Continuous Functions and Parallel Algorithms on Concrete Data Structures

Stephen Brookes Shai Geva

July 1991
CMU-CS-91-160

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in Proceedings of *Mathematical Foundations of Programming Semantics*, Pittsburgh, 1991 (Springer Verlag Lecture Notes in Computer Science).

This research was supported in part by National Science Foundation grant CCR-9006064 and in part by DARPA/NSF grant CCR-8906483.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

Keywords: theory, applicative (functional) programming, semantics, parallelism, category theory

Abstract

We report progress in two closely related lines of research: the semantic study of sequentiality and parallelism, and the development of a theory of intensional semantics. We generalize Kahn and Plotkin's concrete data structures to obtain a cartesian closed category of generalized concrete data structures and continuous functions. The generalized framework continues to support a definition of sequential functions. Using this ccc as an extensional framework, we define an intensional framework — a ccc of generalized concrete data structures and parallel algorithms. This construction is an instance of a more general and more widely applicable category-theoretic approach to intensional semantics, encapsulating a notion of intensional behavior as a computational comonad, and employing the co-Kleisli category as an intensional framework. We discuss the relationship between parallel algorithms and continuous functions, and supply some operational intuition for the parallel algorithms. We show that our parallel algorithms may be seen as a generalization of Berry and Curien's sequential algorithms.

1 Introduction

In this paper we present progress in two closely related themes of research in programming language semantics. The first concerns the semantic study of sequentiality and parallelism, and the second is the development of a general theory of intensional semantics in which one may give a variety of semantics to a language, at differing levels of intensional detail, and establish natural relationships between the meanings of terms at each level.

There has been much work on the search for a semantic characterization of sequential computation. Since the conventional continuous functions semantic model contains inherently parallel functions, such as parallel-or, a suitable definition of sequential functions is a necessary pre-requisite in the search for a natural (*i.e.*, syntax- and language-independent) fully abstract semantic model for sequential programming languages such as PCF [Plo77, Mil77, BCL85, Sto88].

A general definition of sequential functions has been given by Kahn and Plotkin in the restricted setting of concrete data structures [KP78]. Berry and Curien have shown, however, that concrete data structures are not closed under any of the continuous function space, stable function space or sequential function space; as a consequence, concrete data structures do not form a ccc when the morphisms are taken to be any of the continuous functions, stable functions or sequential functions [BC82]. To date, no sequential extensional model has been found.

Our first contribution is the definition of a new class of *generalized concrete data structures*, introduced in section 2. Essentially, the generalization consists in adding a poset structure to the cells of a concrete data structure; the original Kahn-Plotkin concrete data structures correspond to cases where the cell poset is discrete. We show that generalized concrete data structures are closed under the continuous function space, and form a ccc with continuous functions as morphisms. The states of a generalized concrete data structure, ordered by set inclusion, form what we call a generalized concrete domain. Every generalized concrete domain is also a Scott domain, but the converse is false. We define distributive generalized concrete data structures, a generalization of the deterministic (or stable) concrete data structures, and we show that they form a full sub-ccc of the category of generalized concrete data structures. We also sketch the construction of a ccc of distributive gCDSs and stable functions, obtained by varying the notion of a state.

The generalized concrete data structures continue to support a definition of sequentiality, so that we have significantly expanded the setting where sequential functions may be identified. We believe that the category of generalized concrete data structures and continuous functions is the first non-trivial ccc in which one may identify the sequential functions between any two objects. The identity function on a generalized concrete data structure is sequential, and the sequential functions between generalized concrete data structures are closed under composition. We do not know yet if the set of sequential functions between two generalized concrete data structures itself forms a generalized concrete data structure, so we do not claim (yet) to have produced a satisfactory sequential extensional model.

The failure of concrete data structures to support an extensional semantic model has led Berry and Curien to define an *intensional* semantic model: a cartesian closed category of deterministic concrete data structures and sequential algorithms [BC82, Cur86]. A sequential algorithm may be seen as a sequential function paired with a sequential computation strategy.

The appeal of intensional semantics lies in making it possible to use semantic methods to reason about a broader range of properties of programs. Traditionally, the denotational semantics approach focuses on the extensional aspects of programs, and abstracts away all intensional details; other tools must be used to reason about intensional properties. By employing a different level of abstraction that retains intensional information about programs (at a level appropriate to the

task at hand), one should be able to use an intensional denotational semantics to reason about the intensional aspects of programs, such as laziness and complexity (see for instance [Col89] for a potential application).

One of our initial goals in this study has been the definition of a richer intensional semantic model by generalizing Berry and Curien's sequential algorithms between concrete data structures to parallel algorithms. Our thesis is that, by analogy with the characterization of sequential algorithms, a parallel algorithm should correspond to a continuous function paired with a parallel computation strategy. A previous attempt was our "query model" of parallel algorithms [BG90]; although this work has generated some useful insights, it was only partly successful in providing the desired generalization of sequential algorithms, since we were unable to equip this model with a satisfactory categorical structure. Our continued efforts to generalize sequential algorithms have led to the progress reported herein.

We have been able to formalize the construction of an intensional semantic framework, given an extensional semantic framework and a notion of intensional behavior [BG91]. In accordance with this approach, we use here the terms "extensional" and "intensional" as *relative* terms – they serve to identify different levels of abstraction. Category-theoretically speaking, the extensional framework is a ccc $\mathcal{C}$, the intensional behavior is defined by a computational comonad T over $\mathcal{C}$, and the derived intensional framework is the co-Kleisli category $\mathcal{C}_T$ of $\mathcal{C}$ and T. We remark that if $\mathcal{C}$ is a ccc and T preserves products then $\mathcal{C}_T$ is also a ccc [See89]. This construction is quite general, and completely divorced of the concrete data structures setting where we first observed its applicability. We believe that this approach can serve as the basis for the development of a rather general theory of intensional semantics.

However, since there is no suitable extensional ccc with concrete data structures as objects (and some class of functions as morphisms), the desired parallel generalization of sequential algorithms cannot be obtained by a direct application of the co-Kleisli construction. Instead, we move to the setting of generalized concrete data structures, which does not suffer from these limitations. Using the ccc of generalized concrete data structures and continuous functions as an extensional framework, we define in section 3 a simple and intuitive notion of intensional behavior based on the idea that a computation is a sequence of incremental evaluation steps. We encapsulate this notion in the definition of a comonad of *paths*, and we use the co-Kleisli construction to obtain a ccc of generalized concrete data structures and parallel algorithms. We discuss the relationship between the intensional and extensional categories, by showing that every algorithm determines a continuous *input-output function* and that every continuous function is the input-output function of some algorithm. This shows that a parallel algorithm may indeed be viewed as a continuous function paired with a computation strategy. Although we do not give a formal definition of the notion of computation strategy, we do supply some operational intuition.

In section 4 we show how our parallel algorithms on generalized concrete data structures generalize Berry and Curien's sequential algorithms on concrete data structures. We define an embedding function that takes each Berry-Curien algorithm to its analogue in our model, which may be thought of as a degenerate parallel algorithm that operates sequentially.

We conclude by outlining a number of directions for further work.

In this paper we do not present the details behind the co-Kleisli construction and the related category-theoretic development. Instead we focus directly on the specific case at hand. For an exposition in more general terms, with full explanations of the relevant category-theoretic definitions and results, we refer the reader to [BG91], which also contains a detailed exploration of the relationships between extensional and intensional semantic models that may be defined within the frameworks described here.

2 Generalized Concrete Data Structures

Definition 2.1 A *Generalized Concrete Data Structure* or gCDS $(\mathrm{C}, \mathrm{V}, \mathrm{E}, \vdash)$ consists of

- A countable poset $(\mathrm{C}, \leq)$ of *cells*.
- A countable set V of *values*.
- A set $\mathrm{E} \subseteq \mathrm{C} \times \mathrm{V}$ of *events*.

  The set of events must be upwards-closed with respect to the cell ordering: if $(c, v) \in \mathrm{E}$ and $c \leq c'$ then $(c', v) \in \mathrm{E}$.
- An *enabling* relation $\vdash$ between finite sets of events and cells.

  The enabling relation must be upwards-closed with respect to the cell ordering: if $y \vdash c$ and $c \leq c'$ then $y \vdash c'$.

  The enabling relation defines a *precedence* relation $\ll$ over cells: $c \ll c'$ iff $y \cup \{(c, v)\} \vdash c'$ for some v and y. We require that the precedence relation be well-founded.

Let M, M', *etc.*, denote gCDSs in the following discussion. •

We say that a cell c is *filled* in a set y of events iff $(c, v) \in y$ for some v; we write $\mathrm{F}(y)$ for the cells filled in y. If $y' \vdash c$ we say that y' is an enabling of c. A cell c is *enabled* in y iff there exists an enabling $y' \subseteq y$ of c. We write this as $y' \vdash_y c$, and we let $\mathrm{E}(y)$ be the set of cells enabled in y. A cell is *accessible* from y iff it is enabled in y but not filled; we let $\mathrm{A}(y) = \mathrm{E}(y) \setminus \mathrm{F}(y)$. A cell is *initial* iff it is enabled by the empty set of events.

Definition 2.2 A *state* of M is a set $x \subseteq \mathrm{E}$ with the following three properties:

- Functional: if $(c, v_1), (c, v_2) \in x$ then $v_1 = v_2$.
- Safe: every cell filled in x has an enabling in x.
- Upwards-closed with respect to the cell ordering: if $(c, v) \in x$ and $c \leq c'$ then $(c', v) \in x$.

  Equivalently, this property may be stated as the requirement that $x = \mathsf{up}(x)$, where up is the upwards-closure operation over sets of events:

$$\mathsf{up}(x) = \{(c', v) \mid \exists c \leq c' \,.\, (c, v) \in x\}\,.$$

We write $\mathcal{D}(M)$ for the poset of states of M, ordered by set inclusion. We say that this is the domain generated by M. We refer to the domains generated by generalized concrete data structures as *generalized concrete domains*. •

Example 2.3 The gCDS `Null` has no cells, values, events or enablings. It has a single state, the empty set.

The gCDS `Two` has a single cell $*$, which is initial and may be filled with the value $*$. It generates (a domain isomorphic to) the two point domain, with states $\emptyset$ and $\top = \{(*, *)\}$.

