# Parallel Exponentiation of Concrete Data Structures

Stephen Brookes Shai Geva

December 1989
CMU-CS-89-206

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by the Defense Advanced Research Projects Agency (DOD) and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), United States Air Force, Wright-Patterson AFB, OHIO 45433-6543, under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20.

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 the Defense Advanced Research Projects Agency or the U.S. government.

**Abstract**

Berry and Curien, building on Kahn and Plotkin's theory of Concrete Data Structures and sequential functions, have defined a sequential exponentiation of Concrete Data Structures. Their construction gives rise to an appealing model of sequential algorithms and a functional programming language CDS0, exhibiting some interesting semantic properties, such as: loss of extensionality; a notion of program equivalence sensitive to computation strategy; a lazy, coroutine-like semantics; full abstraction; identification of syntax and semantics.

We propose to develop a similar model of *concurrent computation* by formulating an appropriate notion of *parallel algorithm* and generalizing their construction to a *parallel exponentiation*, while retaining analogous semantic properties. The introduction of parallelism into this setting raises several interesting issues, such as the existence, nature and properties of a satisfactory model of parallel computation employing parallel algorithms; ideally one would like to obtain a cartesian closed category with appropriate notions of parallel application, currying and composition. We hope that our ideas will yield insights into the proper mathematical treatment of parallel programming.

We present here our notion of parallel algorithm and our parallel exponentiation of Concrete Data Structures. We motivate carefully the ideas behind our construction, and we explain how it can be viewed as a natural generalization of the Berry-Curien construction. We introduce application and currying operations suitable for our parallel setting. In order to justify our ideas and to place our work into context, we show some properties of our construction, and give a variety of examples. Finally, we indicate some directions for further research on issues raised by our model.

# 1 Introduction

The search for a satisfactory syntactic and semantic account of sequential computation, in particular the desire to achieve full abstraction, has led to a considerable body of research. In the classic paper [Plo77], Plotkin showed that under its standard interpretation the programming language PCF is inherently sequential, and that the standard continuous function semantic model is not fully abstract because of the fact that the model contains inherently parallel functions (such as parallel-or) that cannot be defined in PCF. The continuous functions model is, however, fully abstract for a parallel version of PCF obtained by including a parallel conditional primitive. A substantial body of work has been directed at obtaining a truly sequential model for the original PCF with a suitably restricted notion of function [BCL85].

Kahn and Plotkin [KP78] have defined *Concrete Data Structures*, or CDSs, together with their order-theoretic counterparts, *Concrete Domains*. They introduced a notion of *sequential function* that leads to a product-closed category. The notion of *stability* was introduced by Berry [Ber78] in an attempt to find a suitably restrictive property of functions intermediate between sequentiality and continuity. Berry introduced the *stable ordering* on stable functions and showed that it possesses some natural and interesting properties. This investigation pointed out that other orderings than the usual extensional ordering on functions might be more suitable in achieving a satisfactory semantic treatment. However, Berry and Curien [BC82,Cur86] showed that the category of concrete domains fails to be cartesian closed when the morphisms in the category are taken to be the continuous functions, or the stable functions, or the sequential functions, between concrete domains; consequently, neither stability nor sequentiality produces a sufficiently restricted notion of function to serve satisfactorily as the basis for a definition of exponentiation for concrete domains.

Berry and Curien were able to define an exponentiation for concrete data structures, by replacing functions by a notion of *sequential algorithms*. For *deterministic* CDSs, or DCDSs, the resulting category of DCDSs and sequential algorithms turns out to be cartesian closed. Furthermore, a notation for elements of DCDSs is a basis for a functional language CDS0, which is given a semantic model with several interesting properties:

- The model is not extensional; a sequential algorithm may be viewed as a sequential input-output function paired with a computation strategy.
- The operational semantics is essentially based on an extension of Kahn-MacQueen's coroutine mechanism [KM77], employing lazy evaluation.
- The semantics is fully abstract, with respect to a suitable notion of observability that is sensitive to computation strategy.
- Observability extends uniformly to higher-order DCDSs.
- DCDS constants are interpreted as themselves, thereby blurring the "syntax vs. semantics" distinction.

The Berry–Curien model of sequential algorithms provides deep insights into the nature of deterministic sequential computation. Although Berry and Curien also discussed briefly an attempt to introduce non-determinism into their model [Cur86, section 2.7], they were unable to obtain a cartesian closed category.

