same set
\(\{(c_1, v_1), \ldots, (c_n, v_n)\}\) (else c would be enabled more than once in z). By
induction on the length of the proof of \(c \in E(x)\), we may suppose that
\(c_1, \ldots, c_n \in E(\bigcap X)\), so \(c \in E(\bigcap X)\). □

**Corollary 1** If M is a dcqs, then \((D(M), \leq)\) is a distributive domain.

**Definition 6** A cds is called **filiform** if: \(x_0 \vdash c \Rightarrow |x_0| \leq 1\). A cds is called
**sequential** if, for any state x and any cell c' such that:

\[c' \not\in F(x) \text{ and } \exists z \geq x \ c' \in F(z)\]

there exists a cell c \(\in A(x)\) such that \(\forall y \geq x \ c' \in F(y) \Rightarrow c \in F(y)\). Such
a cell c is called a **sequentiality index** of M for c' at x.
The following proposition is from [5]

**Proposition 4** A filiform dcds is sequential.

**Proof** Let \( z \geq x \) and \( c' \not\in F(x) \), \( c' \in F(z) \). From the safety condition, we conclude \( c' \in E(z) \), so, there exists a (unique) sequence \( (c_1, v_1), \ldots, (c_n, v_n) \in z \), such that \( \vdash c_1, (c_{i-1}, v_{i-1}) \vdash c_i \), \( i = 2, \ldots, n \) and \( c_n = c' \). Let \( k \) be the smallest index for which \( c_k \not\in F(x) \); clearly, \( \forall i > k, c_i \not\in F(x) \) (else \( c_i \) would have multiple enablings in \( z \) for the smallest such \( i \)). We shall prove that \( c_k \) is the sequentiality index for \( c' = c_n \) at \( x \). W.l.o.g., we may suppose that \( c_k \) is the sequentiality index for all the cells \( c_{k+1}, \ldots, c_{n-1} \). So let \( y \geq x \), be such that \( c' \in F(y) \). As above, there are \( (c'_1, v'_1), \ldots, (c'_{n'}, v'_{n'}) \in y \), such that \( \vdash c'_1 \) and \( (c'_{i-1}, v'_{i-1}) \vdash c'_i \), \( i = 2, \ldots, n' \), \( c'_{n'} = c' \). Also, there is an index \( k' \) which is the smallest for which \( c'_{k'} \not\in F(y) \). If \( c_k \) occurs in \( c'_1, \ldots, c'_{n'-1} \), we are done. Else, the state \( y = x \cup \{(c_1, v_1), \ldots, (c_n, v_n), (c'_1, v'_1), \ldots, (c'_{n'-1}, v'_{n'-1})\} \) cannot have any of \( c_k, \ldots, c_{n-1} \) filled, because it doesn’t contain \( c_k \) which is their sequentiality index. This proves that the set \( \{(c_1, v_1), \ldots, (c_n, v_n), (c'_1, v'_1), \ldots, (c'_{n'-1}, v'_{n'-1})\} \) is a state (it is consistent because the cells are either in \( x \), or distinct), which contains two enablings of \( c' (= c_n = c'_{n'}) \), contradicting the hypothesis that \( M \) is sequential. \( \square \)

### 2.2 Examples

All examples given in this section are deterministic cds’s. For the rest of this report, we will write \( (c_1, v_1), \ldots, (c_n, v_n) \vdash c \) instead of \( \{(c_1, v_1), \ldots, (c_n, v_n)\} \vdash c \).

1. \( O = (\{c\}, \{T\}, \{(c,T)\}, \vdash) \), with \( \vdash c \) generates the Sierpinsky space \( D(O) = \{\{\}, \{(c,T)\}\} \).

2. \( T = (\{c\}, \{t, f\}, \{(c, t), (c, f)\}, \vdash) \) with \( \vdash c \), generates the lifted set of booleans \( D(T) = \{t, f\}_\perp \).

3. \( N = (\{c\}, \{0, 1, 2, \ldots\}, \{(c, 0), (c, 1), \ldots\}, \vdash) \) with \( \vdash c \), generates the flat domain of integers \( N_\perp \).

4. Let \( Var \) be a set of variables. \( LAMBDA = (C, V, E, \vdash) \) where \( C = \{0, 1\}^* \), \( V = \{\cdot\} \cup \{\lambda x. x \in Var\} \cup Var \), \( E = C \times V \) and \( \vdash \epsilon \) (the
empty word), \((w, \lambda x.) \vdash w0, (w, \cdot) \vdash w0, (w, \cdot) \vdash w1\). The domain associated with \textit{LAMBDA} is related to the \(\lambda\)-terms in the following way: consider \(\Lambda(\Omega)\) the set of \(\lambda\)-terms extended with a symbol for the “undefined” term, \(\Omega\), i.e. the terms are defined by:

\[
\text{term ::= } x \ (x \in \text{Var}) \mid \lambda x.\text{term} \mid (\text{term}.\text{term}) \mid \Omega
\]

Define the syntactic ordering \(\leq_\Omega\) on \(\Lambda(\Omega)\) to be the reflexive and transitive closure of the relation: \(C[\Omega] \leq_\Omega C[t]\), for any context \(C[\ ]\) and any term \(t\). Then \((D(\text{LAMBDA}), \leq)\) is the ideal completion of \((\Lambda(\Omega), \leq_\Omega)\).

The cells in \textit{LAMBDA} stand for occurrences in a \(\lambda\)-term.

5. The typed lambda terms can be obtained in a similar, but more complicated way, as dcds. Let \(B\) be a set of base types, let \textit{types} be the set of types (i.e. contains \(B\) and is closed under \(\rightarrow\)), and \(\text{Var}\) be a family of sets of variables, indexed by \textit{types} (i.e. any variable \(x^\tau\) has a fixed type \(\tau\)). Then the cells of a typed \(\lambda\)-term have to identify both an occurrence of a subterm, and the type of that subterm. So instead of \(\{0,1\}^*\), we shall have the cells labeled by \(\{0,1\}^* \times \textit{types}\). For \((w, \sigma) \in \{0,1\}^* \times \textit{types}\), we shall write \(w^\sigma\). In order to uniquely identify the type \(\tau\) of the root, we introduce a new cell, \(c\), which is filled with \(\tau\).

Then let \textit{TLAMBDA} = \((C, V, E, \vdash)\) where:

\[
\begin{align*}
C &= \{c\} \cup \{\{0,1\}^* \times \textit{types}\} \\
V &= \textit{types} \cup \{\cdot_{\sigma\tau} / \sigma, \tau \in \textit{types}\} \cup \\
&\quad \{\lambda x^\sigma. / x^\sigma \in \textit{Var}\} \cup \textit{Var} \\
E &= \{\{c\} \times \textit{types}\} \cup \{(w^\tau, \cdot_{\sigma\tau}) / w \in \{0,1\}^*, \sigma, \tau \in \textit{types}\} \cup \\
&\quad \{(w^{\sigma\rightarrow\tau}, \lambda x^{\sigma\cdot}) / w \in \{0,1\}^*, \sigma, \tau \in \textit{types}, x^{\sigma} \in \textit{Var}\} \cup \\
&\quad \{(w^\sigma, x^\sigma) / w \in \{0,1\}^*, \sigma \in \textit{types}, x^\sigma \in \textit{Var}\}
\end{align*}
\]

and \(\vdash c, (c, \tau) \vdash e^\tau, (w^{\sigma\rightarrow\tau}, \lambda x^{\sigma\cdot}) \vdash w0^\tau, (w^\tau, \cdot_{\sigma\tau}) \vdash w0^{\sigma\rightarrow\tau}, (w^\tau, \cdot_{\sigma\tau}) \vdash w1^\sigma\).
6. This is an example of a dcds which does not have the property: \( x_0, x'_0 \in E, x_0 \vdash c, x'_0 \vdash c \Rightarrow x_0 = x'_0 \). Let \( M = (V, C, E, \vdash) \), where: \( V = \{0, 1\} \), \( C = \{(i, j) \mid i, j \in N, i \geq j\} \), \( E = C \times V \) and \( (i, j), (i, j), 0 \vdash (i + 1, j), ((i, j), 1) \vdash (i, j + 1) \), where \( i > j \). Then \( (D(M), \leq) \) is isomorphic to the domain of \textit{tapes}, which is the ideal completion of \( \{0, 1\}^* \) with the prefix ordering.

### 2.3 Concrete Domains

We shall give the complete characterization of the domains \((D(M), \leq)\) associated to cds's and dcds's. We follow [5], which cites Winskel as the author of the simplified proof of Kahn and Plotkin's representation theorem.

**Definition 7** An \textit{event structure} is \((E, \# \vdash)\), where: \( E \) is a countable set of \textit{events}, \( \# \) is a symmetric relation on \( E \) called the \textit{conflict relation}, and \( \vdash \subseteq \mathcal{P}_{\text{fin}}(E) \times E \) is the \textit{enabling relation}.

For a given subset of events \( x \in E \), we define the set of events \textit{enabled} by \( x \) to be the smallest set \( E(x) \) closed under the rule:

\[
\frac{e_1, \ldots, e_n \in E(x) \cap x \quad \{e_1, \ldots, e_n\} \vdash e}{e \in E(x)}
\]

**Definition 8** A \textit{state} of an event structure \((E, \# \vdash)\) is a set \( x \in E \) satisfying the following conditions:

- **consistence** \( e_1 \in x, e_2 \in x \Rightarrow \neg (e_1 \# e_2) \).
- **safety** \( x \subseteq E(x) \)

The set of states of \( E \), ordered by inclusion, is denoted by \((D(E), \leq)\).

Intuitively, the event structures are related to cds's: the main difference is that event structures don't have cells. The next definition and proposition state precisely how these two are connected.

**Definition 9** An \textit{event structure} \( E = (E, \# \vdash) \) has \textit{property (E)} iff the following conditions hold:
(E1) The reflexive closure of $\#$ is transitive

(E2) $\forall x_0, e_1, e_2, \text{ if } x_0 \vdash e_1 \text{ and } e_1 \# e_2, \text{ then } x_0 \vdash e_2$

The intuition behind (E1) is that, for some cds $(V, C, E, \vdash)$, the relation $\#$ is: $(c_1, v_1) \# (c_2, v_2) \iff c_1 = c_2$ and $v_1 \neq v_2$. Then $\#$ itself is not transitive closed, because $(c, v) \# (c, w)$ and $(c, w) \# (c, v)$ when $v \neq w$, but it is not the case that $(c, v) \# (c, v)$. But the reflexive closure of $\#$ is transitive closed.

Proposition 5

1. Let $M = (C, V, E, \vdash)$ be a cds. Define the relation $\#$ on $E$ by:

$$e_1 \# e_2 \iff \exists c, v_1, v_2, \quad e_1 = (c, v_1), e_2 = (c, v_2), v_1 \neq v_2$$

and define $\vdash_E$ by: $x_0 \vdash_E (c, v)$ iff $x_0 \vdash c$. Then $(E, \#, \vdash_E)$ is an event structure having property (E), and $(D(M), \leq) = (D(E), \leq)$.