The gCDS `Bool` has a single cell `b`, which is initial and may be filled with either of the values `tt` or `ff`. It generates (a domain isomorphic to) the usual boolean domain, with states $\emptyset$, $\{(\mathtt{b}, \mathtt{tt})\}$ and $\{(\mathtt{b}, \mathtt{ff})\}$.

The gCDS **Vnat** has the natural numbers as cells, ordered discretely. Each cell may be filled with the value $*$. The cell 0 is initial, and for every k, $\{(k,*)\} \vdash k+1$. The domain $\mathcal{D}(\mathtt{Vnat})$ is isomorphic to the vertical ordering of the natural numbers (*i.e.*, $n < n+1$), with a limit point added at infinity: an integer n corresponds to the state $\{(k,*) \mid k < n\}$, and ω corresponds to the state $\{(k,*) \mid k \in \mathbb{N}\}$. We may use the integers and ω to denote the states of **Vnat**. •

2.1 Generalized Concrete Domains

We now give a partial domain-theoretic characterization of the generalized concrete domains.

Proposition 2.4 *Generalized concrete domains are Scott domains, i.e., consistently complete, directed complete ω-algebraic posets. The empty set is the least element, and the lub of an upper-bounded or directed set of states is given by its union. The finite elements (i.e., isolated elements) of a generalized concrete domain are states that are the upwards-closure of some finite set of events.*

Not all Scott domains are generalized concrete domains. This is because all generalized concrete domains have property ($Q_!$), the uniqueness part of property (Q) enjoyed by CDSs [KP78].

For x and y elements of a domain D, we say that y covers x iff $x < y$ and there is no z such that $x < z < y$. We say that a domain D has property ($Q_!$) iff:

($Q_!$) For any $x, y, z_1, z_2 \in D$, if y, z_1 and z_2 cover x, y is inconsistent with both of z_1 and z_2, and z_1 and z_2 are consistent, then $z_1 = z_2$.

Proposition 2.5 *Every generalized concrete domain has property ($Q_!$).*

Proof: For a generalized concrete domain $\mathcal{D}(M)$, if x is covered by y then there exists some event (c,v) (with c maximal in the cell ordering) such that $y = x \cup \{(c,v)\}$.

Moreover, if x is covered by y and z, and y and z are inconsistent, then $y = x \cup \{(c,v)\}$ and $z = x \cup \{(c,v')\}$ for some c (again, maximal) and $v \neq v'$.

It follows that $\mathcal{D}(M)$ has property ($Q_!$). ▪

2.2 The Continuous Functions Category

We define the category **gCDScont** with gCDSs as objects and continuous functions between $\mathcal{D}(M)$ and $\mathcal{D}(M')$ as the morphisms between M and M'. Composition is taken to be function composition, and the identity morphisms are just the identity functions. An equivalent category is the category of generalized concrete domains and continuous functions, a full sub-category of the category of Scott domains and continuous functions. We now show that **gCDScont** is cartesian closed.

The gCDS **Null** is a terminal object in **gCDScont**.

The product construction is a straightforward generalization of the product for concrete data structures [Cur86]. We write $c.i$ for the pair (c,i), where c is a cell and i is a tag – we use 1 and 2 as tags for the product. For a set of cells C and a set of events y, we write $C.i$ and $y.i$ for $\{c.i \mid c \in C\}$ and $\{(c.i,v) \mid (c,v) \in y\}$, respectively. We build the product of two gCDSs by taking a disjoint union of the two posets of cells, of the two sets of events, and of the two enabling relations.

Definition 2.6 The product of two gCDSs M_1 and M_2 is defined by:

- $\mathrm{C}_{M_1 \times M_2} = \mathrm{C}_{M_1}.1 \cup \mathrm{C}_{M_2}.2$, ordered by: $c.i \leq_{M_1 \times M_2} c'.i'$ iff $c \leq_{M_i} c'$ and $i = i'$.
- $\mathrm{V}_{M_1 \times M_2} = \mathrm{V}_{M_1} \cup \mathrm{V}_{M_2}$.

- $\mathrm{E}_{M_1 \times M_2} = \mathrm{E}_{M_1}.1 \cup \mathrm{E}_{M_2}.2$.
- $y.i \vdash_{M_1 \times M_2} c.i$ iff $y \vdash_{M_i} c$.

•

Proposition 2.7 *The product is well defined, i.e., events and enablings are upwards closed, and countability and well foundedness are preserved.*

We define pairing and projections. For $x \in \mathcal{D}(M_1 \times M_2)$ and $x_i \in \mathcal{D}(M_i)$ for $i = 1, 2$,

$$\begin{aligned} \langle x_1, x_2 \rangle &= x_1.1 \cup x_2.2 \\ \pi_i(x) &= \{(c, v) \mid (c.i, v) \in x\}. \end{aligned}$$

Proposition 2.8 *The domain $\mathcal{D}(M_1 \times M_2)$ is isomorphic to $\mathcal{D}(M_1) \times \mathcal{D}(M_2)$ (ordered componentwise).*

Corollary 2.9 *The gCDS product is a categorical product in* **gCDScont**.

Definition 2.10 Given two gCDSs M and M', we define the gCDS $M \multimap M'$ by:

- $\mathrm{C}_{M \multimap M'} = \mathcal{D}_{\mathrm{fin}}(M) \times \mathrm{C}_{M'}$ where $\mathcal{D}_{\mathrm{fin}}(M)$ consists of the finite elements of $\mathcal{D}(M)$, ordered by inclusion, and the poset product is ordered componentwise.

  We use juxtaposition for the cells of an exponentiation, writing xc' for the cell (x, c').
- $\mathrm{V}_{M \multimap M'} = \mathrm{V}_{M'}$.
- $\mathrm{E}_{M \multimap M'} = \{(xc', v') \in \mathrm{C}_{M \multimap M'} \times \mathrm{V}_{M \multimap M'} \mid (c', v') \in \mathrm{E}_{M'}\}$.
- $\{(x_j c'_j, v'_j) \mid 1 \leq j \leq l\} \vdash_{M \multimap M'} xc'$ iff $\{(c'_j, v'_j) \mid 1 \leq j \leq l\} \vdash_{M'} c'$ and $\forall j \leq l,\ x_j \subseteq x$.

•

Essentially, the cells of $M \multimap M'$ are cells of M' tagged with (finite) information about the input, represented as a finite state of M. The enabling relation ensures the appropriate combination of this input information. There is a very close correspondence between our definition of $M \multimap M'$ for gCDSs and the extensional components (the *output* values) of Berry and Curien's sequential algorithms exponentiation of CDSs [Cur86] (see the definition in section 4).

Proposition 2.11 *For all gCDSs M and M', $M \multimap M'$ is well defined: i.e., events and enablings are upwards closed, and countability and well foundedness are preserved.*

Proposition 2.12 *The domain $\mathcal{D}(M \multimap M')$ is isomorphic to the continuous function space between $\mathcal{D}(M)$ and $\mathcal{D}(M')$, ordered pointwise.*

The isomorphism is given, for $a \in \mathcal{D}(M \multimap M')$ and $f : \mathcal{D}(M) \multimap \mathcal{D}(M')$, by:

$$\begin{aligned} a &\mapsto \lambda z \in \mathcal{D}(M).\ \{(c', v') \mid \exists x \subseteq z.\ (xc', v') \in a\} \\ f &\mapsto \{(xc', v') \in \mathrm{E}_{M \multimap M'} \mid (c', v') \in f(x)\}. \end{aligned}$$

Given the isomorphism, it is clear that the morphisms from M to M' may equivalently be taken to be the states of $M \multimap M'$. Since application is continuous and currying is well behaved, it is clear that $M \multimap M'$ is in fact an exponentiation object for M and M' in the category **gCDScont**.

Corollary 2.13 **gCDScont** *is a cartesian closed category.*

Example 2.14 The gCDS `Vnat` $\multimap$ `Two` has cells $\{n* \mid n \in \mathbb{N}\}$, ordered vertically (*i.e.*, $n* < n+1*$). Each cell may be filled with the value $*$. The isomorphism between $\mathcal{D}(\texttt{Vnat} \multimap \texttt{Two})$ and the continuous function space from $\mathcal{D}(\texttt{Vnat})$ to $\mathcal{D}(\texttt{Two})$ may be easily discerned. •

2.3 Distributive gCDSs

Definition 2.15 We extend the cell ordering to events as follows: $(c,v) \leq (c',v')$ iff $c \leq c'$ and $v = v'$. We then extend the ordering to finite sets of events by: $y \leq y'$ iff there exists a bijection $\phi : y \rightarrow y'$ such that $(c,v) \leq \phi(c,v)$, for all $(c,v) \in y$. •

Definition 2.16 A gCDS M is *distributive* iff for all states x of M, if $y_1 \vdash_x c$ and $y_2 \vdash_x c$ then there exists y such that $y_i \leq y$, for $i = 1,2$. •

This property is a generalization of the *stability* and *determinism* properties of concrete data structures [Cur86], and similar results follow. We recall that Berry and Curien's sequential algorithms model was limited to deterministic CDSs.

Proposition 2.17 *If M is distributive then the glb in $\mathcal{D}(M)$ of any two consistent states is their intersection.*

Proof: Let M be a distributive gCDS, and let x_1 and x_2 a consistent pair of states of M. Clearly $x_1 \cap x_2$ is their glb as sets of events, so we only need to show that it is a state. The intersection clearly preserves functionality and upwards-closure. To show that it preserves safety, let $c \in \mathrm{F}(x_1 \cap x_2)$. For $i = 1,2$, $c \in \mathrm{F}(x_i)$, so that there exists $y_i \vdash_{x_i} c$, and therefore $y_i \vdash_{x_1 \cup x_2} c$. Now, by distributivity there exists y such that, for $i = 1,2$, $y_i \leq y$, and by upwards closure $y \vdash_{x_i} c$, and, finally, $y \vdash_{x_1 \cap x_2} c$. It follows that $x_1 \cap x_2$ is a state. ▪

Definition 2.18 A consistently-complete poset is *distributive* iff for all x and all consistent pairs x_1 and x_2, $(x \wedge x_1) \vee (x \wedge x_2) = x \wedge (x_1 \vee x_2)$. •

Proposition 2.19 *If M is distributive then $\mathcal{D}(M)$ is distributive.*

Proof: An immediate corollary of 2.17. ▪

Proposition 2.20 *The category of distributive gCDSs and continuous functions is a full subcategory of* **gCDScont**, *and it is cartesian closed.*

Proof: `Null` is distributive and product and exponentiation preserve distributivity.