We report here an attempt to generalize Berry and Curien's sequential exponentiation construction so as to incorporate concurrency into the framework, while retaining the appropriate analogues of the above semantic properties. We believe that there are fundamental insights into

the semantic treatment of parallelism to be gained by doing this. We have made the decision to restrict our attention so far to deterministic computation, thereby allowing us to retain an essentially functional semantics. We will, of course, allow non-determinism in the scheduling of computations; indeed, one of our objectives is to improve on the original construction so as to avoid unnecessary scheduling constraints.

We hope ultimately to achieve an appealing model of parallel computation. This is our first exposition of the genesis of this model. We present here a notion of parallel algorithm and a new parallel exponentiation for deterministic concrete data structures, and we formalize what it means to *execute* a parallel algorithm by defining a suitable *application* operation. We introduce currying and uncurrying operations and show that they have the expected properties. We give a variety of examples to illustrate the problems that arise when incorporating parallelism into this setting, and to demonstrate our solutions.

We do not give proofs of our results here for lack of space, and in anticipation of better formulations of the model, as it evolves. We also do not provide yet a formal operational semantics and a generalization of the programming language, although we do take care to supply informal and intuitive motivation and justification for our ideas. Our exposition relies heavily on an informal understanding of our intended operational semantics, and we supply the necessary informal explanations. Future work will provide formal details and present a parallel programming language based on these ideas together with an operational semantics. A major objective of our research is to obtain a cartesian closed category suitable for modelling parallelism.

Readers familiar with Berry and Curien's work may proceed directly to section 2. In the remainder of this section we present briefly CDSs and their sequential exponentiation and product, as well as the definitions of stable and sequential functions. Here we follow the development of [Cur86] closely, and point out any changes we make. We omit parts of the development which may be derived as a specialization of our corresponding definitions and results. The operational semantics is presented only informally.

In section 2 we present our parallel exponentiation of CDSs, beginning with our notion of parallel algorithm. We explain carefully how our construction arises out of an attempt to generalize the Berry-Curien concepts to incorporate an intuitively appealing treatment of parallelism. We also present "trim powerdomains", which arise naturally in the construction, and we define currying of parallel algorithms. In section 3 we define an application operation, and a restricted notion of ground application. For a restricted class of CDSs, the filiform DCDSs, we are able to simplify the definition of application. Finally, we discuss the work remaining to be done, and directions for further investigation.

## 1.1 Concrete Data Structures

**Definition 1.1.1** A *concrete data structure*, or *CDS*, $(\mathrm{C}, \mathrm{V}, \mathrm{E}, \vdash)$ consists of

- A set C of *cells*.
- A set V of *values*.
- A set $\mathrm{E} \subseteq \mathrm{C} \times \mathrm{V}$ of *events*.
- An *enabling* relation $\vdash \subseteq \mathcal{P}_{fin}(\mathrm{E}) \times \mathrm{C}$ between finite sets of events and cells.
  For $y \subseteq_{fin} \mathrm{E}$, we say that $y$ is an *enabling* of $c$, or that $y$ *enables* $c$, iff $y \vdash c$. If $\{e_1, \ldots e_k\} \vdash c$ we may write $e_1, \ldots e_n \vdash c$. If $\emptyset \vdash c$ we may write $\vdash c$, and say that $c$ is *initial*.
  We denote events as pairs $(c, v)$. We also use an alternative notation $c = v$ in examples and discussions, but only the former in formal contexts.

Where necessary, we may subscript the constituents of a CDS, so that for a CDS $M$ we have $M = (\mathrm{C}_M, \mathrm{V}_M, \mathrm{E}_M, \vdash_M)$.

•

**Definition 1.1.2** For a CDS $M$, $y \subseteq \mathrm{E}_M$, and $c \in \mathrm{C}_M$:

- $\mathrm{F}(y) = \{c | (c, v) \in y \text{ for some } v\}$, cells *filled* in $y$.
- $\mathrm{E}(y) = \{c | y' \vdash_M c \text{ for some } y' \subseteq y\}$, cells *enabled* in $y$.
- $\mathrm{A}(y) = \mathrm{E}(y) \setminus \mathrm{F}(y)$, cells *accessible* in $y$. A cell is accessible iff it is enabled but not filled.