2. Let $E = (E, \#, \vdash)$ be an event structure having property (E). Let $M = (C_M, V_M, E_M, \vdash_M)$, where: $V_M = E$, $C_M = \{[e]/e \in E\}$ (where $[e]$ is the equivalence class of $e$, under the reflexive closure of $\#$), $E_M = \{([e], e)/ e \in E\}$ and $\{(e_1, e_1), \ldots, ([e_n], e_n)\} \vdash_M ([e], e)$ iff $\{e_1, \ldots, e_n\} \vdash e$. Then $M$ is an event structure, and $(D(M), \leq)$ is isomorphic to $(D(E), \leq)$.

Proof Is trivial $\square$

Winskel’s idea for simplifying the proof of the representation theorem was to first characterize the domains associated to event structures, and then to further identify those associated to cds’s (which are essentially event structures with property (E)). We need some definitions.

Definition 10 Let $(D, \leq)$ be a partial order.

- We say that $x$ is covered by $y$ ($x \prec y$), iff $x < y$ and $\forall z \in D, \ x \leq z \ \text{and} \ z \leq y \ \Rightarrow \ x = z$ or $z = y$.

- $x, y$ are consistent (or compatible) iff $\exists z, \ x \leq z \ \text{and} \ y \leq z$. We write $x \uparrow y$. Otherwise, $x, y$ are inconsistent and we write $x \# y$. 

13
A covering chain (or simply chain) from $x$ to $y$ is a sequence $x = x_0 \prec x_1 \prec \ldots \prec x_n = y$.

A prime interval is a pair $[x, x']$ such that $x, x'$ are isolated (compact) and $x \prec x'$.

$[x, x'] \prec [y, y']$ iff $x \prec y$, $y \neq x'$ and $x' \prec y'$.

We write $\succ$ for the equivalence relation on prime intervals generated by the relation $\prec$.

When $(D, \leq)$ is the set of states of some cds $M$, then $x \prec x'$ means that $x'$ has exactly one more event than $x$: $x' = x \cup \{(c, v)\}$. We write $x \triangleleft c x'$ in that case. $[x, x'] \prec [y, y']$ implies that $x' - x = y' - y$, which in turn implies that $x \triangleleft c x'$ and $y \triangleleft c y'$ for the same cell $c$.

The idea behind the representation theorem is to recover events (and later cells) from prime interval equivalence classes. For this to work, the domain must satisfy certain conditions.

**Definition 11** An event domain is an $\omega$-algebraic cpo $(D, \leq)$ satisfying the following axioms (where $x, x', x'', y, z \ldots$ are isolated):

1. **I (called F in [5])** $(x) \downarrow$ is finite.
2. **C** If $x \prec y, x \prec z, y \neq z, y \uparrow z$, then $y \vee z$ exists and $y \prec y \vee z$ and $z \prec y \vee z$.
3. **R** $[x, y] \succ [x, z] \Rightarrow y = z$.
4. **V** If $[x, x'] \succ [y, y'], [x, x''] \succ [y, y']$ and $x' \uparrow x''$ then $y' \uparrow y''$.

As usual, we shall write $K(D)$ for the set of isolated (compact) elements of $(D, \leq)$.

**Definition 12** Let $(D, \leq)$ be an event domain.

1. The set of events associated with $(D, \leq)$ is $E(D) = \{[x, y]_{\succ} / x, y \in K(D)\}$, i.e. the set of equivalence classes (under the $\succ$ relation) of primes intervals.
2. The incompatibility relation \( \parallel \) on \( E(D) \) is defined by:

\[ z \parallel z', z \parallel z'', z \parallel z'' \Rightarrow [z, z'] \not\parallel [z, z''] \]

3. The enabling set of \( x \in K(D) \) is

\[ s(x) = \{ [x_i, x_{i+1}] \not\parallel / \bot = x_0 < x_1 < \ldots < x_n = x \} \]

(see the next lemma).

**Lemma 2** The set \( s(x) \) in the above definition is well defined (i.e. does not depend on the particular choice of the chain \( \bot = x_0 < x_1 < \ldots < x_n = x \)).

**Proof** (Sketch) The proof in [5] proceeds by the following steps, which we shortly enumerate (all elements are supposed to be isolated, i.e. compacts):

1. We show that if \( x \leq y \Rightarrow \) there exists a chain \( x = x_0 < x_1 < \ldots x_n = y \). This is shown by induction on \( n = |\{z \mid z \leq x\}| \), which is finite by property (I).

2. We show that, if \( x = x_0 < x_1 < \ldots x_n = y \) and \( x = y_0 < x_1 < \ldots y_m = y \) are two chains from \( x \) to \( y \), then for any equivalence class \( e \) of prime intervals (under the relation \( \parallel \)), the number of occurrences of \( e \) in the two chains is the same, i.e. \( |\{i \mid /[x_i, x_{i+1}] \not\parallel = e\}| = |\{j \mid /[y_j, y_{j+1}] \not\parallel = e\}| \). In particular, the two chains have the same length.

This is done by induction on \( n \). Two cases are considered: \( x_1 = y_1 \) (which is trivial), and \( x_1 \neq y_1 \): then, axiom (C) is used to show that \( x_1 \lor y_1 \) exists, and, by the previous fact, there is a chain from \( x_1 \lor y_1 \) to \( y \). Induction hypothesis is applied for the chains \( x_1 < x_2 < \ldots < y \) and \( x_1 < x_1 \lor y_1 < \ldots y \), and we conclude that the two chains have the same length. This allows us to use the induction hypothesis to for \( y_1 < x_1 \lor y_1 < \ldots y \) and \( y_1 < \ldots y \).

\( \square \)

**Theorem 1** (G. Winskel)
1. Let \((E, \ll, \vdash)\) be an event structure. Then \((D(E), \leq)\) is an event domain.

2. Let \((D, \leq)\) be an event domain. Then \(E = (E(D), \ll, \vdash)\), with \(\vdash\) given by \(s(x) \vdash [x, x'] \Rightarrow\), is an event structure and its associated domain is isomorphic with \((D, \leq)\).

Proof (Sketch)

1. The proof that \((D(E), \leq)\) is an \(\omega\)-algebraic domain is similar to that of Proposition 1: the isolated (compact) elements are the finite states. Checking the axioms \((I), (C), (R)\) and \((V)\) is routine.

2. This part is highly technical, and we omit it. It can be found in [5].

Note, in particular, that an event domain is a consistently (bounded) complete domain. Actually this has to be proven as a part of Winskel’s theorem. (The consistently completeness property is equivalent to: \(\forall x, y \in K(D), x \uparrow y \Rightarrow \exists (x \lor y)\). See, e.g. [7]).

Definition 13 A concrete domain is an event domain satisfying the supplementary axiom:

\(Q\) If \(z, x, y\) are isolated elements and \(z \ll x, z < y\) and \(x \ll y\), then there exists a unique \(x'\) such that \(z \ll x' \leq y\) and \(x \ll x'\).

Note that for an \(\omega\)-algebraic domain, the property \((V)\) can be derived from the other conditions\((I, C, R\) and \(Q)\) (see [5]).

Theorem 2 1. Let \(M = (V, C, E, \vdash)\) be a cds. Then the domain associated with \(M\), \((C(M), \leq)\) is a concrete domain.

2. Let \((D, \leq)\) be a concrete domain. Then there exists a cds \(M\) having its associated domain \((D(M), \leq)\) isomorphic to \((D, \leq)\).

Proof The proof is lengthy, and is omitted (it can be found in [5]). The idea is to show that the event structure associated to \((D, \leq)\) has the property \((E)\), which implies the existence of a cds having the same associated domain.

Recall the definition of distributive domains:
**Definition 14** A consistently complete domain \((D, \leq)\) is **distributive** if and only if the following property holds:

\[ x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z). \]

**Theorem 3**

1. Let \(M = (V, C, E, \vdash)\) be a dcds. Then the domain associated with \(M\), \((C(M), \leq)\) is a distributive concrete domain.

2. Let \((D, \leq)\) be a distributive concrete domain. Then there exists a dcds \(M\) having its associated domain \((D(M), \leq)\) isomorphic to \((D, \leq)\).

Again, the lengthy proof is omitted, and can be found in [5]. Actually, the result is even stronger. In the deterministic cds associated, by the proof of this theorem, to a distributive concrete domain, every cell has exactly one enabling, which is even more than being deterministic.

As an example of a nondeterministic cds with a nondistributive domain associated to it, consider \(M = (V, C, E, \vdash)\) with \(V = \{\ast\}\) (one value, so we omit it in the enabling relation), \(C = \{c_1, c_2, d, e\}, \vdash c_1, \vdash c_2, c_1 \vdash d, c_1, d \vdash e, c_2 \vdash e, c_2, e \vdash d\). Then \(D(M) = \{\emptyset, \{c_1\}, \{c_2\}, \{c_1, c_2\}, \{c_1, d\}, \{c_2, e\}, \{c_1, c_2, d\}, \{c_1, c_2, e\}, \{c_1, d, e\}, \{c_2, d, e\}, \{c_1, c_2, d, e\}\}\). This is a lattice (in which the greatest lower bound is not intersection), and contains the sublattice \(\{\emptyset, \{c_2\}, \{c_1, d, e\}, \{c_2, d, e\}, \{c_1, c_2, d, e\}\}\), which is the well known nondistributive, five elements lattice.
Chapter 3

Sequential Functions and Sequential Algorithms

3.1 Sequential Functions

Stable functions have been introduced primarily for modeling sequentiality: they seemed suitable for this purpose, because parallel-or is not a stable function. But, unfortunately, there are other stable functions which are not sequential. Yet, the notion of stable functions is an important concept for understanding sequentiality in the framework of sequential algorithms.

Proposition 6 Let $D, E$ be consistently complete dcpo’s, and $f : D \to E$ a continuous function. Consider the following properties:

S1 $\forall X \subseteq D$, $X \uparrow$ (i.e. $X$ is bounded) $\Rightarrow f(\bigwedge X) = \bigwedge f(X)$

S2 $\forall z \in D$, $\forall y \in E$, $y \leq f(z) \Rightarrow \exists m_{f,z}(y) \in D$ such that $\forall x \leq z$, $(y \leq f(x) \Leftrightarrow m_{f,z}(y) \leq x)$.

S3 $\forall x, y \in D$, $x \uparrow y$ $\Rightarrow f(x \wedge y) = f(x) \wedge f(y)$.

S4 $\forall b \in K(E)$ $\Rightarrow$ the minimal elements of $f^{-1}(b \uparrow)$ are inconsistent.