To see that the exponentiation preserves distributivity, let $a \in \mathcal{D}(M \multimap M')$ and assume that, for $i = 1,2$, $\{(x_{ij}c'_{ij}, v'_{ij}) \mid 1 \leq j \leq l_i\} \vdash_a xc'$; then, for $i = 1,2$, $\{(c'_{ij}, v'_{ij}) \mid 1 \leq j \leq l_i\} \vdash_{a(x)} c'$, where $a(x) \in \mathcal{D}(M')$ is the value on x of the continuous function corresponding to a (by the isomorphism of proposition 2.12). By distributivity of M' there must be an enabling $\{(c'_j, v'_j) \mid 1 \leq j \leq l\} \vdash_{a(x)} c'$, where $l = l_1 = l_2$, and for $j \leq l$, $v'_j = v'_{ij}$ and $c'_{ij} \leq c'_j$ – without loss of generality assume that the bijections are identities. But now $\{(xc'_j, v'_j) \mid 1 \leq j \leq l\}$ serves as an upper-bound in the extended cell ordering of the two enablings of xc' in a, so that we may conclude that $M \multimap M'$ is distributive. ▪

2.4 Relationship to Original Definition

Proposition 2.21 *Kahn and Plotkin's original definition of concrete data structures and their generated domains [KP78, Cur86] can be obtained by considering gCDSs with a discrete cell ordering (i.e., $c \leq c'$ iff $c = c'$).*

Proof: Under the discreteness assumption all upwards-closure requirements are vacuously satisfied, and our definition collapses to the original definition. ∎

Note that the gCDS product preserves discreteness: if the cells of M_1 and M_2 are ordered discretely, then so are the cells of their product. As a corollary, the gCDS product is a conservative extension of CDS product.

Importantly, the exponentiation does not preserve discreteness; even if M and M' are discrete, $\mathcal{D}_{\mathsf{fin}}(M)$ will not be discrete in general, so that the cells of the exponentiation will not be ordered discretely. This is of course necessary for our purposes, since (discrete generalized) concrete data structures are not closed under continuous function space.

The intuition behind the introduction of an ordering on cells may perhaps be explained thus: in the concrete data structures setting, a cell corresponds to a flat domain – a choice between a number of (mutually inconsistent) ways to increase information. An appropriate domain may be "decomposed" into such atomic choices. The notion of cell may itself be seen as a generalization of an argument position, the notion used by early approaches to defining sequentiality [Vui73, Mil77]. Once we introduce an ordering on cells, it is possible to talk not only of a discrete choice between alternatives for a given cell, but also of the extent to which the choice must be pursued. This seems to be essential if higher-order domains are to be represented using this approach to decomposition.

The concrete domains are the domains generated by concrete data structures (or, equivalently, discrete generalized concrete data structures). Kahn and Plotkin's representation theorem characterizes concrete domains as Scott domains satisfying a number of axioms. In particular, concrete domains satisfy axiom (I):

(I) Every finite element dominates finitely many elements.

But the continuous function space does not, in general, preserve property (I), and this is the key to Berry and Curien's proof that concrete domains and continuous functions do not form a cartesian closed category [BC82]. Our generalization of concrete domains must not, in general, satisfy axiom (I). See example 2.14 for a continuous function space and a gCDS that violate (I).

2.5 Stable Functions on gCDSs

We have concentrated so far on continuous functions, and defined a ccc **gCDScont** of gCDSs and continuous functions. Other classes of functions may be considered, by varying the definitions of a state and the domain generated by a gCDS. We will now introduce the category **gCDSstab** of distributive gCDSs and *stable functions*, a full sub-ccc of dI-domains and stable functions.

First, a few definitions are needed.

Definition 2.22 A function f between two domains D and E is said to be *stable* iff it is continuous and for every $d \in D$ and $e \leq f(d)$ the set $\{d' \mid d' \leq d \;\&\; e \leq f(d')\}$ has a least element, denoted $\mathsf{M}(f,d,e)$.

For stable functions f, g from D to E, f is below g in the stable ordering iff f is pointwise below g and, for each $d \in D$ and $e \leq f(d)$, $\mathsf{M}(f,d,e) = \mathsf{M}(g,d,e)$. ∎

A dI-domain is a distributive Scott domain that has property (I). It is well known that the category of dI-domains and stable functions is a ccc. See [Ber78] for a fuller treatment, as well as alternative (but equivalent) definitions of stability and the stable ordering.

We qualify the states introduced so far as being **ct**-states, and use $\mathcal{D}^{\mathbf{ct}}(M)$ for the domain of **ct**-states of M, ordered by set inclusion – we call this the **ct**-domain generated by M. (In particular,

our partial domain-theoretic characterization of generalized concrete domains only applies to the **ct**-domains). We now define the "stable states" of a gCDS.

Definition 2.23 A **st**-*state* of M is a set of events $x \subseteq E_M$ with the following three properties:

- Functional: if $(c, v_1), (c, v_2) \in x$ then $v_1 = v_2$.
- Safe: every cell filled in x has an enabling in x.
- Stable: if c_1 and c_2 are filled in x and c_1 and c_2 have an upper bound in the cell ordering, then $c_1 = c_2$.

Let $\mathcal{D}^{\mathbf{st}}(M)$ be the domain of **st**-states of M, ordered by set inclusion – we say that this is the **st**-domain generated by M. •

The difference between **ct**-states and **st**-states amounts to the replacement of the upwards-closure requirement of **ct**-states by a "stability" condition.

Proposition 2.24 *For a distributive M, $\mathcal{D}^{\mathbf{st}}(M)$ is a dI-domain. The empty set is the least element, and the lub of an upper-bounded or directed set of states is given by its union. The finite elements (i.e., isolated elements) are states that are finite sets of events.*

We now emulate the development carried out above for **gCDScont**.

Definition 2.25 The category **gCDSstab** has distributive gCDSs as objects, and the stable functions from $\mathcal{D}^{\mathbf{st}}(M)$ to $\mathcal{D}^{\mathbf{st}}(M')$ as the morphisms from M to M'. •

Proposition 2.26 *The product of gCDSs defined above is a product in* **gCDSstab**, *and* `Null` *is a terminal object.*

Proposition 2.27 *The definition of $M \multimap M'$ given above, modified so that $\mathcal{D}^{\mathbf{st}}{}_{\mathrm{fin}}(M)$ is used instead of $\mathcal{D}^{\mathbf{ct}}{}_{\mathrm{fin}}(M)$ in constructing the cells of $M \multimap M'$, produces an exponentiation in* **gCDSstab**. *There is an isomorphism between $\mathcal{D}^{\mathbf{st}}(M \multimap M')$ and the stable function space between $\mathcal{D}^{\mathbf{st}}(M)$ and $\mathcal{D}^{\mathbf{st}}(M')$, ordered by the* stable *ordering. The category* **gCDSstab** *is a full sub-ccc of the ccc of dI-domains and stable functions.*

Seen from a different angle, inclusion on **st**-states corresponds to the stable ordering on stable functions, while inclusion on **ct**-states corresponds to the pointwise ordering on continuous functions.

Note that for discrete gCDSs the stability requirement is vacuously satisfied, as is the upwards-closure requirement, so that the **ct**-domain $\mathcal{D}^{\mathbf{ct}}(M)$ and the **st**-domain $\mathcal{D}^{\mathbf{st}}(M)$ coincide when M is a discrete gCDS. The two notions diverge, however, on gCDSs with non-trivial cell ordering (such as exponentiations). Moreover, the classes of (distributive) **ct**-domains and **st**-domains are incomparable – we have shown that property (I) holds for **st**-domains, but is violated by **ct**-domains; on the other hand, property ($Q_!$) holds for **ct**-domains, but not for **st**-domains.

Example 2.28 For an example of a **st**-domain which violates ($Q_!$), consider $\mathcal{D}^{\mathbf{st}}(\mathtt{Bool} \multimap \mathtt{Two})$, shown in figure 1. This example is also used by Berry and Curien [BC82] to show that (deterministic) CDSs are not closed under the stable function space. Contrast this **st**-domain with the **ct**-domain $\mathcal{D}^{\mathbf{ct}}(\mathtt{Bool} \multimap \mathtt{Two})$, shown in figure 2. •

We will not delve deeper here into the category **gCDSstab**, and we will consider exclusively **gCDScont** and **ct**-domains in the remainder of the development. However, we point out that most of the ensuing development may be carried out with **gCDSstab** replacing **gCDScont** as the extensional framework.

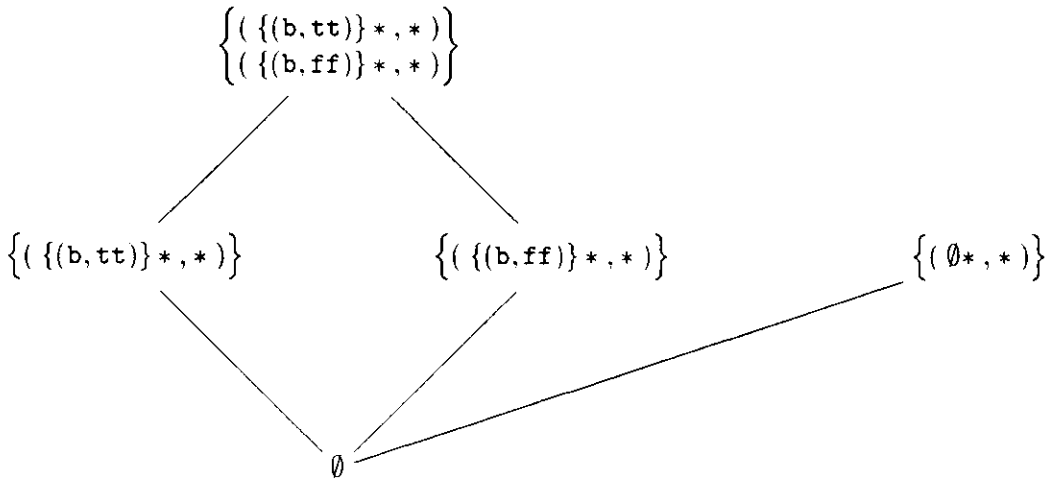

Figure 1: Hasse diagram of $\mathcal{D}^{\mathtt{st}}(\mathtt{Bool} \to \mathtt{Two})$