•

**Definition 1.1.3** For a CDS $M$ and $c, c' \in \mathrm{C}_M$, we say that $c$ *immediately precedes* $c'$ and $c'$ *immediately follows* $c$, denoted $c \ll_M c'$, iff there exists an enabling $y' \vdash_M c'$ of $M$ such that $c \in \mathrm{F}(y')$. Taking the reflexive and transitive closure of $\ll_M$, we say that $c$ *precedes* $c'$ and $c'$ *follows* $c$ iff $c \ll_M^* c'$.

A CDS $M$ is *well founded* iff $\ll_M$ is well founded. •

**Definition 1.1.4** For $M$ a well founded CDS and $y \subseteq \mathrm{E}_M$:

- $y$ is *functional*[1] iff $(c, v_1), (c, v_2) \in y$ implies that $v_1 = v_2$.
- $y$ is *safe*[2] iff $\mathrm{F}(y) \subseteq \mathrm{E}(y)$.
- $y$ is a *state* of $M$ iff it is functional and safe.

And we define the following collections of sets of events:

- $\mathcal{E}(M) = \mathcal{P}(\mathrm{E}_M)$, the collection of sets of events of $M$.
- $\mathcal{F}(M)$, the collection of functional sets of events of $M$.
- $\mathcal{S}(M)$, the collection of safe sets of events of $M$.
- $\mathcal{D}(M) = \mathcal{F}(M) \cap \mathcal{S}(M)$, the collection of states of $M$.

We add a subscript to indicate finiteness of the sets of events involved, e.g, $\mathcal{F}_{fin}(M)$ for the collection of finite functional sets of events, and $\mathcal{D}_{fin}(M)$ for the collection of finite states of $M$. •

We order $\mathcal{E}(M)$ by set inclusion, obtaining a poset $(\mathcal{E}(M), \subseteq)$. We also order functional sets of events and states by set inclusion, but denote the orders $\subseteq_{\mathcal{F}}$ and $\subseteq_{\mathcal{D}}$, respectively, to emphasize the intended framework (and in anticipation of the possible need to generalize the order later). We denote the lub in these two posets as $\cup_{\mathcal{F}}$ and $\cup_{\mathcal{D}}$, respectively. We say that a poset is consistently complete iff every consistent (i.e- upper-bounded) subset has a least upper bound. We then obtain the following result for the posets $(\mathcal{F}(M), \subseteq_{\mathcal{F}})$, $(\mathcal{D}(M), \subseteq_{\mathcal{D}})$, and their finite versions.

---

[1] Berry and Curien use the term *consistent* instead of functional. We choose to use the latter term so as to avoid confusion with consistency in a poset.

[2] We simplify the definition of safety by assuming a well founded CDS. The definition in [Cur86] requires that each cell filled in $y$ have a linear deduction in $y$ (w.r.t the enabling relation). This is then shown equivalent to having a tree-like proof in $y$, and, for a well founded CDS, to having an enabling in $y$.

**Proposition 1.1.5** *Let $M$ be a well founded CDS. Then the posets $(\mathcal{F}(M), \subseteq_{\mathcal{F}})$, $(\mathcal{F}_{fin}(M), \subseteq_{\mathcal{F}})$, $(\mathcal{D}(M), \subseteq_{\mathcal{D}})$ and $(\mathcal{D}_{fin}(M), \subseteq_{\mathcal{D}})$ are all consistently complete, with the empty set as minimal element, and set unions as least upper bounds. Lubs of a set of elements contained in any two of the posets coincide, provided they exist.*

$(\mathcal{D}(M), \subseteq_{\mathcal{D}})$ is in fact a concrete domain[3]. See [KP78] and [Cur86, section 2.2] for the representation theorem relating concrete domains and states of CDSs ordered by inclusion.

**Definition 1.1.6** A well founded CDS $M$ is *stable* iff for any state $x$ and cell $c$ enabled in $x$, $c$ has a unique enabling in $x$, *i.e.* if $y \vdash_M c$, $y' \vdash_M c$, and $y, y' \subseteq x$ then $y = y'$.

A CDS $M$ is a *deterministic* CDS, or DCDS for short, iff it is well founded and stable. •

For simplicity, we will work from now on exclusively with DCDSs, although some of the development could be carried out in somewhat more general terms.