Then $S1 \iff S2 \Rightarrow S3 \Rightarrow S4$. More, if both $D$ and $E$ are algebraic and $D$ satisfies property I (see definition 11), then $S4 \Rightarrow S2$, thus all four are equivalent.
Proof

1. S1 \Leftrightarrow S2. This is classical: consider the restriction of \( f \) to \( f_z : (z) \downarrow \rightarrow (f(z)) \downarrow \). Then S2 says that \( f_z \) has a left adjoint (i.e. can be completed to a Galois connection), while S1 says that it preserves arbitrary greatest lower bounds (i.e. as a functor, \( f \) preserves products). So suppose for any \( z \), \( f_z \) has the left adjoint \( (m_f, z(y), \forall y \leq f(z)) \). Let \( X \) be bounded by \( z \), and let \( y = \bigwedge f(X) \). Then:

\[
\forall x \in X \ y \leq f(x) \Rightarrow \\
\forall x \in X \ m_f, z(y) \leq x \Rightarrow \\
m_f, z(y) \leq \bigwedge X \Rightarrow \\
y \leq f(\bigwedge X)
\]

For the other direction, suppose for any \( z \), \( f_z \) preserves arbitrary glb’s. Define \( m_{f, z}(y) = \bigwedge \{ x / y \leq f(x) \} \). Then \( y \leq f(x) \Rightarrow m_{f, z}(y) \leq x \) is obvious. More, if \( m_{f, z}(y) \leq x \) then \( f(m_{f, z}(y)) \leq f(x) \). Let \( X = \{ x / y \leq f(x) \} \); by definition \( m_{f, z}(y) = \bigwedge X \) so \( y \leq \bigwedge f(X) = f(\bigwedge X) \leq f(x) \).

2. S1 \Rightarrow S3, by definition.

3. S3 \Rightarrow S4. Consider \( a, a' \in f^{-1}(b \uparrow) \) two distinct minimal elements. If they are consistent \( (a \uparrow a') \), then, by S3, \( f(a \wedge a') = f(a) \wedge f(a') \), so \( a \wedge a' \in f^{-1}(b \uparrow) \) contradicting the minimality of \( a \) and \( a' \).

4. S4 \Rightarrow S2 when \( D, E \) are algebraic and \( D \) satisfies property I. Let \( X \subseteq D \) be bounded by \( z \in D \). We have to show \( \bigwedge f(X) \leq f(\bigwedge X) \), and for this, using the fact that \( E \) is algebraic, it suffices to prove that \( \forall b \in K(E), b \leq \bigwedge f(X) \Rightarrow b \leq f(\bigwedge X) \). Suppose \( b \leq \bigwedge f(X) \). Clearly \( X \subseteq f^{-1}(b \uparrow) \). Using the property I of \( D \), we establish the following fact:

- \( \forall x \in f^{-1}(b \uparrow), \exists a_x \in K(D), a_x \leq x \), which is a minimal element in \( f^{-1}(b \uparrow) \). For the proof, let \( K_x = \{ a / a \leq x, a \in K(D) \} \). Because \( D \) is algebraic, \( K_x \) is directed and \( \bigvee K_x = x \). Then \( b \leq f(x) = f(\bigvee K_x) = \bigvee f(K_x) \). But \( b \) is compact, and \( f(K_x) \) directed, so
\[ \exists a \in K_x \text{ s.t. } b \leq f(a). \] Now we use the property I of \( D \) to find a minimal \( a \): if \( a \) is not minimal in \( f^{-1}(b) \), pick \( a' < a \) a smaller one, then pick \( a'' < a' \) to be even smaller etc. This sequence is finite, so we have to reach a minimal one, call it \( a_\alpha \), which is also compact (this is also a consequence of the property I: all elements below a compact element \( a \) are compact).

Because \( X \) is bounded, the set \( \{a_x \mid x \in X\} \) is also bounded, so, by property S4, all elements \( a_x \) must be equal to some element, call it \( a \), and we have, by construction, \( a \in f^{-1}(b) \). By the construction, \( \forall x \in X, a \leq x \), so \( a \leq \bigwedge X \) and we conclude \( b \leq f(a) \leq f(\bigwedge X) \).

\[ \square \]

**Definition 15** Let \( D, E \) be two consistently complete domains. A function \( f : D \to E \) is called **stable** iff \( f \) is continuous and satisfies S1 (or, equivalently, S2). We write \( [D \to_s E] \) for the set of stable functions.

Remark. When \( D, E \) are dI domains, stable functions are sometimes defined as being functions satisfying property S3.

The space \( ([D \to_s E], \leq_e) \) of stable functions with the extensional ordering (i.e. \( f \leq_e g \iff \forall x, f(x) \leq g(x) \)) does not form a exponentiation of \( D \) and \( E \) in the category of dI domains and stable functions, because \( eval : [D \to_s E] \times D \to D \) is not stable. Indeed it doesn’t satisfy S3: when \( x \leq y \) and \( f \leq g \), then \( (f, y) \) and \( (g, x) \) are bounded by \( (g, y) \) but \( eval(f, x) = eval((f, y) \land (g, x)) \) and is, in general, different from \( eval(f, y) \land eval(g, x) \). However, a different order relation on \( ([D \to_s E], \leq_e) \) is providing for the right exponentiation.

**Proposition 7** Let \( D, E \) be consistently complete domains, and \( f, g : D \to E \) be stable, \( f \leq_e g \). Consider the following:

1. **O1** \( \forall z, \forall y \leq f(z), m_{f,z}(y) = m_{g,z}(y) \).
2. **O2** \( x \leq x' \to f(x) = f(x') \land g(x) \).
3. **O3** \( \forall b \in K(E), the set of minimal elements in f^{-1}(b) \) is a subset of the minimal elements in \( g^{-1}(b) \).
Then $O1 \Rightarrow O2 \Rightarrow O3$, and, when $D, E$ are algebraic and $D$ satisfies property I, then $O3 \Rightarrow O1$.

**Proof**

1. $O1 \Rightarrow O2$.

   \[
   f(x') \land g(x) \leq g(x) \Rightarrow \\
   m_{g,x}(f(x') \land g(x)) \leq x \Rightarrow \\
   m_{f,x}(f(x') \land g(x)) \leq x \Rightarrow \\
   f(x') \land g(x) \leq f(x)
   \]

2. $O2 \Rightarrow O3$. Clearly $f^{-1}(b \uparrow) \subseteq g^{-1}(b \uparrow)$, so we just have to show that any minimal element in $f^{-1}(b \uparrow)$ is also minimal in $g^{-1}(b \uparrow)$. Let $a \in f^{-1}(b \uparrow)$ be minimal, and suppose $a_0 \in g^{-1}(b \uparrow)$, $a_0 < a$. Then $f(a_0) = f(a) \land g(a_0) \geq b$, contradicting the minimality of $a \in f^{-1}(b \uparrow)$.

3. $O3 \Rightarrow O1$, when $D, E$ are algebraic and $D$ satisfies property I. Recall, $m_{f,z}(y) = \land\{x / y \leq f(x)\}$. Let $y = b$, a compact element. Then, $m_{f,z}(b) = \land f^{-1}(b \uparrow)$. But, as we have shown, $f^{-1}(b \uparrow)$ has a unique minimal element $a$, which is compact, so $m_{f,z}(b) = a$. By O3, $a$ is also the unique minimal element of $g^{-1}(b \uparrow)$, so $m_{g,z}(b) = a$.

$\Box$

**Remark** O1 does not say that $m_{f,z} = m_{g,z}$ - this would imply $f = g$! It says only that $m_{f,z}$ and $m_{g,z}$ coincide on the domain of $m_{f,z}$, which is included in the domain of $m_{g,z}$. $\Box$

**Definition 16** The **stable order** on the set of stable functions $[D \rightarrow_s E]$ is:

\[
f \leq_s g \iff f \leq_c g \text{ and } f, g \text{ satisfy } O1
\]

It is known that the category of dI domains and stable functions is cartesian closed, with $([D \rightarrow_s E], \leq_s)$ the exponentiation of $D$ and $E$. But not all stable functions are sequential (under the definition given below, or in the sense that they cannot be defined in PCF with a sequential evaluator):
Example: Let $T = (\{\bot, t, f\}, \leq)$ and $O = (\{\bot, T\}, \leq)$. Consider the set $A \subseteq T^3$, $A = \{(t, f, \bot), (\bot, t, f), (f, \bot, t)\}$. Then $f : T^3 \to O$, $f(x) = T$ if $x \in A$, and $f(x) = \bot \iff x \not\in A$, is stable, but it is not sequential (as we shall prove below).

The notion of sequentiality of some function $f$, is intuitively related to the fact that $f$ inspects the components of its input, in a sequential fashion: thus, the “good” domains for defining sequential functions are cds, where “components” are cells. It happens that the notion of sequential function fits well only with dcds: a motivation could be that sequential functions are stable, and dI domains are the “right” domains for stable functions.

When $M, M'$ are cds, we shall write $f : M \to M'$ for a function $f : D(M) \to D(M')$. Also, when $x, y \in M$, we shall write $x \leq c y$ when $c \in A(x)$ and $c \in F(y)$.

**Definition 17** Let $M, M'$ be two dcds, and let $f : M \to M'$ be continuous. $f$ is called **sequential** at $x \in D(M)$ **w.r.t.** $c' \in A(f(x))$, if either $x$ is maximal (i.e. $A(x) = \phi$), or $\exists c \in A(x)$ such that $\forall y \geq x (f(x) <_{c'} f(y) \Rightarrow x <_{c} y)$. Such a cell $c$ is called a **sequentiality index** of $f$ at $x$ **w.r.t.** $c'$. We say that $c$ is a **strict index** if there exists $y \geq x$ such that $f(x) <_{c'} f(y)$.

$f$ is called **sequential** if it is sequential at all $x$, **w.r.t.** all $c' \in f(x)$. We denote $[M \to_{\text{seq}} M']$ or $[D(M) \to_{\text{seq}} D(M')]$ for the set of sequential functions from $M$ to $M'$.

**Proposition 8** A continuous function $f : M \to M'$ is sequential iff it is sequential at any finite $x \in D(M)$.

**Proof** Let $x \in D(M)$, $y = f(x)$ and $c' \in A(y)$. It is easy to check that $E$ is a continuous function, so from $c' \in E(y)$ we conclude $\exists y_0 \subseteq y, y_0$ finite, such that $c' \in E(y_0)$. Now $f$ is continuous, so $\exists x_0 \subseteq x, x_0$ finite, such that $y_0 \subseteq f(x_0)$, and $c' \in A(f(x_0))$. Then any sequentiality index of $f$ at $x_0$ **w.r.t.** $c'$ is also a sequentiality index of $f$ at $x$ **w.r.t.** $c'$.

**Proposition 9** A sequential function is stable.

**Proof** Suppose $f(x \land y) < f(x) \land f(y)$. Then $\exists c' \in A(f(x \land y))$ such that $f(x \land y) <_{c'} f(x) \land f(y)$. Let $c$ be a sequentiality index at $x \land y$ **w.r.t.** $c'$: then $x \land y <_{c'} x$, $x \land y <_{c'} y$, which is impossible, since, for dcds's, glb's of states are intersections. 