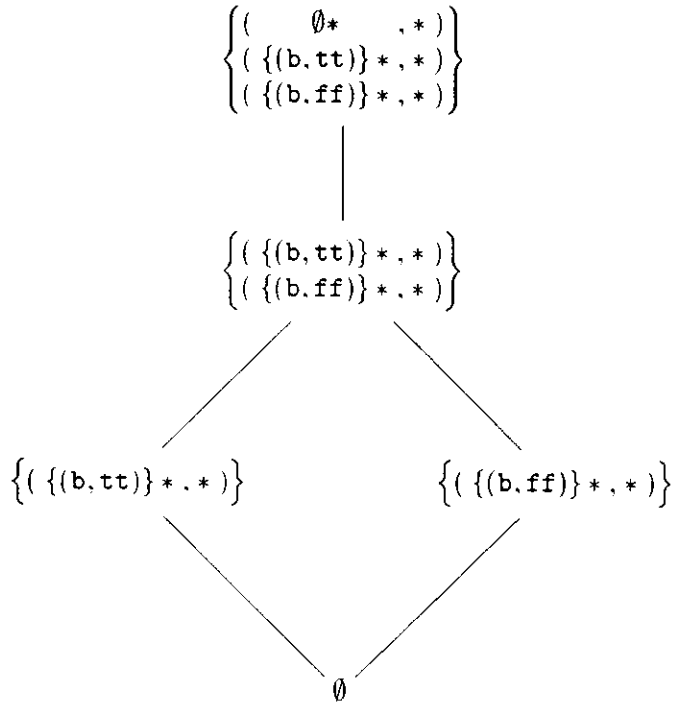

Figure 2: Hasse diagram of $\mathcal{D}^{\mathtt{ct}}(\mathtt{Bool} \to \mathtt{Two})$

2.6 Sequential Functions on gCDSs

Kahn and Plotkin's definition of sequential functions between CDSs may be stated unchanged for gCDSs, since one may still use the cells to define sequentiality indices:

Definition 2.29 A function $f : \mathcal{D}(M) \to \mathcal{D}(M')$ is *sequential* at $x \in \mathcal{D}(M)$ for $c' \in \mathrm{A}(f(x))$ iff $\mathrm{A}(x) = \emptyset$ or there exists $c \in \mathrm{A}(x)$ such that c is filled in all supersets y of x for which c' is filled in $f(y)$. In this case c is said to be an *index of sequentiality* of f at x for c'.

f is *sequential* iff it is continuous and it is sequential at all $x \in \mathcal{D}(M)$, for all $c' \in \mathrm{A}(f(x))$. •

While this definition still makes sense in our generalized setting, it remains to be shown that our class of sequential functions is well behaved. Some elementary properties are easy to establish:

Proposition 2.30 *For all gCDSs M, the identity function on M is sequential. The composition of two sequential functions between gCDSs is again sequential.*

Many more properties remain to be checked. We are currently investigating whether the set of sequential functions between two gCDSs itself forms (the states of) a gCDS. If so, we might finally obtain a ccc of gCDSs and sequential functions. Even if this fails with the above definition of sequentiality, we may be able to generalize the definition to take more explicit account of the cell ordering (while collapsing onto the original definition when the cell ordering is discrete). We are also trying to discover whether, following the general approach exemplified by the construction of **gCDScont** and **gCDSstab**, one may define a notion of "sequential state" and use this to generate a third kind of domain from a gCDS, ideally to yield a class of domains closed under the sequential function space.

3 Parallel Algorithms on Generalized Concrete Data Structures

In this section we present the category **gCDSalg** of gCDSs and parallel algorithms, using *paths* as a notion of intensional behavior with respect to **gCDScont**. We do not present the construction in its full generality – this may be found in [BG91], where a similar construction is carried out over the category of Scott domains and continuous functions.

3.1 Paths

Definition 3.1 Given a gCDS M, we define the *path gCDS* PM to be $\mathtt{Vnat} \multimap M$, and we refer to $\mathcal{D}(PM)$ as the domain of *paths over* M. •

The path domain over M is isomorphic to the continuous function space from `Vnat` to M, ordered pointwise. Yet another equivalent way of viewing paths is as infinite non-decreasing sequences of states of M, ordered componentwise. We work freely with the different representations of paths, omitting explicit mention of the isomorphisms. We write, *e.g.*, ti for the application of (the function corresponding to) the path t to (the state of `Vnat` corresponding to) the integer i, leaving the various isomorphisms implicit.

We will use paths over M to represent computations over M. Events are regarded as quanta of information produced by the computation, so that ti is the information known about the computed value by time point $i+1$, starting with no information at all at time point 0. The ordering of paths may be viewed as comparing paths by their *eagerness*: $t \subseteq t'$ iff for every i, $ti \subseteq t'i$, *i.e.*, for every $(c, v) \in ti$ we also have $(c, v) \in t'i$. Informally, $t \subseteq t'$ if the computation represented by t' computes everything that t computes, and each event in t occurs no later than it does in t'.

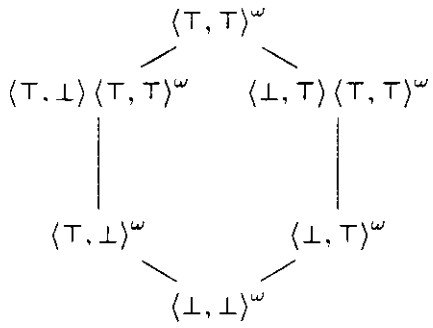

Figure 3: A partial Hasse diagram of $\mathcal{D}(P(\mathtt{Two} \times \mathtt{Two}))$

Example 3.2 Figure 3 presents a partial Hasse diagram of paths over $\mathtt{Two} \times \mathtt{Two}$. We present paths as infinite sequences of states, using the notation x^ω for a constant x suffix. •

Definition 3.3 We complete P to a functor P : **gCDScont** — **gCDScont** by setting $Pf = \mathsf{map}\, f$ for any continuous function $f : M \text{ — } M'$, where $\mathsf{map}\, f : PM \text{ — } PM'$ is defined by

$$\mathsf{map}\, f = \lambda t \in \mathcal{D}(PM)\ .\ \lambda i\ .\ f(ti).$$

It is easy to verify that $\mathsf{map}\, f$ and map itself are continuous. •

Proposition 3.4 *The functor P preserves product in* **gCDScont**, *i.e., $P(M_1 \times M_2)$ is a product, with projections $\mathsf{map}\, \pi_i$ for $i = 1, 2$. We therefore obtain the following natural isomorphisms:*

$$\begin{array}{rcl} \mathsf{split}_{M_1,M_2} & : & P(M_1 \times M_2) \text{ — } PM_1 \times PM_2 \\ \mathsf{split}_{M_1,M_2} & = & \langle \mathsf{map}\, \pi_1, \mathsf{map}\, \pi_2 \rangle \\ \mathsf{merge}_{M_1,M_2} & : & PM_1 \times PM_2 \text{ — } P(M_1 \times M_2) \\ \mathsf{merge}_{M_1,M_2} & = & \lambda t \in \mathcal{D}(PM_1 \times PM_2)\ .\ \lambda i\ .\ \langle (\pi_1\, t)i, (\pi_2\, t)i \rangle\ . \end{array}$$

In other words, there is a uniform way of converting back and forth between a pair of computations and a computation of a pair.

Definition 3.5 For each M, define the following:

$$\begin{array}{rcl} \mathsf{val}_M & = & \lambda t \in \mathcal{D}(PM)\ .\ \bigcup \{ti \mid i \in \mathcal{D}(\mathtt{Vnat})\} \\ \mathsf{pre}_M & = & \lambda t \in \mathcal{D}(PM)\ .\ \lambda i\ .\ \lambda j\ .\ t \min(i, j) \\ \mathsf{path}_M & = & \lambda x \in \mathcal{D}(M)\ .\ \lambda i\ .\ x. \end{array}$$

•

Intuitively, $\mathsf{val}_M\, t$ is the value computed by t; $\mathsf{pre}_M\, t$ is the computation built from the *prefixes* of t; and $\mathsf{path}_M\, x$ is the constant path to x, regarded as a canonical "degenerate" computation of x.

Proposition 3.6 val_M, pre_M *and* path_M *are continuous functions, and* $\mathsf{val} : P \dot{\to} Id$, $\mathsf{pre} : P \dot{\to} P^2$ *and* $\mathsf{path} : Id \dot{\to} P$ *are natural transformations. The following identities hold:*

$$\begin{array}{rcl} (\mathsf{map}\, \mathsf{pre}_M) \circ \mathsf{pre}_M & = & \mathsf{pre}_{PM} \circ \mathsf{pre}_M \\ \mathsf{val}_{PM} \circ \mathsf{pre}_M & = & \mathsf{id}_{PM} \\ (\mathsf{map}\, \mathsf{val}_M) \circ \mathsf{pre}_M & = & \mathsf{id}_{PM} \\ \mathsf{val}_M \circ \mathsf{path}_M & = & \mathsf{id}_M \\ \mathsf{path}_{PM} \circ \mathsf{path}_M & = & \mathsf{pre}_M \circ \mathsf{path}_M\ . \end{array}$$

The first three identities assert that $(P, \mathsf{val}, \mathsf{pre})$ is a comonad over **gCDScont**. The remaining two assert that $(P, \mathsf{val}, \mathsf{pre}, \mathsf{path})$ is a computational comonad (in the sense of [BG91]).

An additional inequality that stems from the choice of canonical computations is

$$\mathsf{id}_{PM} \leq \mathsf{path}_M \circ \mathsf{val}_M .$$

3.2 The Category of Algorithms

Definition 3.7 The category **gCDSalg** has gCDSs as objects, and continuous functions from PM to M' as the morphisms from M to M'. We define an *algorithm* to be a morphism in this category, and we will write $M \Rightarrow M'$ for the set of all algorithms from M to M'. Composition of algorithms $a : M \Rightarrow M'$ and $a' : M' \Rightarrow M''$, written $a' \bar{\circ} a$, is defined by:

$$a' \bar{\circ} a = a' \circ (\mathsf{map}\, a) \circ \mathsf{pre}_M .$$

The identity algorithm from M to M is val_M. •

In words, the algorithm composition of a and a' applies a' to the computation produced by mapping a over the prefixes of the argument computation. The identity algorithm disregards everything except the value computed by its argument, and it returns this value.