The glbs of consistent sets of states of a DCDS are just set intersections.

**Proposition 1.1.7** *For a DCDS $M$, if $X$ is a consistent, non-empty, set of states of $M$, then $\cap X$ is a state of $M$, and it is the glb of $X$ in $(\mathcal{D}(M), \subseteq_{\mathcal{D}})$.*

**Example 1.1.8** The empty DCDS **Null** has no cells, no values, no events, and an empty enabling relation:

$$\mathrm{C}_{\mathbf{Null}} = \mathrm{V}_{\mathbf{Null}} = \mathrm{E}_{\mathbf{Null}} = \vdash_{\mathbf{Null}} = \emptyset$$

Its only state is the empty state $\emptyset$. •

**Example 1.1.9** The DCDS **Bool** has a single, initial, cell, which may be filled with values representing the boolean truth values:

- $\mathrm{C}_{\mathbf{Bool}} = \{\mathtt{b}\}$
- $\mathrm{V}_{\mathbf{Bool}} = \{\mathtt{tt}, \mathtt{ff}\}$
- $\mathrm{E}_{\mathbf{Bool}} = \{\mathtt{b} = \mathtt{tt}, \mathtt{b} = \mathtt{ff}\}$
- $\vdash_{\mathbf{Bool}} \mathtt{b}$

Its states are $\emptyset$, $\{\mathtt{b} = \mathtt{tt}\}$ and $\{\mathtt{b} = \mathtt{ff}\}$, and thus $(\mathcal{D}(\mathbf{Bool}), \subseteq_{\mathcal{D}})$ is isomorphic to the conventional flat boolean cpo. •

**Example 1.1.10** The DCDS **Nat** has a single, initial, cell, which may be filled with a natural number. We use $\mathbb{N}$ for the set of natural numbers. Thus,

- $\mathrm{C}_{\mathbf{Nat}} = \{\mathtt{n}\}$
- $\mathrm{V}_{\mathbf{Nat}} = \mathbb{N}$
- $\mathrm{E}_{\mathbf{Nat}} = \{\mathtt{n} = k \mid k \in \mathbb{N}\}$
- $\vdash_{\mathbf{Nat}} \mathtt{n}$

Its states are $\emptyset$ and $\{\mathtt{n} = k\}$ for $k \in \mathbb{N}$, so that $(\mathcal{D}(\mathbf{Nat}), \subseteq_{\mathcal{D}})$ is isomorphic to the conventional flat natural numbers cpo. •

[3] When suitable countability requirements are imposed.

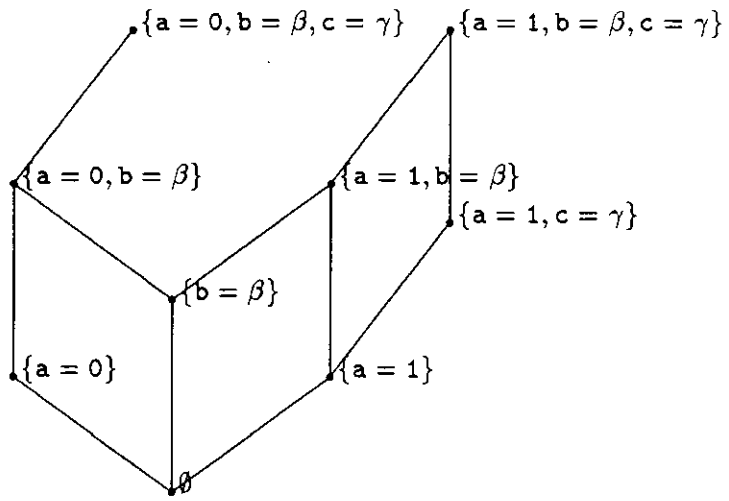



Figure 1: The poset $(\mathcal{D}(\mathbf{Foo}), \subseteq_{\mathcal{D}})$

**Example 1.1.11** The DCDS **Foo** is given by:

- $\mathrm{C}_{\mathbf{Foo}} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
- $\mathrm{V}_{\mathbf{Foo}} = \{0, 1, \beta, \gamma\}$
- $\mathrm{E}_{\mathbf{Foo}} = \{\mathtt{a} = 0, \mathtt{a} = 1, \mathtt{b} = \beta, \mathtt{c} = \gamma\}$
- $\vdash_{\mathbf{Foo}} \mathtt{a}$,
  $\vdash_{\mathbf{Foo}} \mathtt{b}$,
  $\mathtt{a} = 0, \mathtt{b} = \beta \vdash_{\mathbf{Foo}} \mathtt{c}$,
  $\mathtt{a} = 1 \vdash_{\mathbf{Foo}} \mathtt{c}$