22
Remark Two connections between sequential cds and sequential functions are immediate:

1. Consider $M = (C, V, E, \vdash)$, and $M' = (C, V, E, \vdash')$ where $\forall c \in C, \vdash' c$. Then the function $f : D(M) \to D(M')$, $f(x) = x$ is sequential iff $M$ is sequential. □

2. Let $M = (C, V, E, \vdash)$, and $c' \in C$ be a cell, which occurs in at least one event $(c', v') \in E$. Let $O = (\{\bot, T\}, \leq)$ be the Sierpinsky space, and $f_{c'} : D(M) \to O$ be defined by: $f_{c'}(x) = T$ iff $c' \in E(x)$ ($c'$ is enabled in $x$). Then, $M$ is a sequential cds implies $f_{c'}$ is a sequential function.

The question arises: which are the right morphisms between concrete domains, or distributive concrete domains, which provide for a structure of a cartesian closed category?

Proposition 10 Neither of the following categories is cartesian closed:

1. cds (or dcds) and continuous functions.
2. dcds and stable functions.
3. dcds and sequential function.

Proof We give just a sketch, showing that the natural candidates for exponentiation don’t do the job. [1] proves that these candidates are also the only possible choices.

1. The exponentiation of $N_\bot$ and $O$ should be: $([N_\bot \to O], \leq_s)$. But this domain doesn’t satisfy property I, since the constant function $T$ is isolated, but dominates infinitely many elements.

2. The exponentiation of $T$ and $O$ should be: $([T \to_s O], \leq_s)$. This space has five elements, namely:
   
   - $z(\bot) = z(t) = z(f) = \bot$
   - $x(\bot) = \bot$, $x(t) = x(f) = T$
   - $x'_i(\bot) = x'_i(f) = \bot$, $x'_i(t) = T$ and $x'_j(\bot) = x'_j(t) = \bot$, $x'_j(f) = T$

23
• \( y(\bot) = y(t) = y(f) = T \)

Then axiom Q is not satisfied: \( z \prec_s x, z <_s y \) and \( x_\#_s y \) but there is no unique \( x' \) such that \( z \prec_s x' \prec_s y \) and \( x_\#_s x' \).

3. The exponentiation of \( T^3 \) and \( O \) should be: \( ([T^3 \to_{\text{seq}} O], \leq_s) \). Let \( A_1 = ((\{t, f, \bot\}) \uparrow, A_2 = ((\{\bot, t, f\}) \uparrow, A_1 = ((f, \bot, t)) \uparrow, \) and let \( f_i(x) = T \Leftrightarrow x \in A_i \). Then \( f_1, f_2, f_3 \) are sequential and pairwise compatible, but they have no lub (their lub, as stable functions, is the function \( f \) in the example after definition 16, which is not sequential).

\[ \Box \]

3.2 Concrete Sequential Algorithms

A sequential function may have several sequentiality indexes, at a given state \( x \), w.r.t. some \( c' \). The difference between sequential algorithms (defined below) and sequential functions, is that sequential algorithms provide also a choice of some sequentiality index. Thus, sequential algorithms are not extensional, but they do form a cartesian closed category having the concrete domains as objects.

A concrete sequential algorithm \( a : M \to M' \) will be defined as a subset of \( D_{\text{fin}}(M) \times C' \times (C \cup V') \) (where \( D_{\text{fin}}(M) \) denotes the set of finite states of \( M \)). The motivation behind this definition is that a concrete sequential algorithm acts like an output driven program. Suppose we have a concrete sequential algorithm \( a \) and a state \( x \), and that we want to find the value of the cell \( c' \in C' \), in \( a.x \) (where \( a.x \) is the result of applying \( a \) on the state \( x \)). Initially we don’t know anything about \( x \), so we look for an element of the form \( (\phi, c', c_1) \in a \), and suppose that \( c_1 \in C \) (the other possibility being \( c_1 \in V' \)). This instructs us to ask for the value of the cell \( c_1 \) in \( x \); suppose it is \( v_1 \), i.e. \( (c_1, v_1) \in x \). Now our knowledge about the input state \( x \) has increased to \( \{(c_1, v_1)\} \), so, at the second step, we look for an element \( \{(c_1, v_1)\}, c', c_2 \in a \). Suppose again \( c_2 \in C \), which instructs us to inspect the cell \( c_2 \) in \( M \): so our information about the state \( x \) grows again as we find out that “the value of” \( c_2 \) is \( v_2 \). Eventually, when our knowledge about \( x \) reaches, say, \( \{(c_1, v_1), \ldots, (c_n, v_n)\} \), we shall find some \( \{(c_1, v_1), \ldots, (c_n, v_n)\}, c', v' \in a \), with \( v' \in V' \), which tells us to output \( v' \),
as the value of \( c' \). Of course, this process might never terminate, or might get stuck either because we are instructed to inspect some cell in \( x \) which is not filled, or because we don’t find the right “instruction” in \( a \).

The definition of a sequential algorithm as a subset \( a \subseteq D_{\text{fin}}(M) \times C' \times (C \cup V') \) has to insure that:

1. For any state \( x \), the set \( a.x \in C' \times V' \) (informally described above) is indeed a state in \( M' \).

2. The function associated with \( a \) is sequential.

3. The set of concrete sequential algorithms \( CA[M, M'] \), ordered by set inclusion, is again a distributive concrete domain, or, equivalent, that this set can be presented as the set of states of a suitable dcds.

**Definition 18** Let \( M = (C, V, E, I) \), \( M' = (C', V', E', I') \) be two dcds. A concrete sequential algorithm \( a : M \rightarrow M' \) is a subset \( a \subseteq D_{\text{fin}}(M) \times C' \times (C \cup V') \), satisfying the following conditions (elements of \( a \) are denoted by \( (xc', u) \)):

\[ CA0 \] If \( (xc', v') \in a \) and \( v' \in V' \), then \( (c', v') \in E' \). Let \( a/E' := \{(c', v')/c' \in C', v' \in V', \exists x \in D_{\text{fin}}(M), (xc', v') \in x \} \) is included in \( E' \).

\[ CA1 \] For any \( c \in C' \), the set \( a/c' = \{(x, u) \mid (xc', u) \in a \} \) is an indexed forest on \( M, M' \), i.e. satisfies the following conditions:

\[ IF0 \] If \( (x, c) \in a/c' \) with \( c \in C \), then \( c \in A(x) \) (i.e. \( a \) is allowed to instruct us to ask the value of some cell only when we know that the cell is enabled !).

\[ IF1 \] If \( (x, u), (y, w) \in a/c' \) and \( x \uparrow y \), then \( x \leq y \) or \( y \leq x \) (this requirement is connected to the sequentiality of the function implemented by \( a \): from the state \( x \cap y \) we should have only “one way” to get to the state \( x \cup y \). More, if \( x = y \) then \( u = w \) (this ensures the consistency condition of \( a.x \): we shouldn’t be able to fill the cell \( c' \) with two different values).

\[ IF2 \] If \( (x, u), (y, w) \in a/c' \) with \( x \leq y \), then there exists a covering chain from \( (x, u) \) to \( (y, w) \), i.e. there are \( z_0, z_1, \ldots, z_n, c_0, c_1, \ldots, c_n \), with \( x = z_0, u = c_0 \), such that:
and \((z_0, c_0), \ldots, (z_n, c_n) \in a/c'.\)

**CA2** Define \(E(a) \subseteq D(M) \times C'\) to be the smallest set closed under the following rules (elements of \(E(a)\) are denoted by \(xc'\)):

- **R1** \((xc', c) \in a \quad xc' \in E(a) \quad x \prec_y y \quad yc' \in E(a)\)

- **R2** \((x_1c'_1, v'_1) \in a, \ x_1c'_1 \in E(a), i = 1, \ldots, n, \ (c'_i, v'_i), \ldots, (c'_n, v'_n) \vdash c' \quad (x_1 \cup \ldots \cup x_n)c' \in E(a)\)

Then \(a/c' \subseteq E(a)\) (this insures that, whenever we ask about a cell \(c'\), there is enough evidence in \(x\) to guarantee that \(c'\) is enabled in \(a.x\)).

First we prove that to each concrete sequential algorithms corresponds a sequential function.

**Definition 19** Let \(a : M \rightarrow M'\) be a concrete sequential algorithm. For \(x \in D(M)\), let \(a.x = \{(c', v') \in E / \exists x_0 \subseteq x, \ (x_0c', v') \in a\}\).

**Proposition 11** \(a.x\) is a state of \(M'\).

**Proof** By CA0, \(a.x \subseteq E'\). We have to check consistency and safety:

1. Consistency. Suppose \((c', v'_0), (c', v'_1) \in a.x\). Then \(\exists x_0, x_1 \subseteq x, \ x_0, x_1\) finite, such that \((x_0c', v'), (x_1c', v'') \in a\), i.e. \((x_0, v'), (x_1, v'') \in a/c'\), which is an indexed forest. So, by IF1, we may suppose \(x_0 \subseteq x_1\). Because \(v'\) is not a cell, the covering chain (see IF2) from \((x_0, v')\) to \((x_1, v'')\) is empty, so \(x_0 = x_1\) and, by IF1, \(v' = v''\).
2. Safety. We have to prove: if \( x_0 \subseteq x \) and \((x_0c', v') \in a\) then \((c', v') \in E(a.x)\). Its enough to show that \((c', v') \in E(a.x)\), For this, we prove that \((xc', v') \in a\) implies \((c', v') \in E(a.x)\), by induction on the number of steps in the proof of \(xc' \in E(a)\). The key here is to observe that the rule R1 does not change the cell \(c'\)! So suppose \((xc', v') \in a\): we go back in the proof, up to the preceding use of the rule R2: let it be:

\[
\begin{align*}
(x_i c_i', v_i') &\in a, \\
x_i c_i' &\in E(a), i = 1, \ldots, n, \\
(c_i', v_i'), \ldots, (c_n', v_n') &\vdash' c' \\
(x_1 \cup \ldots \cup x_n)c' &\in E(a)
\end{align*}
\]

with \(x_1 \cup \ldots \cup x_n = z_0\). By CA1, \(a/c'\) is an indexed forest, so there is a covering chain from \(z_0\) to \(x\). By induction, \(c'_i \in E(a.x_i)\). Clearly \((c'_i, v'_i) \in E(a.x_i)\), and \(E(a.x_i) \subseteq E(a.z_0) \subseteq E(a.x)\). Using \((c'_1, v'_1), \ldots, (c'_n, v'_n) \vdash' c'\), we conclude that \(c' \in E(a.x)\).

\(\square\)

This proposition justifies the following:

**Definition 20** For a concrete sequential algorithm \(a : M \rightarrow M'\), we write \(\tilde{a} : D(M) \rightarrow D(M')\) for the function \(\tilde{a}(x) = a.x\).

Next, we prove that \(\tilde{a}\) is a sequential function. For this we associate to some concrete sequential algorithm \(a\) a function \(i_a : D(M) \times C' \rightarrow C \cup \{\omega\}\). \(i_a(x, c')\) essentially chooses a sequentiality index for \(\tilde{a}\) at \(x\) w.r.t. \(c'\).