Algorithm composition is a continuous function on algorithms. It is straightforward to verify, using the algebraic identities given earlier, that **gCDSalg** is indeed a category: with these definitions composition is associative and the identity algorithm is a unit for composition. In fact, **gCDSalg** is just the co-Kleisli category of **gCDScont** and the comonad $(P, \mathsf{val}, \mathsf{pre})$ [ML71, BG91]. For clarity and ease of comparison with the underlying category, we use $\Rightarrow$ for morphisms in **gCDSalg**, and we retain $\rightarrow$ for morphisms in **gCDScont**.

Since **gCDScont** has finite products, it is easy to show that:

Proposition 3.8 *The algorithms category* **gCDSalg** *has finite products. If $M_1 \times M_2$ is a product in* **gCDScont** *with projections π_i $(i = 1, 2)$, then $M_1 \times M_2$ is a product in* **gCDSalg** *with projection algorithms given by*

$$\hat{\pi}_i = \pi_i \circ \mathsf{val}_{M_1 \times M_2} .$$

Equivalently, by naturality of val *and the definition of* split,

$$\hat{\pi}_i = \mathsf{val}_{M_i} \circ \pi_i \circ \mathsf{split}_{M_1, M_2} .$$

The pairing of algorithms $a_1 : M \Rightarrow M_1$ and $a_2 : M \Rightarrow M_2$, denoted $\langle a_1, a_2 \rangle : M \Rightarrow M_1 \times M_2$, is their pairing as continuous functions, *i.e.*, $\langle a_1, a_2 \rangle = \lambda t \in \mathcal{D}(M)\ .\ \langle a_1 t, a_2 t \rangle$. The product of the algorithms $a_1 : M_1 \Rightarrow M_1'$ and $a_2 : M_2 \Rightarrow M_2'$, denoted $a_1 \bar{\times} a_2 : M_1 \times M_2 \Rightarrow M_1' \times M_2'$, is given by

$$a_1 \bar{\times} a_2 = \langle a_1 \bar{\circ} \hat{\pi}_1, a_2 \bar{\circ} \hat{\pi}_2 \rangle = \langle a_1 \circ \mathsf{map}\, \pi_1, a_2 \circ \mathsf{map}\, \pi_2 \rangle = (a_1 \times a_2) \circ \mathsf{split} .$$

`Null` is a terminal object in **gCDSalg**, since it is terminal in **gCDScont**.

Proposition 3.9 *The category* **gCDSalg** *has exponentiations. The exponentiation of M and M' in* **gCDSalg** *is the gCDS $M \Rightarrow M' = PM \multimap M'$, with the application algorithm given by*

$$\widehat{\mathsf{app}}_{M,M'} = \mathsf{app}_{PM,M'} \circ (\mathsf{val}_{M \Rightarrow M'} \bar{\times} \mathsf{id}_{PM}).$$

Currying and uncurrying of algorithms are given by the following continuous functions:

$$\begin{aligned} \overline{\mathsf{curry}} &= \lambda a : M_1 \times M_2 \Rightarrow M'\ .\ \mathsf{curry}(a \circ \mathsf{merge}_{M_1, M_2}) \\ \overline{\mathsf{uncurry}} &= \lambda a : M_1 \Rightarrow (M_2 \Rightarrow M')\ .\ \mathsf{uncurry}(a) \circ \mathsf{split}_{M_1 \times M_2} . \end{aligned}$$

Intuitively, algorithm application disregards the computation of the algorithm being applied, and is only concerned with the actual algorithm and the computation of its argument. Currying and uncurrying are simple adaptations of the standard currying and uncurrying operations on functions to take account of the structure of paths. The fact that P preserves product is used crucially here.

Putting these results together, we have:

Proposition 3.10 gCDSalg *is cartesian closed.*

See [BG91] for a more detailed category-theoretic treatment from which these results follow.

Example 3.11 Figure 4 presents a partial Hasse diagram of the algorithm space from $\mathtt{Two} \times \mathtt{Two}$ to $\mathtt{Two}$. We present each algorithm by its action on the paths of $\mathtt{Two} \times \mathtt{Two}$ in figure 3, a shaded circle for a result of $\top$ (with actions on other paths determined by monotonicity). We present below the operational intuition behind these algorithms. •

3.3 Relating the Categories

We define the *input-output function* $\mathsf{fun}(a)$ of an algorithm a, and the *canonical algorithm* $\mathsf{alg}(f)$ for a continuous function f. These turn out to be the morphism parts of a pair of functors between the two categories.

Definition 3.12 For any gCDSs M and M', define

$$\begin{array}{lcl} \mathsf{fun} & : & (M \Rightarrow M') \multimap (M \multimap M') \\ \mathsf{fun} & = & \lambda a : M \Rightarrow M' \,.\, a \circ \mathsf{path}_M \\ \mathsf{alg} & : & M \multimap M' \multimap M \Rightarrow M' \\ \mathsf{alg} & = & \lambda f : M \multimap M' \,.\, f \circ \mathsf{val}_M \,. \end{array}$$

•

Proposition 3.13 fun *and* alg *satisfy the following conditions:*

$$\begin{array}{rcl} \mathsf{fun}(\mathsf{id}_M) & = & \mathsf{val}_M \\ \mathsf{alg}(\mathsf{val}_M) & = & \mathsf{id}_M \\ \mathsf{fun}(a' \mathbin{\bar{\circ}} a) & = & \mathsf{fun}\, a' \circ \mathsf{fun}\, a \\ \mathsf{alg}(f' \circ f) & = & \mathsf{alg}\, f' \mathbin{\bar{\circ}} \mathsf{alg}\, f. \end{array}$$

Thus, fun *and* alg *are the morphism parts of a pair of functors* fun : **gCDSalg** $\multimap$ **gCDScont** *and* alg : **gCDScont** $\multimap$ **gCDSalg**, *each of which is just the identity on objects (in the respective category).*

Proposition 3.14 *For every* $f : M \multimap M'$, $\mathsf{fun}(\mathsf{alg}\, f) = f$.

Thus, $\mathsf{alg}\, f$ has f as its input-output function, and every continuous function between gCDSs is the input-output function of some algorithm between gCDSs.

Definition 3.15 For $a_1, a_2 : M \Rightarrow M'$, we say that a_1 *input-output approximates* a_2, written $a_1 \leq^{io} a_2$ iff $\mathsf{fun}\, a_1$ pointwise approximates $\mathsf{fun}\, a_2$, *i.e.*, $\mathsf{fun}\, a_1 \leq \mathsf{fun}\, a_2$. We say that a_1 and a_2 are *input-output equivalent*, written $a_1 =^{io} a_2$, iff $\mathsf{fun}(a_1) = \mathsf{fun}(a_2)$. •

In words, two algorithms are input-output equivalent iff they have the same input-output function; this is the equivalence relation induced by the input-output approximation pre-order.

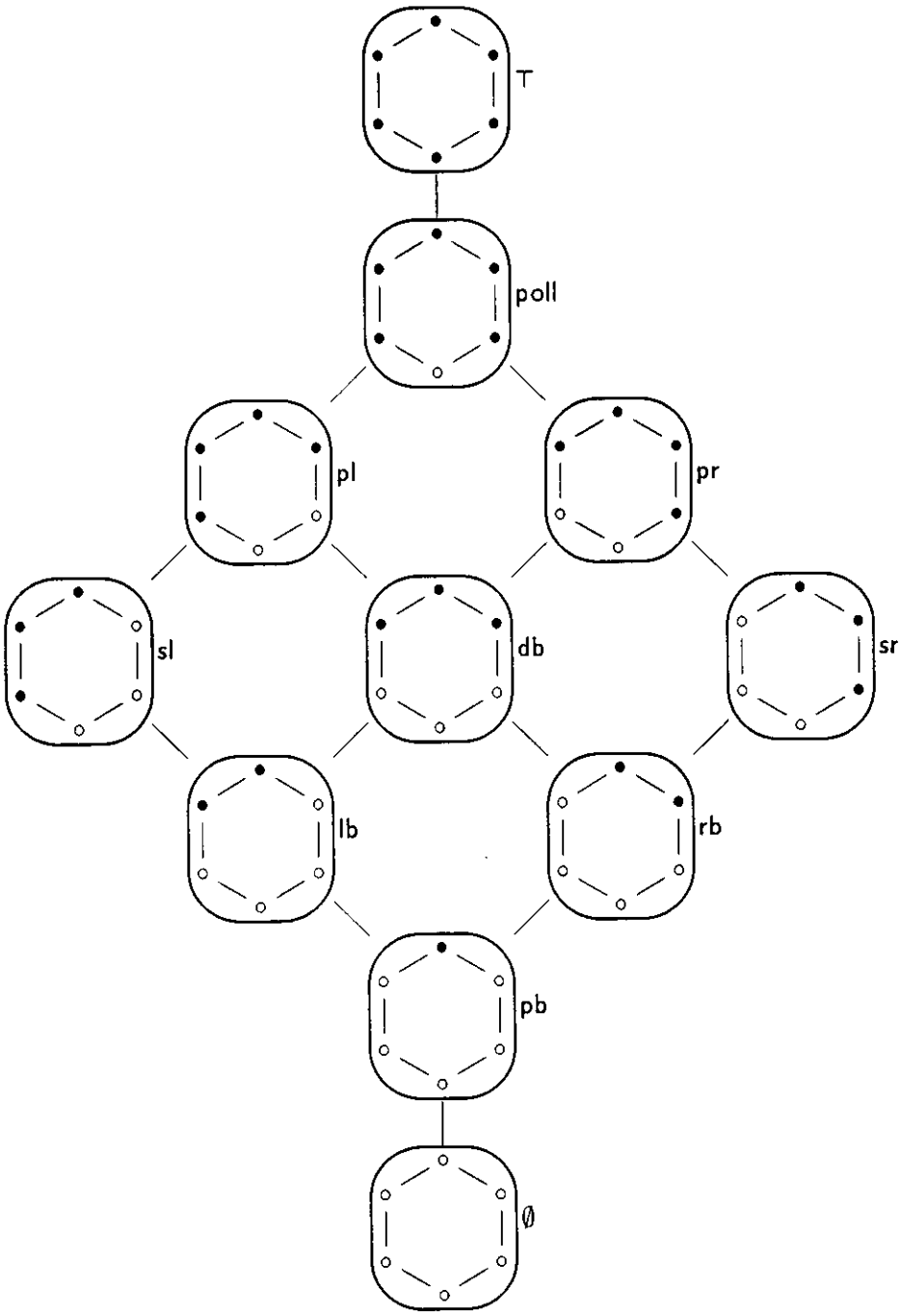

Figure 4: A partial Hasse diagram of $\mathcal{D}(\texttt{Two} \times \texttt{Two} \Rightarrow \texttt{Two})$

Proposition 3.16 *If $f_1 \leq f_2$ then* $\mathsf{alg}\, f_1 \leq \mathsf{alg}\, f_2$. *If $a_1 \leq a_2$ then $a_1 \leq^{io} a_2$, but the converse is not generally true.*

This indicates that the pointwise ordering on algorithms (as continuous functions) takes into account intensional aspects of algorithms that are disregarded by the input-output approximation ordering.

Proposition 3.17 *The quotient of the domain $\mathcal{D}(M \Rightarrow M')$ by input-output equivalence is isomorphic to the domain $\mathcal{D}(M \multimap M')$, with the isomorphism induced by* fun *and* alg*:*

$$\mathcal{D}(M \Rightarrow M')/_{=^{io}} \cong \mathcal{D}(M \multimap M').$$

Consider now the input-output equivalence class of algorithms that share an input-output function f. Since $\mathsf{id}_{PM} \leq \mathsf{path}_M \circ \mathsf{val}_M$ we have $\mathsf{id}_{M \Rightarrow M'} \leq \mathsf{alg} \circ \mathsf{fun}$. That is, the canonical algorithm $\mathsf{alg}\, f$ is maximal among the algorithms with input-output function f. Intuitively this means that the canonical algorithm is the "laziest" algorithm with input-output function f; it provides a result based solely on the input value, independent of the way in which the value is computed. This may be contrasted with the behavior of the algorithm $\mathsf{minalg}\, f$, defined by

$$\mathsf{minalg} = \lambda f : M \multimap M' \;.\; \lambda t \in \mathcal{D}(PM) \;.\; f(t0).$$

It is easy to see that $\mathsf{fun} \circ \mathsf{minalg} = \mathsf{id}_{M \multimap M'}$, but $\mathsf{minalg} \circ \mathsf{fun} \leq \mathsf{id}_{M \Rightarrow M'}$, and therefore $\mathsf{minalg}\, f$ is the *least* algorithm with input-output function f. Intuitively, this is the "most eager" algorithm with input-output function f, since it specifies that the computation of the input value must be completed in one time step. (Note, however, that minalg does not define a functor.)

Proposition 3.18 *For every continuous function $f : M \multimap M'$, the set of algorithms in $M \Rightarrow M'$ with input-output function f, ordered pointwise, forms a complete lattice.*

Proof: The domain $\mathcal{D}(M \Rightarrow M')$ is consistently complete, and a has input-output function f iff $\mathsf{minalg}(f) \leq a \leq \mathsf{alg}(f)$. ∎

3.4 Remarks on Canonicity

Note that the identity algorithm val_M is canonical: $\mathsf{val}_M = \mathsf{alg}\, \mathsf{id}_M$. The projection algorithms are also canonical, $\hat{\pi}_i = \mathsf{alg}\, \pi_i$. The application algorithm $\widehat{\mathsf{app}}$, however, is not canonical. Let $\overline{\mathsf{app}}$ be the input-output function for $\widehat{\mathsf{app}}$. We have:

$$\begin{aligned}
\widehat{\mathsf{app}}_{M,M'} &= \mathsf{app}_{PM,M'} \circ (\mathsf{val}_{M \Rightarrow M'} \times \mathsf{id}_M) \circ \mathsf{split} \\
&= (\lambda(s,t) \;.\; (\mathsf{val}_{M \Rightarrow M'}\, s)t) \circ \mathsf{split} \\
&= (\mathsf{uncurry}\, \mathsf{val}_{M \Rightarrow M'}) \circ \mathsf{split} \\
&= \overline{\mathsf{uncurry}}\, \mathsf{val}_{M \Rightarrow M'} \\
\overline{\mathsf{app}}_{M,M'} &= \mathsf{fun}\, \widehat{\mathsf{app}}_{M,M'} \\
&= \widehat{\mathsf{app}}_{M,M'} \circ \mathsf{path}_{M \Rightarrow M' \times M} \\
&= \mathsf{app}_{PM,M'} \circ (\mathsf{val}_{M \Rightarrow M'} \times \mathsf{id}_M) \circ \mathsf{split}_{M \Rightarrow M', M} \circ \mathsf{path}_{M \Rightarrow M' \times M} \\
&= \mathsf{app}_{PM,M'} \circ (\mathsf{val}_{M \Rightarrow M'} \times \mathsf{id}_M) \circ (\mathsf{path}_{M \Rightarrow M'} \times \mathsf{path}_M) \\
&= \mathsf{app}_{PM,M'} \circ (\mathsf{id}_{M \Rightarrow M'} \times \mathsf{path}_M) \\
&= \lambda(a,x) \in \mathcal{D}(M \Rightarrow M' \times M) \;.\; a(\mathsf{path}_M\, x) \\
&= \mathsf{uncurry}(\mathsf{fun}) \\
\mathsf{alg}\, \overline{\mathsf{app}}_{M,M'} &= \overline{\mathsf{app}}_{M,M'} \circ \mathsf{val}_{M \Rightarrow M' \times M} \\
&= (\lambda(s,t) \;.\; (\mathsf{val}_{M \Rightarrow M'}\, s)(\mathsf{path}_M(\mathsf{val}_M\, t))) \circ \mathsf{split}\,.
\end{aligned}$$

That is, the application algorithm is uniquely determined as the uncurrying of the identity algorithm, and in general it is not maximal in its input-output equivalence class (nor is it minimal). This reflects the fact that $\widehat{\mathsf{app}}$ ignores the computation of the algorithm to be applied, but does pay attention to the computation of the argument of the application, while the algorithm application function ignores the computation of both the applied algorithm and its argument.

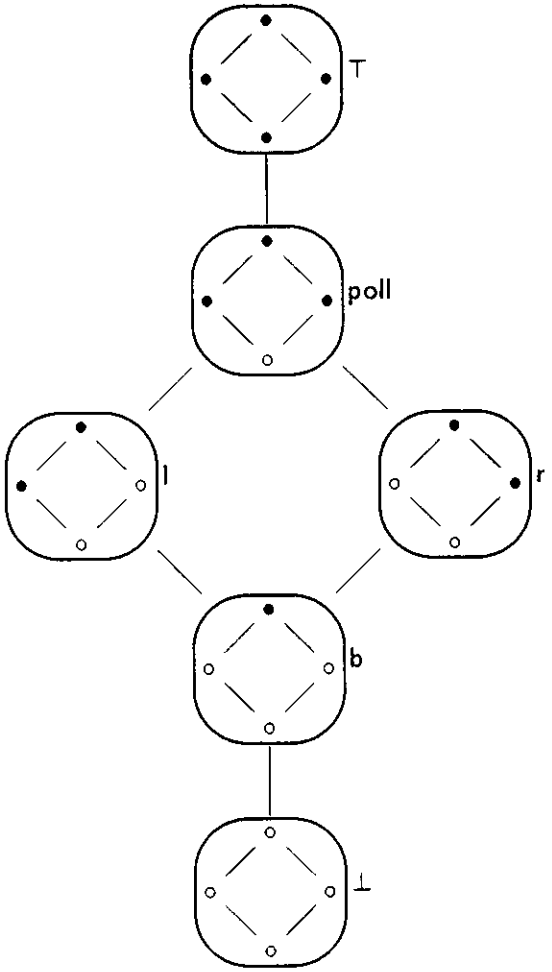

Figure 5: $\mathcal{D}(\mathtt{Two} \times \mathtt{Two} \to \mathtt{Two})$

Example 3.19 In table 1 we list input-output functions for some of algorithms of $\mathtt{Two} \times \mathtt{Two} \Rightarrow \mathtt{Two}$, shown in figure 4. The corresponding function space, $\mathtt{Two} \times \mathtt{Two} \to \mathtt{Two}$, is shown in figure 5. We use identical names for some of the algorithms and functions, such as poll and $\top$, but it should be clear from the context whether we refer to the algorithm or to the function.

We take this opportunity to give an operational intuition, *in lieu* of a formalization of what constitutes a computation strategy, or a detailed discussion of an operational semantics for algorithms.

We take a coroutine-like view of the computation, much as in Berry and Curien's operational semantics for sequential algorithms [Cur86]. Computation is demand driven: a request for the value of a cell in the result may lead the algorithm to issue sub-computations until enough information

a	fun(a)	Termination	Step 1	Step 2
pb	b	$\langle \top, \top \rangle$	l, r	
lb	b	$\langle \top, \top \rangle$	l	r
rb	b	$\langle \top, \top \rangle$	r	l
db	b	$\langle \top, \top \rangle$	l or r	l, r
pl	l	$\langle \top, \emptyset \rangle$	l or r	l
sl	l	$\langle \top, \emptyset \rangle$	l	
pr	r	$\langle \emptyset, \top \rangle$	l or r	r
sr	r	$\langle \emptyset, \top \rangle$	r	
poll	poll	$\langle \top, \emptyset \rangle$ or $\langle \emptyset, \top \rangle$	l or r	
$\top$	$\top$	$\langle \emptyset, \emptyset \rangle$		
$\emptyset$	$\bot$			

Table 1: Input-output Functions and Operational Intuition for Algorithms in $\mathcal{D}(\mathtt{Two} \times \mathtt{Two} \Rightarrow \mathtt{Two})$

has been gathered about the input value for an output value to be determined. In the sequential case, only one sub-computation may be active at any point in time, and hence the coroutine flavor. In the parallel case, sub-computations may be issued in parallel, and several sub-computations (at differing levels) may be active simultaneously. Note that we assume some discrete global clock, with respect to which all computations are synchronized.

In the above example, one may ask for the value filling the cell $*$ in the application of one of these algorithms to an argument. A sub-computation of the left argument to an algorithm corresponds to a computation of the cell $*.1$ of the argument. Returning to the table, under the *Termination* heading we give the least value on which the algorithm will produce a result $\top$, *i.e.*, fill the cell $*$; this is of course determined by the input-output function. Under *Step 1* and *Step 2* we list the sub-computations that must be performed by the (end of the) first or second step of the computation, respectively, if the algorithm is to fill $*$. In this specific case, only termination of the sub-computations matters, since there is only one way to fill any cell; in general, the value with which a computation terminates will also be important.