**Definition 21** Let \(a : M \rightarrow M'\) be a concrete sequential algorithm, and \(x \in D(M)\), \(c' \in C'\). Define \(i_a : C_{fin}(M) \times C' \rightarrow C \cup \{\omega\}\) as follows:

1. \(i_a(x, c') = c\) iff \(c' \in A(a.x)\), \(c \in A(x)\) and \(\exists x \subseteq x\) s.t. \((x_0c', c) \in a\).
2. \(i_a(x, c') = \omega\) otherwise.

**Proposition 12** \(i_a\) is well defined, i.e. if \(i_a(x, c') = c\), then \(c\) is unique.

**Proof** Suppose there are \(x_0, x_1 \subseteq x\) s.t. \((x_0c', c_0) \in a\), \((x_1c', c_1) \in a\) and \(c_0, c_1 \in A(x)\) (this would imply both \(i_a(x, c') = c_0\) and \(i_a(x, c') = c_1\)). Then, by CA1 (IF1), we may suppose \(x_0 < x_1\). By IF2, \(c_0 \in F(x_1)\), so \(c_0 \in F(x)\), contradicting the hypothesis \(c_0 \in A(x)\). \(\square\)
**Proposition 13** If \( a : M \to M' \) is a concrete sequential algorithm, then \( \bar{a} \) is a continuous, sequential function.

**Proof** Clearly \( \bar{a} \) is continuous, as \( a.x = \cup \{ x_0 / x_0 \leq x, \ x_0 \) finite\}.

Let \( c' \in A(a.x) \) and suppose \( \exists y \geq x \) s.t. \( c' \in F(a.y) \) i.e. \( \exists y' (c', v') \in F(a.y) \), so \( \exists y_0 \subseteq y \) s.t. \( (y_0c', v') \in a. \) Consider the last R2 rule used for proving \( y_0c' \in E(a) \). It has to be of the form:

\[
(x_i c'_i, v'_i) \in a, \ x_i c'_i \in E(a), i = 1, \ldots, n \quad \text{and} \quad (c'_1, v'_1), \ldots, (c'_n, v'_n) \vdash c' \quad \text{R2}
\]

and \( x_1 \cup \ldots \cup x_n \leq y_0 \) (because of the way R1 works). So \( (c'_i, v'_i) \in a.y \) \( i = 1, \ldots, n \) and \( \{(c'_1, v'_1), \ldots, (c'_n, v'_n)\} \vdash c' \) is an enabling of \( c' \) in \( a.y \). Because \( c' \in A(a.x) \), it has also an enabling in \( a.x \), which must be the same as the one above, else we would have two enablings of \( c' \) in \( a.y \), contradicting the fact that \( M' \) is deterministic. Thus \( (c'_i, v'_i) \in A.x \), i.e. \( \exists x_{0i} \leq x \) s.t. \( (x_{0i}c'_i, v'_i) \in a. \) But we also have \( (x_i c'_i, v'_i) \in a \) so, by CA1, \( x_{0i} = x_i. \) Let \( x_0 = x_1 \cup \ldots \cup x_n \): we have shown that \( x_0 \subseteq x \). Recall that R1 had to be applied 0 or more times in order to prove \( y_0c' \in E(a) \) from \( x_0c' \in E(a) \), so there is a sequence of cells \( c_1, c_2, \ldots \) s.t. \( x_0 \prec c_1 \prec c_2 \prec \ldots \prec c_n y_0 \). Let \( c_k \) be the first one for which \( c_k \notin F(x) \). (From \( x_0 \leq x \) and \( y_0 \notin x \), such a cell must exist). Clearly \( c_k \in A(x) \), and by definition \( i_a(x, c') = c_k \), so we have proven \( i_a(x, c') \in F(y_0) \subseteq F(y) \) and, as \( y \) was arbitrary, \( i_a(x, c') \) is a sequentiality index for \( \bar{a} \) at \( x \) w.r.t. \( c' \). \( \square \)

Remark. We used in an essential way the fact that \( M' \) is deterministic.

The next proposition shows how the inclusion ordering on concrete algorithms relates to the function they compute.

**Proposition 14** Let \( a, a' \in CA[M, M'] \) be two concrete sequential algorithms. Then \( a \subseteq a' \) implies \( \bar{a} \leq_s \bar{a}' \) and \( \forall c' \in A(a.x), (i_a(x, c') \neq \omega \Rightarrow i_a(x, c') = i_{a'}(x, c')) \)

**Proof** Suppose \( a \subseteq a' \). Clearly \( \bar{a} \leq_s \bar{a}' \). To check the stable order, suppose \( x \leq y \), and let \( (c', v') \in \bar{a}(y) \cap \bar{a}'(x) \). \( (c', v') \in a.y \) implies \( \exists y_0 \subseteq y \) s.t. \( (y_0c', v') \in a. \) Similarly, \( (c', v') \in a'.x \) implies \( \exists x_0 \subseteq x \) s.t. \( (x_0c', v') \in a'. \) So both \( (y_0c', v') \) and \( (x_0c', v') \) are in \( a' \); by CA1 (IF2), \( x_0 = y_0 \), and \( (c', v') \in a.x \). This proves \( \bar{a} \leq_s \bar{a}' \).
Now let \( c' \in A(a.x) \) and \( i_a(x, c') = c \neq \omega \), i.e. \( c \in A(x) \) and \( \exists x_0 \leq x \) \( (x_0c', c) \in a \). We have to show that \( c' \in A(a'.x) \) (\( = E(a'.x) - F(a'.x) \)) and that \( \exists x'_0 \) s.t. \( (x'_0c', c) \in a' \). From \( c' \in E(a.x) \) we have \( c' \in E(a'.x) \). Suppose \( c' \in F(a'.x) \) i.e. \( \exists x'_0 \leq x \) s.t. \( (x'_0c', v') \in a' \) for some \( v' \). By CA1 there must be a covering chain from \( x_0 \) to \( x'_0 \) and this must start with the cell \( c \) (because \( (x_0c', c) \in a \)), so \( c \in F(x'_0) \) and this implies \( c \in F(x) \), contradiction. So we have proved: \( c' \notin F(a'.x) \), i.e. \( c' \in A(a'.x) \). More, \( (x_0c', c) \in a \) implies \( (x_0c', c) \in a' \) so \( i_{a'}(x, c') = c \). □

Now we show how to express the set of concrete sequential algorithms as the set of states of some dcds.

**Definition 22** Let \( M, M' \) be two dcds. We define the cds \([M \rightarrow_{sa} M']\) to be \((C_0, V_0, E_0, \top_0)\), where:

1. \( C_0 = D_{fin}(M) \times C' \).
2. \( V_0 = C \cup V \)
3. \( E_0 = \{ (xc', c) / c \in A(x) \} \cup \{ (xc', v') / (c', v') \in E' \} \).
4. \( \top_0 \) is defined by the following two rules:

\[
\begin{align*}
\frac{x \prec c y}{(xc', c) \vdash_0 yc} & \quad R1 \\
\frac{(c_1', v_1'), \ldots, (c_n', v_n')}{(x_1c_1', v_1'), \ldots, (x_nc_n', v_n')} & \vdash_0 xc' \quad R2
\end{align*}
\]

We state, without proof, the following:

**Proposition 15** When \( M, M' \) are dcds, then \([M \rightarrow_{sa} M']\) is also a dcds.

This is not obvious: the proof is quite involved, and can be found in [1].

Although we haven't chosen to work with well founded dcds, they do simplify some of the proofs. The following proposition shows that nothing is lost, when restricting to well founded dcds:
Proposition 16 If $M, M'$ are well-founded dcds, then $[M \rightarrow_{sa} M']$ is also well-founded.

Proof $x_1c'_1 < x_2c'_2$ iff $x_1 < x_2$ and $c'_1 = c'_2$ or $x_1 \leq x_2$ and $c'_1 \leq c'_2$. \qed

Proposition 17 The set of states of $[M \rightarrow_{sa} M']$ is exactly $CA[M, M']$

Proof The proof is lengthy and quite straightforward, and is omitted. \qed

3.3 Abstract Sequential Algorithms

A concrete sequential algorithm is a rather complicated object $a$, to which we can associated both the sequential function $\tilde{a}$ computed by $a$, and a function $i_a(x, c')$ which chooses sequentiality indexes for $\tilde{a}$. An abstract sequential algorithm is just the pair $(\tilde{a}, i_a)$. Thus, to each concrete sequential algorithm we can associate an abstract one, but we shall see that the converse also holds: each abstract sequential algorithm “comes” from a concrete one.

Definition 23 Let $M, M'$ be dcds. An abstract sequential algorithm from $M$ to $M'$, $\alpha : M \rightarrow M'$, is $\alpha = (\tilde{\alpha}, i_{\alpha})$, where $\alpha : D(M) \rightarrow D(M')$ and $i_{\alpha} : D(M) \times C' \rightarrow C \cup \{\omega\}$ is an index choice function, such that:

AA1 If $i_{\alpha}(x, c') = c$, then $c \in A(x)$ and $c' \in A(\tilde{\alpha}(x))$.

AA2 If $i_{\alpha}(x, c') = c$, then $\exists x_0 \leq x$ finite, s.t. $i_{\alpha}(x_0, c') = c$.

AA3 If $c' \in A(\tilde{\alpha}(x))$ and $\exists y \geq x c' \in F(\tilde{\alpha}(y))$, then $i_{\alpha}(x, c')$ is defined (i.e. $\neq \omega$) and is an index of $\tilde{\alpha}$ at $x$ w.r.t. $c'$.

This implies that $\tilde{\alpha}$ is sequential.

AA4 If $i_{\alpha}(x, c') = c$, $x \leq y$ and $c \notin F(y)$, then $i_{\alpha}(y, c') = c$.

AA5 If $i_{\alpha}(x, c') = c$, $y \leq x$ and $c' \in A(\tilde{\alpha}(y))$, then $i_{\alpha}(y, c')$ is defined.

We denote with $AA[M, M']$ the set of abstract algorithms from $M$ to $M'$.

Definition 24 Let $\alpha, \beta \in AA[M, M']$. Define the order relation $\leq$ by:

30
\[ \alpha \leq \beta \text{ iff } \tilde{\alpha} \leq \tilde{\beta} \text{ and } \forall x \in D(M), \forall c' \in A(\tilde{\alpha}(x)), (i_\alpha(x,c') \neq \omega \Rightarrow i_\alpha(x,c') = i_\beta(x,c')). \]

**Proposition 18** If \( a \in CA[M,M'] \) is a concrete algorithm, then \( \varphi(a) = (\tilde{a}, i_\alpha) \) is an abstract algorithm. More, if \( a, b \in CA[M,M'] \) and \( a \leq b \), then \( \varphi(a) \leq \varphi(b) \).

**Proof** The first part is straightforward checking of AA1-AA5. The second part is proved by proposition 14. \( \square \)

The next proposition proves that a concrete sequential algorithm \( \psi(a) \) corresponds to each abstract sequential algorithm \( a \). First we define the construction \( \psi \):

**Definition 25** Let \( M, M' \) be dcds, and \( \alpha \in AA[M,M'] \). Define \( \psi(\alpha) \subseteq D_{fin}(M) \times C' \times (C \cup V') \) to be:

\[
\begin{align*}
\psi(\alpha) &= \{(x',v') / \tilde{\alpha}(x) = (c',v') \text{ and } y \leq x \Rightarrow \tilde{\alpha}(y) \neq (c',v')\} \cup \\
&\{(x',c) / i_\alpha(x,c') = c \text{ and } y \leq x \Rightarrow i_\alpha(x,c') \neq c \text{ (i.e. } i_\alpha(x,c') = \omega)\}
\end{align*}
\]

**Proposition 19**
1. If \( \alpha \in AA[M,M'] \) then \( \psi(\alpha) \in CA[M,M'] \).
2. If \( \alpha, \beta \in AA[M,M'] \), \( \alpha \leq \beta \) then \( \psi(\alpha) \subseteq \psi(\beta) \).

**Proposition 20** The functions \( \varphi, \psi \) defined above are inverse to each other.

**Corollary 2** Let \( f : D(M) \to D(M') \) be a sequential function. Then there exists a concrete sequential algorithm \( a : M \to M' \), such that \( f = \tilde{a} \).

**Proof** Omitted. It can be found in [5] \( \square \)

Based on this 1 to 1 correspondence, we shall not distinguish between concrete and abstract sequential algorithms, and write \( A[M,M'] \) for any of the sets \( CA[M,M'] \) or \( AA[M,M'] \).
3.4 The Category of Sequential Algorithms

Definition 26 The category of sequential algorithms DCDS-SA is defined as:

1. Objects are dcads.
2. Morphisms from $M$ to $M'$ are sequential algorithms
3. The identity $1_M$ is: $\tilde{1}_M = 1_{D(M)}$, $i_{1M}(x, c) = c$ iff $c \in A(x)$.
4. Composition of $a : M \rightarrow M'$ and $a' : M' \rightarrow M''$ is defined as:
   
   (a) $a' \circ a = \tilde{a}' \circ \tilde{a}$
   
   (b) $i_{a'\circ a}(x, c'') = i_{a}(x, i'_{a}(a.x, c''))$ (with the convention $i_{a}(x, \omega) = \omega$)

Proposition 21 The above definition is correct, i.e. the identity and the composition $a' \circ a$ are indeed sequential algorithms, and the requirements of a category are satisfied.

Remark: it can be shown that isomorphisms in this category are exactly those sequential algorithms $a$, for which $\tilde{a}$ is an order-isomorphism.

Next we shall sketch the proof of the fact that DCDS-SA is cartesian closed.

Proposition 22 DCDS-SA is closed under $\omega$-products.

Proof (Sketch) Let $(M_i)_{i \in \omega}$, $M_i = (C_i, V_i, E_i, \uparrow_i)$. Then:

$$\Pi_{i \in \omega} M_i = (\bigcup_{i \in \omega} C_i, \bigcup_{i \in \omega} V_i, \bigcup_{i \in \omega} E_i, \bigcup_{i \in \omega} \uparrow_i)$$

(All the unions are supposed to be disjoint). It is easy to check that $D(\Pi_{i \in \omega} M_i) = \Pi_{i \in \omega}(D(M_i))$ where $\Pi$ stands for the product of cpos.

Definition 27 Let $M, M'$ be dcads. Then we define $\text{app} : \left[M \rightarrow_{sa} M'\right] \times M \rightarrow M'$ to be:

1. $\text{app}(a, x) = a.x$
2. Suppose \( c' \in A(a.x) \) then \( i_{app}((a, x), c') \) is defined as follows:

(a) If \( i_a(x, c') = c \) then define \( i_{app}((a, x), c') = c \).

(b) Else, if \( \exists z \leq x \) s.t. \( zc' \in A(a) \) then \( i_{app}((a, x), c') = zc' \).

(c) Else \( i_{app}((a, x), c') = \omega \).

**Proposition 23** \( \text{app} \) is well defined and is a sequential algorithm.

*Proof* We show only that \( i_{app} \) is well defined. Indeed, if there were two \( z \)'s at point 2b, say \( z \) and \( z_1 \), then, for any \( v, v_1, a \cup \{(zc', v), (z_1c', v_1)\} \) would be a correct state of \( [M \rightarrow_{sa} M'] \), but it would invalidate CA1. \( \square \)

We state without proof:

**Theorem 4** DCD-SA is a cartesian closed category, in which exponentiation is \( [M \rightarrow_{sa} M'] \) and the counit is \( \text{app} \).

### 3.5 Dcds as models for PCF

Consider the programming language **PCF**, an extension of the typed lambda-calculus, with two base types (\( i \) for integers and \( o \) for booleans), and the following constants:

- \( n : i \) (a positive integer), \( tt, ff : o, +1, -1 : i \rightarrow i \), \( zero : i \rightarrow o, if_i : o \rightarrow i \rightarrow i \rightarrow i, if_o : o \rightarrow o \rightarrow o \rightarrow o, Y((\sigma \rightarrow \sigma) \rightarrow \sigma) : (\sigma \rightarrow \sigma) \rightarrow \sigma \) (at each type \( \sigma \)).

Operational semantics is defined by a collection of rules of the form \( M \rightarrow M' \), traditionally grouped into three categories:

\[(\beta) (\lambda x^\sigma. M) N \rightarrow M[N/x^\sigma]\]

\[(\delta) (+1)n \rightarrow n + 1, (-1)(n + 1) \rightarrow n, \text{if } i, 0 M_1 M_2 \rightarrow M_1 \text{ etc.}\]

\[(Y) YM \rightarrow M(YM).\]

A program \( P \) is a closed term of ground type \( i \) or \( o \). For any program there is at most one constant \( c \) of its ground type such that \( M \rightarrow^* c \): we write \( P =_{op} c \) in this case. The operational preorder at type \( \sigma \), \( \leq_{op}^\sigma \) is defined as:

\[M, N : \sigma, M \leq_{op}^\sigma N \text{ iff } \forall\ P[\ ]^\sigma, \forall c. \ P[M] =_{op} c \Rightarrow P[N] =_{op} c\]
Definition 28 A continuous model of PCF is defined by:

1. A **value cpo** at each type \( \sigma: (D^\sigma, \leq^\sigma) \), such that the cpo’s \( D^i \) and \( D^o \) are the flat domains of integers and booleans. The product \( \Pi_{x \in \mathbb{V}_a} D^\sigma \) is called the **environment** \( \text{ENV} \).

2. A **semantic cpo** at each type \( \sigma: (E^\sigma, \leq^\sigma) \). In most models, \( E^\sigma = [\text{ENV} \rightarrow D^\sigma] \).

3. A **continuous mapping** \( \text{eval}^\sigma: E^\sigma \times \text{ENV} \rightarrow D^\sigma \). We abbreviate \( \text{eval}(e, \rho) \) by \( e\rho \).

4. A **continuous mapping** \( \cdot: D^\sigma \rightarrow \rightarrow D^\sigma \rightarrow D^\sigma \).

5. A **semantic mapping** \( \llbracket \cdot \rrbracket \) from terms of type \( \sigma \) to \( E^\sigma \), which satisfies the following properties:

   - \((\text{conv})\) \( M \rightarrow^* N \Rightarrow \llbracket M \rrbracket = \llbracket N \rrbracket \).
   - \((\Omega)\) \( \llbracket \Omega \rrbracket = \bot \).
   - \((\text{mon})\) \( \llbracket M \rrbracket \leq \llbracket N \rrbracket \Rightarrow \forall C \cdot \llbracket C[M] \rrbracket \leq \llbracket C[N] \rrbracket \).
   - \((\text{cont})\) \( \bigcup_{i \in \omega} [M_i] = [M] \Rightarrow \bigcup_{i \in \omega} [C[M_i]] = [C[M]] \).
   - \((\text{var})\) \( \llbracket x \rrbracket \rho = \rho(x) \).
   - \((\text{app})\) \( \llbracket MN \rrbracket \rho = (\llbracket M \rrbracket \rho) \cdot (\llbracket N \rrbracket \rho) \).
   - \((\lambda\text{lambda})\) \( \llbracket \lambda x . M \rrbracket \rho \cdot d = [M] \rho[x \leftarrow d] \).
   - \((\text{free})\) \( \llbracket M \rrbracket \rho = \llbracket M \rrbracket \rho' \) if \( \rho(x) = \rho'(x) \) for all \( x \) free in \( M \).
   - \((\text{int})\) \( \llbracket n \rrbracket \rho = n \) for all numbers \( n \).
   - \((\text{bool})\) \( \llbracket \text{tt} \rrbracket \rho = t \), \( \llbracket \text{ff} \rrbracket \rho = f \).

A model \( \mathcal{M} \) induces a preorder \( \leq_{\mathcal{M}} \) on terms, defined by \( M \leq_{\mathcal{M}} N \) if \( \llbracket M \rrbracket \leq [M'] \).

A model is called **environment extensional** if \( e\rho = e'\rho \) for all \( \rho \) implies \( e = e' \), and **value extensional** if \( d \cdot d'' = d' \cdot d'' \) for all \( d'' \) implies \( d = d' \). We call the model **extensional** if it is both environment and value extensional.

We define **order extensional** models in a similar way, replacing equalities with inequalities.

A model is called **\( \eta \)-model** if it satisfies also:
(η) \([\lambda x.Mx] = [M]\), if \(x\) is not free in \(M\).

A model is a **least fixpoint model** if \([Y]\) is the least fixpoint operator.

**Definition 29** A model is **syntactically continuous** if for any term \(M\):

\[
[M] = \sqrt\{\{t\} / t \text{ a finite Boehm tree}, t \leq_{\Omega} BT(M)\}
\]

where \(BT(M)\) is the (possible infinite) Boehm tree associated to \(M\), and \(\leq_{\Omega}\) is the “Ω-match” ordering on Boehm trees, i.e., the congruent order relation generated by \(Ω \leq_{\Omega} T\), for any Boehm tree \(T\).

This property immediately implies “adequacy”:

**Proposition 24** In a syntactically continuous model, the following holds: \(M \leq_{\mathcal{M}} M' \Rightarrow M \leq_{op} M'\). We call this property of a model, **adequacy**.