For instance, the algorithms that compute the function b (standing for *both*, the doubly strict function) can be characterized as follows:

- The algorithm pb specifies that both the left component and the right component of the argument must be computed by the first time point. This is the most eager algorithm for b.
- The algorithm lb specifies that the left component must be computed by the first time point, and that the right component must be computed by the second time point.
- The algorithm db specifies that either the left or the right component must be computed by the first time point, and both must be computed by the second time point.

•

3.5 Intensional and Extensional Aspects of Algorithms

Properties of functions, such as stability and sequentiality, apply to algorithms in several ways. We say that properties of an algorithm's input-output function are *(input-output) extensional* properties

of the algorithm, and properties of the algorithm itself, regarded as a function on paths, are *intensional* properties of the algorithm.

As an example, of the algorithms in figure 4, **poll** is neither extensionally nor intensionally stable and neither extensionally nor intensionally sequential; **db**, **pl** and **pr** are extensionally stable (and sequential) and are not intensionally stable (or sequential); and all other algorithms are stable (and sequential), both intensionally and extensionally. It is not by chance that some possible combinations of properties are not represented:

Proposition 3.20 *The input-output function of a stable (respectively, sequential) algorithm is stable (respectively, sequential).*

A function is stable (respectively, sequential) iff it the input-output function of some stable (respectively, sequential) algorithm.

Proof: If a is stable then $\mathsf{fun}(a)$ is stable, because any counter-example to the stability of $\mathsf{fun}(a)$ generates a counter-example to the stability of a. Similarly if f is stable then $\mathsf{minalg}(f)$ is stable. The proofs for sequentiality are analogous. ∎

Therefore there can be no algorithm that is extensionally, but not intensionally stable (or sequential). The function **poll**, for instance, has no sequential or stable algorithm.

4 Relationship to Berry and Curien's Sequential Algorithms

We discuss now the strong connections between the work presented here and Berry and Curien's sequential algorithms [Cur86]. Although we will not discuss them in detail, similar relationships can be established between our earlier attempts to define parallel algorithms [BG90] and the current work. We believe that these connections show how our view of intensionality as a computational comonad is a natural outcome of the earlier lines of research.

4.1 Berry-Curien Sequential Algorithms on CDSs

We first present some relevant definitions concerning Berry-Curien sequential algorithms. The reader is referred to [Cur86] for a complete exposition.

Definition 4.1 Given (discrete generalized) deterministic CDSs M and M', the Berry-Curien sequential exponentiation, or sequential algorithm space, $M \Rightarrow^{\dagger} M'$, is defined (as in [Cur86]) by:

- $\mathrm{C}_{M\Rightarrow^{\dagger}M'} = \mathcal{D}_{\mathrm{fin}}(M)\times\mathrm{C}_{M'}$ where $\mathcal{D}_{\mathrm{fin}}(M)$ consists of the finite elements of $\mathcal{D}(M)$ – which, for a discrete M, are just states that are finite as sets of events. The cells are ordered discretely.
- $\mathrm{V}_{M\Rightarrow^{\dagger}M'} = \{\mathtt{output}\ v' \mid v' \in \mathrm{V}_{M'}\} \cup \{\mathtt{valof}\ c \mid c \in \mathrm{C}_M\}$.
- $$\mathrm{E}_{M\Rightarrow^{\dagger}M'} = \begin{array}{ll} & \{(xc', \mathtt{output}\ v') \in \mathrm{C}_{M\Rightarrow^{\dagger}M'} \times \mathrm{V}_{M\Rightarrow^{\dagger}M'} \mid (c', v') \in \mathrm{E}_{M'}\} \\ \cup & \{(xc', \mathtt{valof}\ c) \in \mathrm{C}_{M\Rightarrow^{\dagger}M'} \times \mathrm{V}_{M\Rightarrow^{\dagger}M'} \mid c \in \mathrm{A}(x)\}\,. \end{array}$$
- $$\{(x_j c'_j, \mathtt{output}\ v'_j) \mid 1 \le j \le l\} \vdash_{M\Rightarrow^{\dagger}M'} xc' \quad \begin{array}{l}\text{iff } \{(c'_j, v'_j) \mid 1 \le j \le l\} \vdash_{M'} c' \\ \text{and } x = \bigcup\{x_j \mid 1 \le j \le l\}.\end{array}$$

$\{(x_1 c', \mathtt{valof}\ c)\} \vdash_{M\Rightarrow^{\dagger}M'} xc'$ iff there exists an event (c, v) of M such that $x = x_1 \cup \{(c, v)\}$.

The states of $M \Rightarrow^{\dagger} M'$ are the sequential algorithms from M to M'. •

The following results are shown in [Cur86]: the category of (discrete generalized) deterministic CDSs and sequential algorithms is a ccc; the product is the product of gCDSs given above and `Null` is a terminal object; the exponentiation object of M and M' is $M \Rightarrow^{\dagger} M'$.

Definition 4.2 The input-output function $\mathsf{fun}^{\dagger}\, b$ of an algorithm b is given (as in [Cur86]) by:

$$\mathsf{fun}^{\dagger} = \lambda b \in \mathcal{D}(M \Rightarrow^{\dagger} M') \,.\, \lambda x \in \mathcal{D}(M) \,.\, \{(c', v') \mid \exists y \subseteq x \,.\, (yc', \mathtt{output}\ v') \in b\}\,.$$

•

There is no analogue in the Berry-Curien model to our definition of the **alg** map on parallel algorithms, since there is no uniform way to pick a canonical sequential algorithm for a sequential function with more than one sequentiality index (such as the doubly-strict-or function and the function **b**). Nevertheless, Berry and Curien have shown that each element of the sequential function space is the input-output function of some sequential algorithm. Moreover, the quotient of $\mathcal{D}(M \Rightarrow^{\dagger} M')$ by input-output equivalence (having the same input-output function) is isomorphic to the sequential function space from $\mathcal{D}(M)$ to $\mathcal{D}(M')$, ordered by the stable ordering.

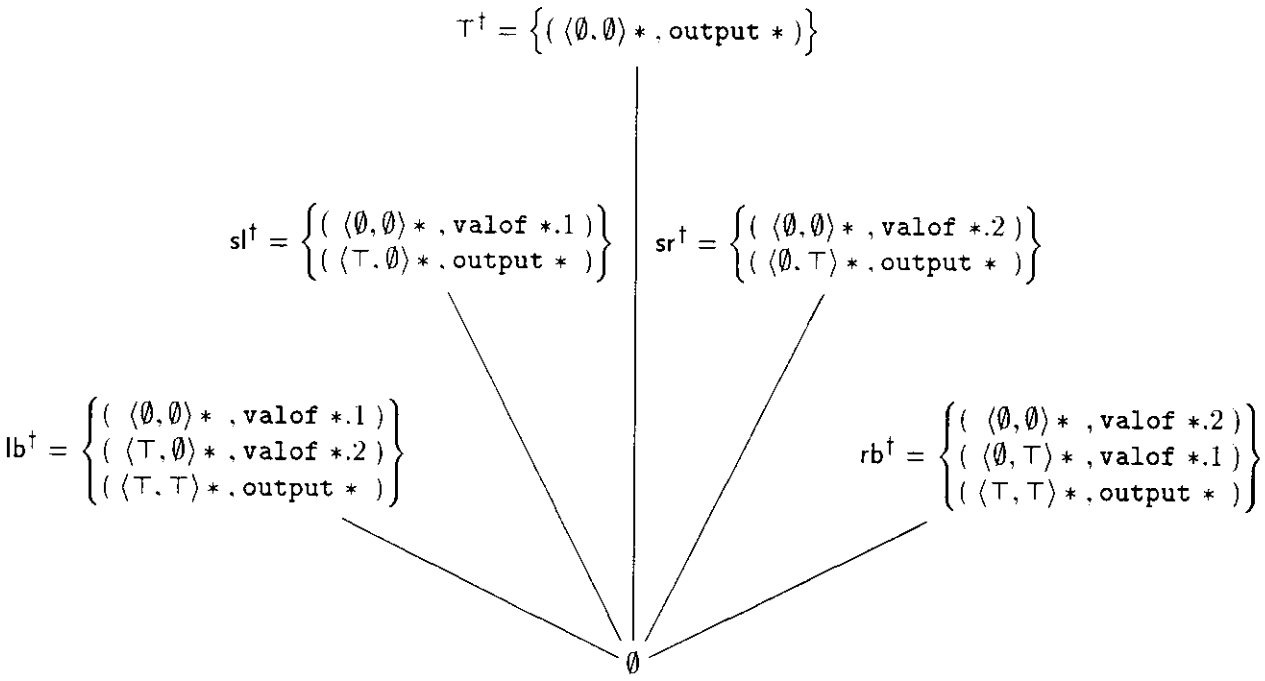

Figure 6: The Berry-Curien sequential algorithm space $\mathcal{D}(\mathtt{Two} \times \mathtt{Two} \Rightarrow^{\dagger} \mathtt{Two})$.

Example 4.3 Figure 6 presents $\mathcal{D}(\mathtt{Two} \times \mathtt{Two} \Rightarrow^{\dagger} \mathtt{Two})$, and table 2 presents the operational intuition for those algorithms. Note that, of the continuous functions in figure 5, only **poll** is not sequential. Also, the sequential functions of this type yield a flat domain when ordered by the stable order.

The operational behavior of sequential algorithms is more straightforward than for the parallel algorithms. Computation is again demand-driven, based on a coroutine-like but sequential operational semantics. For instance, when the algorithm $\mathsf{lb}^{\dagger}$ is applied to some input and a request is

b	$\mathsf{fun}^\dagger(b)$	Termination	Step 1	Step 2	$\mathsf{embed}(b)$
$\mathsf{lb}^\dagger$	b	$\langle\top,\top\rangle$	l	r	lb
$\mathsf{rb}^\dagger$	b	$\langle\top,\top\rangle$	r	l	rb
$\mathsf{sl}^\dagger$	l	$\langle\top,\emptyset\rangle$	l		sl
$\mathsf{sr}^\dagger$	r	$\langle\emptyset,\top\rangle$	r		sr
$\top^\dagger$	$\top$	$\langle\emptyset,\emptyset\rangle$			$\top$
$\emptyset$	$\bot$				$\emptyset$

Table 2: Input-output Functions and Operational Intuition for $\mathcal{D}(\mathtt{Two} \times \mathtt{Two} \Rightarrow^\dagger \mathtt{Two})$

made for the value of the result cell $*$, the algorithm specifies that a sub-computation of the left input argument (the cell $*.1$) must be issued first; if this cell is filled with the value $*$, the right argument is similarly computed; if this too is filled, the algorithm specifies that result cell $*$ can then be filled, with the value $*$. •

Our example is limited in that only one value may fill each of the cells. In the presence of several values, branching may take place, based on the value, and the enabling structure of the algorithm takes on a tree shape (a linear list in this example). A computation simply determines a path in this tree – a sequence of (strictly increasing) states, serving as (finite) approximations to the input. This is the basic intuition which may be carried over to our formulation of parallel algorithms.