**Proof** If \(M =_{op} c\), then \([M] = [c]\) so \([M'] = [c]\) (because the domains associated to the base types are flat). From the syntactic continuity for \(M'\) and from the fact that \([c]\) is compact, we conclude \(\exists t \leq_{\Omega} BT(M')\) s.t. \(t\) is finite and \([t] = [c]\). But then \(t = c\), so \(M' =_{\Omega} c\). □

**Definition 30** A model \(\mathcal{M}\) for which the converse of adequacy is also true, is called **fully abstract**.

**Theorem 5** Any least fixpoint model is syntactically continuous.

**Corollary 3** Any least fixpoint model is adequate.

**Definition 31** The algorithm model \(\mathcal{A}\) is defined by:

1. \(D^i\) and \(D^o\) are the dcds of integers and booleans and \(D^{e \rightarrow r} = [D^e \rightarrow_{sa} D^r]\).
2. \(E^\sigma = [ENV \rightarrow_{sa} D^\sigma]\).
3. \([M]\) is defined by induction on terms, in a standard way. E.g. \([Y]\) is defined to be the least fixpoint algorithm \([Y] : [D \rightarrow_{sa} D] \rightarrow_{sa} D\).
Theorem 6 The algorithm model $\mathcal{A}$ is a least fixpoint $\eta$-model. It is neither environment- nor value-extensional.

The lack of extensionality is due to the fact that several choices of sequenticality indexes are possible of a given sequential function. It has as consequence the fact that the model $\mathcal{A}$ is, trivially, non fully abstract; indeed consider the following terms, corresponding to left and right addition:

$$Add_l = Y(\lambda f. \lambda xy. if_l(zero x)y(+1(f(-1x)y)))$$
$$Add_r = Y(\lambda f. \lambda xy. if_l(zero y)x(+1(f(-1y))))$$

Although $Add_l \equiv_{op} Add_r$ (because for a standard model $[Add_l] = [Add_r]$), we don’t have in the algorithm model $\mathcal{A}$, $[Add_l] = [Add_r]$. However, recent work by [4] and [6] has shown that the ability to distinguish the order of argument evaluation in PCF is enough to make this model fully abstract. We just mention the result from [6]:

**Theorem 7** Let $\text{PCF} + \text{catch}$ be the extension of $\text{PCF}$ by the constants $\text{catch}^a : \sigma \rightarrow i$. The operational semantics of $\text{PCF} + \text{catch}$ is such that, whenever $M : \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow i$ or $M : \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow o$, and, during the evaluation of $MM_1 \ldots M_n$ $M$ “inspects” first the argument with number $k$, then $\text{catch}(M) \rightarrow^* k$. Then the algorithmic model $\mathcal{A}$ can be naturally extended to a model of $\text{PCF} + \text{catch}$ (by giving the right interpretation to $\text{catch}$), which is fully abstract.
Chapter 4

Deterministic Concrete Data Structures as a Programming Language

Following [2], we shall sketch a programming language, called CDS0 after [2], which essentially provides syntax for dcds and states. The language CDS0, has two main parts: the language for specifying dcds and the language for specifying states. States of dcds of the form $[M \rightarrow_s M']$, which are essentially concrete sequential algorithms, may be specified in a friendlier way than by enumerating its components.

4.1 Specifying Concrete Data Structures in CDS0

The syntax in CDS0 for a dcds $M = (C, V, E, \vdash)$ is:

```plaintext
dcbs
    cell <name> values <values> [access ... or ...]
...
end
```

where each line contains the informations about one cell: $<name>$ is the name of a cell $c \in C$, $<values>$ is the set of values which it may receive (i.e.
\{v / (c, v) \in E\}, and the optional “access” part contains all the finite sets \{(c_1, v_1), \ldots, (c_n, v_n)\}, separated by “or”, for which \{(c_1, v_1), \ldots, (c_n, v_n)\} \vdash c.

Names are assigned to dcgs by declarations of the form:

let <name> =
  dcgs
  ...
end

Examples.

let BOOL =
  dcgs
  cell B values T,F
end

let FIRST_FIVE_OR_TEN =
  dcgs
  cell CHOICE values F,T
  cell N5 values [0..5] acces CHOICE=F
  cell N10 values [0..10] acces CHOICE=T
end

let INTEGER =
  dcgs
  cell N values [\ldots]
end

The names for the cells are implicitly supposed to be sequence of characters, so that the following grafting constructor makes sense. In a construct like:

dcgs
  ...
  graft M.c [access <conditions>]

38
where M is supposed to be the name of another dc
ds, all the cells from M
are textually inserted in the new dc
ds, with their names appended with the
symbol c. The values of the inserted cells are those from M,
and the access
conditions are those form M extended with the new <conditions> (when
they are present).

Examples.

let PROD =
  dc
ds
    graft M1.1
    graft M2.2
end

let SUM =
  dc
ds
    cell CHOICE values L,R
    graft M1.1 access CHOICE=L
    graft M2.2 access CHOICE=R
end

Then $D(\text{PROD}) = D(M1) \times D(M2)$, while $D(\text{SUM}) = D(M1) + D(M2)$.

In CDS0, $\#^1$ is a predefined infix binary operation on dc
ds: $M1 \# M2$ is the
product of $M1$ and $M2$, as defined by the dc
ds PROD.

A recursion construction allows for the definition of “infinite” dc
ds, i.e.
dc
ds with infinite many cells.

Example (see example LIST in section 2.1):

letrec LIST(*) =
  dc
ds
    graft INTEGER.1
    cell NEXT values NIL,POINTER

---

$^1$Not to be confused with the incompatibility relation in ordered sets.
The semantics of recursive constructions is much simpler than the semantics of recursive type constructions using domain equations, because it suffices, essentially, to construct the set of cells in the resulting dcds, which is (at least for the examples so far) a regular language.

A session in CDS0 has, conceptually, two parts:

1. First the user specifies a state of some dcds previously declared. This can be done
   
   (a) directly, like in:

   \{N=12\}

   The compiler should check the constrains of the dcds, i.e. that all cell-values pairs are events, that each cell has at most one value, and that all filled cells are enabled.

   (b) by giving them names, like in:

   \text{TRUE=\{B=T\}}

   and later referring to that name. Again, the compiler checks consistency and safety.

   (c) by performing some more complicated operation, like applying a sequential algorithm (to be described in the next section) to a given state.

2. Then the user repeatedly introduces a name of some cell, and receives an answer which can be:

   (a) The value of that cell.
   
   (b) No value at all (if this can be “seen” in the current state)
   
   (c) An infinite computation, never producing any value.
4.2 Sequential Algorithms

Sequential algorithms are states of the dcds $[M \rightarrow_{sa} M']$, so they could be expressed using the syntax described so far, by enriching the alphabet of the names of the cells, to make it closed under finite states. However, sequential algorithms are hardly readable in this syntax, so an alternative syntax is preferred, which emphasizes the “indexed forest” structure of a sequential algorithm.

A declaration of a sequential algorithm $a$ of type $[M \rightarrow_{sa} M']$ is a collection of instructions for each cell $c'$ in the output dcds, preceded by the “request” keyword:

```
algo
  request c' do
    <instruction>
  end
  ...
end
```

(some cells may have no corresponding requests). The `<instruction>` is a collection of “from”-constructions:

```
from <state> do
  <inst1>
  <inst1>
  ...
end
from <state> do
  <inst1>
  <inst1>
  ...
end
```

Each `<state>` is a finite state of $M$ such that $c'$ is enabled in $a$. `<state>`.

When `<state>` is empty, the “from” header and its corresponding “end” may miss altogether.
<inst1> can have one of the following two forms:

1. output <value>, with \((c',<value>)\) an event of \(M'\).

2. \(\text{valof } c\)
   
   \[
   \begin{align*}
   v_1: & <\text{inst1}>_1 \\
   \vdots  & \\
   v_n: & <\text{inst1}>_n \\
   \text{end}
   \end{align*}
   \]

The meaning of such a declaration is best understood if we think of sequential algorithms as output-driven devices. Whenever we want to find the value of some cell \(c'\) in \(a.x\) (where \(a\) is the sequential algorithm and \(x\) is the input state), we search for “request \(c'\) do <instruction> end”, in the body of \(a\). If there is no such construction, then \(c'\) is definitely not defined in \(a.x\). If there is, suppose for the moment that <instruction> doesn’t contain any “from” construction, so it consists solely of “valof” and “output” instructions. A “valof \(c\)” instructs us to inspect the value of the cell \(c\) in \(x\); if it is filled, then according to its value \(v_i\) we have to proceed with the corresponding <inst1> instruction, which might be another “valof”. Eventually we will reach an “output \(v'\)” instructions, which tells us that \(c'\) is filled with \(v'\) in \(a.x\).

The presence of “from” constructions is connected with the enabling relation in \(M'\); we are not allowed to ask about the value of some cell \(c'\) in \(a.x\) unless we have enough knowledge about the input state \(x\) so that we can guarantee that \(c'\) is enabled in \(a.x\). The states following the “from” constructions contain exactly such minimal substates of \(x\). It is possible for the compiler to check that the “from” constructions are correct, in the sense that they correspond exactly to the \(\vdash_0\) enabling relation from definition 22.

Macro-generation are available through the & construction, as in:

\[
\begin{align*}
\text{valof } c & \text{ is} \\
& \&v: \text{ output } \&v \\
\text{end}
\end{align*}
\]

which tests \(c\) and outputs the value read in \(c\). This instruction being very common, it is abbreviated by:
transmit c

The following examples are the identity algorithm and the first projection:

let IDENTITY:* → * =
  algo
    request &c do
      transmit &c
    end
  end

let FST:*1#*2 → *1 =
  algo
    request &c do
      transmit &c.1
    end
  end

The conversion of sequential algorithms from M to M’ into states of the
dcds [M →₇ M'] is easily performed by a compiler. The result of the
conversion is called the “internal” form of the sequential algorithm.

We shall follow the notation of [2] and write the values of [M →₇ M’] as
valof c’ and output v, instead of simply c’ and v. This convention is
essential in the case of more complicated dcds, such as [M →₇ [M →₇ M]],
where (xc’, valof c) and (xc’, output(valof c)) denote different events.

The following combinators on sequential algorithms are predefined in
CDS0:

**Fixpoint** If a : [M →₇ M], then fix(a) : M.

**Pairing** If a₁ : [M →₇ M₁] and a₂ : [M →₇ M₂] then < a₁, a₂ > : [M →₇ M₁#M₂]. (Recall that M₁#M₂ is the product).

**Currying** If a : [M₁#M₂ →₇ M], then curry(a) : [M₁ →₇ [M₂ →₇ M]].

**Uncurrying** If a : [M₁ →₇ [M₂ →₇ M]] then uncurry(a) : [M₁#M₂ →₇ M].
Their semantics (as states in the corresponding dcds) should be clear from section 3.4, except for $fix$; for details, see [1].