4.2 Embedding Berry-Curien Algorithms into Parallel Algorithms

We sketch how the Berry-Curien sequential algorithms space $\mathcal{D}(M \Rightarrow^\dagger M')$ may be embedded into the parallel algorithm space $\mathcal{D}(M \Rightarrow M')$ in a way that preserves the input-output function and the computation strategy. The embedding also respects the ordering (set inclusion) of the Berry-Curien model.

Definition 4.4 For $b \in \mathcal{D}(M \Rightarrow^\dagger M')$ and $xc' \in \mathrm{F}(b)$, define (by induction on the enabling of xc'):

$$\mathsf{hist}(b, xc') = \begin{cases} \mathsf{hist}(b, x_0 c')x & \text{if } \{(x_0 c', \mathtt{valof}\ c)\} \vdash_b xc' \\ \bigvee \{\mathsf{hist}(b, x_j c'_j) \mid 1 \le j \le l\} & \text{if } \{(x_j c'_j, \mathtt{output}\ v_j) \mid 1 \le j \le l\} \vdash_b xc'. \end{cases}$$

•

Intuitively, $\mathsf{hist}(b, xc')$ is a finite sequence of states of M, that may be seen as a computation undertaken when b is applied to an argument approximated by x, trying to fill the cell c' in the result. We use juxtaposition for concatenation of finite sequences. The lub $\bigvee S$ of a set S of finite sequences is obtained by a componentwise lub (*i.e.*, componentwise union) of the sequences, with last components of repeated as necessary to obtain sequences as long as the longest one[1]. The lub of the empty set of sequences is the empty sequence, and the empty sequence is extended by repeating $\emptyset$ as often as necessary.

[1]It would also make sense to work with a pre-order on events, rather than a linear order implied by the paths comonad, with $\bigvee S$ given by the pre-order union of elements of S. We return to this alternative in the conclusion, when considering alternate notions of comonads.

Definition 4.5 The embedding function $\mathsf{embed} : \mathcal{D}(M \Rightarrow^{\dagger} M') \to \mathcal{D}(M \Rightarrow M')$ is given by:

$$\mathsf{embed} = \lambda b \in \mathcal{D}(M \Rightarrow^{\dagger} M') \;.\; \mathsf{up}(\{(\mathsf{hist}(b, xc')c', v') \mid (xc', \mathtt{output}\ v') \in b\}).$$

Here we implicitly extend the finite sequences $\mathsf{hist}(b, xc')$ to infinity by repeating their last component, with the convention that the empty sequence represents $\emptyset^{\omega}$. •

Embedding preserves the input-output function:

Proposition 4.6 *For any* $b \in \mathcal{D}(M \Rightarrow^{\dagger} M')$, $\mathsf{fun}(\mathsf{embed}\, b) = \mathsf{fun}^{\dagger} b$.

Example 4.7 The embedding of figure 6 into figure 4 is straightforward, shown in the $\mathsf{embed}(b)$ column in table 2. It is easy to see that the (informally given) computation strategy is preserved. The image of $\mathcal{D}(\mathtt{Two} \times \mathtt{Two} \Rightarrow^{\dagger} \mathtt{Two})$ under embedding is a proper subset of the intensionally sequential algorithms in $\mathcal{D}(\mathtt{Two} \times \mathtt{Two} \to \mathtt{Two})$. While pb is intensionally sequential, it does not impose a linear order of evaluation on the two sequentiality indices and therefore does not correspond to a Berry-Curien algorithm. •

Although we cannot give a rigorous proof without first formalizing the notion of computation strategy, it should be intuitively clear that the embedding function always preserves the computation strategy of its argument.

It is easy to show that embedding preserves order, in the following sense:

Proposition 4.8 *For all (discrete generalized) deterministic CDSs* M *and* M', *and all* $b_1, b_2 \in \mathcal{D}(M \Rightarrow^{\dagger} M')$, *if* $b_1 \subseteq b_2$ *then* $\mathsf{embed}(b_1)$ *approximates* $\mathsf{embed}(b_2)$ *in the stable ordering.*

5 Directions for Further Research

In this paper we have introduced a generalization of concrete data structures and concrete domains, and parallel algorithms between generalized concrete data structures. We would like to demonstrate the utility of our new structures in supporting the development of a theory of sequentiality and parallelism and in the development of a framework for intensional semantics. The results of this paper constitute a foundation on which to build, but there are many topics for further investigation and several directions for us to follow.

We have presented a cartesian closed category of gCDSs and continuous functions, and we discussed briefly a ccc of distributive gCDSs and stable functions. These two categories employ a common underlying concrete representation – the gCDS – but use different notions of states to obtain different notions of generated domains. We would like to give a domain-theoretic characterization to both families of domains. We have made a start in this direction, with the partial characterization of the domains generated by the continuous notion of state.

It seems likely that some other natural classes of functions may yield to an analogous development. In particular, we would like to try to use the same approach to define a category of gCDSs and sequential functions, centered on a suitable notion of a sequential state. This task is harder, since we cannot rely on the desired category being a full sub-category of some already known ccc, such as Scott domains and continuous functions or dI-domains and stable functions. Nevertheless, our initial investigations in this direction are encouraging. We have pointed out that a definition of sequentiality may be formulated in the gCDS setting. We intend to study the implications of such a definition, and whether it proves useful in obtaining sequential semantic models. We have already made some remarks concerning these issues in section 2.6.

We have presented an intensional semantic model – the category of gCDSs and parallel algorithms, obtained by the co-Kleisli category construction from the ccc of gCDSs and continuous functions and the comonad of paths. Since the paths comonad preserves product the obtained intensional category is a ccc, so that it may be used in a standard way to provide an (intensional) model for the simply-typed lambda calculus. This choice of comonad is not the only reasonable one. Indeed, for some purposes the paths comonad may be regarded as too detailed. For instance, if one is not interested in the number of steps between successive increments in a computation, but only in the relative order in which the increments occur, it would seem appropriate to use the comonad of *strictly increasing* (rather than non-decreasing) paths[2]. Another possible choice of comonad might be obtained by regarding events not as linearly ordered but merely as partially ordered or even pre-ordered, so that we may dispense with any assumption of a global clock.

However, comonads based on strictly increasing paths or on pre- or partial orders on events do not preserve finite products. This means that the intensional category for these notions of intensional behavior will not normally be a ccc, even if the underlying extensional category is cartesian closed. Nevertheless, we believe that algorithm categories built with such comonads may still provide sensible intensional models for the λ-calculus, and we will report on this separately.

As an aside, the reasoning here helps to explain the shortcomings of our earlier "query model" [BG90]; in our current terminology, we were attempting there to obtain a ccc (with currying built-in), using (the analogue of) the non-product preserving comonad of strictly increasing paths. We now realize that this combination does not yield a ccc.

We have exhibited a generalization of Berry and Curien's sequential algorithms into parallel algorithms, together with an embedding of the former into the latter. Essentially the same embedding should also work when we consider comonads based on strictly increasing paths, or on partial orders, or on pre-orders over events. A "tighter" embedding could be obtained into the *stable algorithms*, which we may construct as the co-Kleisli category of **gCDSstab**, using any of the above variations on the paths comonad. We would now like to understand better the Berry-Curien construction itself. On the face of it, the Berry-Curien category of deterministic concrete data structures and sequential algorithms is not attainable as an application of the co-Kleisli construction, since there is no underlying extensional category of concrete data structures. We conjecture, however, that there are strong connections between the Berry-Curien category and an intensional category of gCDSs employing a suitable notion of sequential algorithms. In order to establish this conjecture we must first, of course, exhibit an appropriate extensional category of gCDSs and sequential functions, one of the goals listed above.

Once we have a sufficiently established theory of intensional semantics, we would like to turn to its application to reasoning about intensional properties of programs, such as efficiency.

References

[BC82] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20:265–321, 1982.

[BCL85] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 3, pages 89–132. Cambridge University Press, 1985.

[2] In diagrams 3 and 4 we only listed explicitly the strictly increasing paths, and algorithms that are defined by their actions on strictly increasing paths. There are many paths containing repeated "idle steps" between successive increments, and hence many more algorithms than those listed.

[Ber78] G. Berry. Stable models of typed λ-calculi. In *Proc. 5^{th} Coll. on Automata, Languages and Programming*, number 62 in Lecture Notes in Computer Science, pages 72–89. Springer-Verlag, July 1978.

[BG90] S. Brookes and S. Geva. Towards a theory of parallel algorithms on concrete data structures. In *Semantics for Concurrency, Leicester 1990*, pages 116–136. Springer-Verlag, 1990. Expanded and improved version to appear in Theoretical Computer Science.

[BG91] S. Brookes and S. Geva. A cartesian closed category of parallel algorithms between Scott domains. Submitted for publication in *Semantics of Programming Languages and Model Theory, Dagstuhl Castle, June 1991*, Algebra, Logic and Applications, London, 1991. Gordon and Breach Science Publishers.

[Col89] L. Colson. About primitive recursive algorithms. In *Proceedings of ICALP89*, volume 372 of *Lecture Notes in Computer Science*, pages 194–206. Springer-Verlag, 1989.

[Cur86] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science. Pitman, 1986.

[KP78] G. Kahn and G. D. Plotkin. Domaines concrets. Rapport 336, IRIA-LABORIA, 1978.

[Mil77] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[ML71] S. Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[See89] R. A. G Seely. Linear logic, *-autonomous categories and cofree coalgebras. *Contemporary Mathematics*, 92:371–382, 1989.

[Sto88] A. Stoughton. *Fully Abstract Models of Programming Languages.* Research Notes in Theoretical Computer Science. Pitman, 1988.

[Vui73] J. Vuillemin. *Proof techniques for recursive programs.* PhD thesis, Stanford University, 1973.