4.3 Operational Semantics

The mechanism of the evaluation in CDS0, as explained in [2], is based on the notion of communication agents or coroutines [9]. Consider the application $A = T.U$ where $T$ has type $[M \to_{sa} M']$ and $U$ has type $M$. Then three communication agents are generated:

1. A main agent $A$ corresponding to the application node.
2. Two subagents $T$ and $U$.

The main agent $A$ communicates with both subagents, which ignore each other. $A$ maintains a local memory containing a state $x$ of $M$, which is the finite substate of $U$ discovered so far by $A$. Initially $x_0 = \phi$. When $A$ receives a request from some agent above it (e.g. the user) about the value of some cell $c'$ in $T.U$, it asks $T$ about the content of its cell $x_0c'$: suppose it is $c_0$ (also written $valof c_0$), instructing $A$ to inspect the value of $c_0$ in $U$. After receiving the answer from $U$ that the value of $c_0$ is, say, $v_0$, $A$ updates its state to $x_1 = x_0 \cup \{(c_0, v_0)\}$, and again asks $T$ about the value of $x_1c'$, etc. As the state of $A$ increases, $x_0 \leq x_1 \leq \ldots \leq x_n$, it will eventually ask $T$ about the value of some cell $x_n c'$ for which it receives the answer $v'$; then, $A$ passes this value to the agent above it.

Two problems arise with this model of computation:

1. When the cell $c'$ is not an initial cell (i.e. $\not\in c'$ in $M'$), then the cell $\phi c'$ is not filled in $T$; recall that a cell $xc'$ is filled in a sequential algorithm $T$ only iff $c'$ is enabled in $T.x$. So when $A$ asks $T$ about the content of $\phi c'$, $T$ will give no answer, although $c'$ might be filled in $T.U$. There are two possible solutions to this problem.

   (a) The first one is based on the principle that an agent never asks about the content of some cell $c'$ until it has enough evidence that $c'$ is filled. So, before being asked about $c'$, the agent $A$ was probably asked about the values of other cells $c'_1, \ldots, c'_n$ which enable
\( c' \). Then it suffices for \( A \) to keep the state \( x \) from the previous question(s), which is the finite substate of \( U \) discovered while answering that/those question(s), and to start the actions for \( c' \) with this \( x \). This idea leads to the CDS01 operational semantics.

(b) The second solution requires restricting only to the *sequential* dcds (see section 2.1). The evaluator for these dcds is only mentioned in [5] and [2].

2. A second problem occurs for CDS01 and is that of over-information: when \( A \) starts with some state \( x \), it might not find the cell \( xc' \) filled in \( T \) because \( x \) is “too big”. Recall that \( xc' \) is filled with, say \( v' \), in \( T \) only if \( x \) is a minimal state s.t. \((c', v') \in T.x\). This problem is simpler to handle: \( T \) should answer questions about the value of \( xc' \) by returning the value of the “biggest cell” below \( xc' \), when such one exists. Note that such a “biggest cell” need not necessarily have the form \( yc' \) with \( y \leq x \): the order relation on cells must be defined inductively for higher types.

We start by defining this order relation on the cells:

**Definition 32** The cell names in DCDS may have the following form:

1. \( u \), a token from a given alphabet.
2. \( c.u \) where \( c \) is a cell name and \( u \) is a token.
3. \( xc' \) where \( x \) is a finite state (i.e. a finite set of pairs cell-value \((c, v)\), with \( c \) a cell name, and \( v \) a value) and \( c' \) is a cell name.

**Definition 33** The order relation on cell names \( \leq \) is defined by:

1. \( \leq \) is the equality on tokens.
2. \[
\frac{c_1 \leq c_2}{c_1.u \leq c_2.u}
\]
Now we define the CDS01 expressions, which are essentially CDS0 expressions decorated with states.

**Definition 34** A CDS01 expression:

1. Any CDS0 state of type \( M \).
2. \( \text{curry}(T), \text{uncurry}(T), < T_1, T_2 > \) are CDS01 expressions, where \( T, T_1, T_2 \) are CDS0 expressions.
3. If \( T, U \) are CDS01 expressions of type \( [M \rightarrow s_a M'] \) and \( M \), and if \( x \) is a state of \( M \), the \( [T.U, x] \) is a CDS01 expression of type \( M' \).
4. If \( T \) is a CDS01 expression of type \( [M \rightarrow s_a M] \) and \( x \in D(M) \), then \( [\text{fix}(T), x] \) is a CDS01 expression of type \( M \).
5. If \( T, T' \) are CDS01 expressions of type \( [M \rightarrow s_a M'] \) and \( [M' \rightarrow s_a M''] \) and \( F \) is a set of pairs \( (x, x') \) of states of \( M, M' \), then \( [T|T', F] \) is a CDS01 expression of type \( [M \rightarrow s_a M''] \).

The CDS01 semantics is a set of rules showing how to convert a question, i.e. a pair \( T_l?c \) with \( T_l \) a CDS01 expression and \( c \) a cell, into an answer, i.e. a pair \( T_2!v \) with \( T_2 \) a CDS01 expression and \( v \) a value. Here \( T_l \) and \( T_2 \) have the same corresponding CDS0 expressions, but \( T_2 \) has a richer internal state than \( T_l \), as it gains information during the evaluation process.

The rules are:

State

\[
\bar{x}(c) = v \\
\bar{x}c \rightarrow \bar{x}v
\]

where \( \bar{x}(c) \) is the value in \( x \) of the cell \( \sup\{c_1 \mid c_1 \leq c \text{ and } c_1 \in F(x)\} \).

It can be shown that the least upper bound exists.
Pairing

\[
T ? x'c \rightarrow^* T_1 ! u \\
< T, U > ? x(c'.1) \rightarrow < T_1, U > ! u
\]

\[
U ? x'c \rightarrow^* U_1 ! u \\
< T, U > ? x(c'.2) \rightarrow < T, U_1 > ! u
\]

Currying

\[
T ? (x, x')c'' \rightarrow^* T_1 ! \text{valof } c.1 \\
\text{curry}(T) ? x(x'c'') \rightarrow \text{curry}(T_1) ! \text{valof } c
\]

\[
T ? (x, x')c'' \rightarrow^* T_1 ! \text{valof } c'.2 \\
\text{curry}(T) ? x(x'c'') \rightarrow \text{curry}(T_1) ! \text{output } \text{valof } c'
\]

\[
T ? (x, x')c'' \rightarrow^* T_1 ! \text{output } v'' \\
\text{curry}(T) ? x(x'c'') \rightarrow \text{curry}(T_1) ! \text{output } \text{output } v''
\]

Uncurrying

\[
T ? x(x'c'') \rightarrow^* T_1 ! \text{valof } c' \\
\text{uncurry}(T) ? (x, x')c'' \rightarrow \text{uncurry}(T_1) ! \text{valof } c.1
\]

\[
T ? x(x'c'') \rightarrow^* T_1 ! \text{valof } c' \\
\text{uncurry}(T) ? (x, x')c'' \rightarrow \text{uncurry}(T_1) ! \text{valof } c'.2
\]

\[
T ? x(x'c'') \rightarrow^* T_1 ! \text{output } v'' \\
\text{uncurry}(T) ? (x, x')c'' \rightarrow \text{uncurry}(T_1) ! \text{output } v''
\]
Application

\[ T \cdot xc' \rightarrow^* T_1!\text{output } v' \]
\[ [T \cdot U, x] \cdot c' \rightarrow [T_1 \cdot U, x] \cdot v' \]

\[ T \cdot xc' \rightarrow^* T_1!\text{valof } c \quad U \cdot c \rightarrow^* U_1!v \]
\[ [T \cdot U, x] \cdot c' \rightarrow [T_1 \cdot U_1, x \cup \{(c, v)\}] \cdot c' \]

Fixpoint

\[ T \cdot xc \rightarrow^* T_1!\text{output } v \]
\[ [\text{fix}(T), x] \cdot c \rightarrow [\text{fix}(T_1), x] \cdot v \]

\[ T \cdot xc \rightarrow^* T_1!\text{valof } c_1 \quad [\text{fix}(T_1), x] \cdot ?c_1 \rightarrow^* [\text{fix}(T_2), y] \cdot v \]
\[ [\text{fix}(T), x] \cdot ?c \rightarrow [\text{fix}(T_2), y \cup \{(c_1, v)\}] \cdot ?c \]

Composition For \( F \) a set of pairs \((z, z')\) of states and \( x \) a state, let

\[(y, y') = \bigcup\{(z, z') / (z, z') \in F \mid z \leq x\}\]

Then:

\[ T'' \cdot yc'' \rightarrow^* T_1!\text{output } v'' \]
\[ [T'[T, F] \cdot xc'' \rightarrow [T_1'[T, F]!\text{output } v'' \]

\[ T'' \cdot yc'' \rightarrow^* T_1!\text{valof } c' \quad T \cdot yc' \rightarrow^* T_1!\text{valof } c \quad c \not\in F(x) \]
\[ [T'[T, F] \cdot xc'' \rightarrow [T_1'[T, F]!\text{valof } c \]

\[ T'' \cdot yc'' \rightarrow^* T_1!\text{valof } c' \quad T \cdot yc' \rightarrow^* T_1!\text{valof } c \quad (c, v) \in x \]
\[ [T'[T, F] \cdot xc'' \rightarrow [T_1'[T, F \cup \{(y \cup \{(c, v)\}, y')}] \cdot xc'' \]

\[ T'' \cdot yc'' \rightarrow^* T_1!\text{valof } c' \quad T \cdot yc' \rightarrow^* T_1!\text{output } v' \]
\[ [T'[T, F] \cdot xc'' \rightarrow [T_1'[T, F \cup \{(y, y' \cup \{(c', v')\})}] \cdot xc'' \]
Theorem 8 (Full abstraction) Let $T$ be a CDS0 term: call $T_0$ the CDS01 term obtained from $T$ by adding empty storage units. Then $(c,v) \in T$ iff there exists $c_0, c_1, \ldots, c_n = c, v_0, v_1, \ldots, v_n = v$ and $T_1, \ldots, T_{n+1}$ s.t.:

$$T_0 ? c_0 \rightarrow^* T_1 ! v_0$$

$$T_1 ? c_1 \rightarrow^* T_2 ! v_1$$

$$\ldots$$

$$T_n ? c_n \rightarrow^* T_{n+1} ! v$$
Chapter 5

Conclusions

This report described:

1. The notion of concrete data structures and its domain theoretic characterization.

2. The notion of sequential functions and sequential algorithms on cds, and some consequences of using sequential algorithms as models for PCF.

3. A language CDS0 for denoting cds and sequential algorithms.

Overall, this report has shown that concrete data structures provide a good concept for giving semantics to sequential programming languages. Recent work ([6]) shows that dcids and sequential algorithms provide a fully abstract model for PCF+catch, an extension of PCF with a construction which allows us to distinguish between, say, the left addition and the right addition.

As a general concept, cds lie somewhere between semantics and syntax. Both the definition of cds and that of sequential algorithms are of a syntactic flavor. The language CDS0, which is just syntax for cds and sequential algorithms, is another argument for the claim above. However, as opposed to Milners syntactic domain construction ([10]), cds and sequential algorithms admit also a fair, semantic, characterization - as concrete domains, and as abstract algorithms, respectively.

A possible further development would be to use cds as inspiration for new concepts, designed for understanding parallel programming languages. We intend to investigate this direction in the future.
Bibliography