Figure 1 presents the Hasse diagram for the poset of its states, $\mathcal{D}(\mathbf{Foo})$. Note that **Foo** is stable, since the two distinct enablings of c are inconsistent and cannot co-exist in a state. •

## 1.2 Product of DCDSs

We define a binary product of DCDSs and a convenient representation for compound sets of events.

First, we need a notation for tagging cells. If $c$ is a cell of a DCDS $M$ and $i$ is a tag or label, we write $c.i$ to indicate the labelled version of cell $c$. Formally, $c.i$ will be the pair $(c, i)$, and in defining products we will use integers as labels. We extend the notation to sets of cells and of events, so that, for $C \subseteq \mathrm{C}_M$ and $y \in \mathcal{E}(M)$, $C.i = \{c.i | c \in C\}$ and $y.i = \{(c.i, v) | (c, v) \in y\}$.

**Definition 1.2.1** For $M_1, M_2$ DCDSs, define the *product* of $M_1$ and $M_2$, denoted by $M_1 \times M_2$, as follows:

| | | | |
|---|---|---|---|
| **Cells** | $\mathrm{C}_{M_1 \times M_2}$ | = | $\mathrm{C}_{M_1}.1 \cup \mathrm{C}_{M_2}.2$ |
| **Values** | $\mathrm{V}_{M_1 \times M_2}$ | = | $\mathrm{V}_{M_1} \cup \mathrm{V}_{M_2}$ |
| **Events** | $\mathrm{E}_{M_1 \times M_2}$ | = | $\mathrm{E}_{M_1}.1 \cup \mathrm{E}_{M_2}.2$ |
| **Enablings** | For $i = 1, 2$ . $y.i \vdash_{M_1 \times M_2} c.i$ iff $y \vdash_{M_i} c$ | | |

Define the *pair* of $z_1 \in \mathcal{E}(M_1)$ and $z_2 \in \mathcal{E}(M_2)$ to be

$$\langle z_1, z_2 \rangle = z_1.1 \cup z_2.2 \in \mathcal{E}(M_1 \times M_2)$$

As a notational convention, we will use $\bar{x}$, $\bar{y}$, etc. to denote such compound sets of events.
Define the projections fst and snd

$$\begin{array}{rcl} \text{fst} & : & \mathcal{E}(M_1 \times M_2) \to \mathcal{E}(M_1) \\ \text{snd} & : & \mathcal{E}(M_1 \times M_2) \to \mathcal{E}(M_2) \end{array}$$

by

$$\begin{array}{rcl} \text{fst}(\langle z_1, z_2 \rangle) & = & z_1 \\ \text{snd}(\langle z_1, z_2 \rangle) & = & z_2 \end{array}$$

•

We let $\times$ associate to the right, so that

$$M_1 \times M_2 \times M_3 = M_1 \times (M_2 \times M_3).$$

The product trivially preserves well foundedness and stability.
Pairing preserves functionality, safety and finiteness:

**Proposition 1.2.2** *$\bar{z} \in \mathcal{E}(M_1 \times M_2)$ iff $\exists z_1 \in \mathcal{E}(M_1), z_2 \in \mathcal{E}(M_2)$ . $\bar{z} = \langle z_1, z_2 \rangle$.*
*If $\bar{z} = \langle z_1, z_2 \rangle$, then*

- *$\bar{z}$ is functional iff for $i = 1, 2$, $z_i$ is functional.*
- *$\bar{z}$ is safe iff for $i = 1, 2$, $z_i$ is safe.*
- *$\bar{z}$ is finite iff for $i = 1, 2$, $z_i$ is finite.*

In particular, a pair of (finite) states is a (finite) state of the product, and a pair of (finite) functional sets of events is a (finite) functional set of events of the product. As a corollary, the projections preserve functionality, safety and finiteness (and thus stateness and finite stateness) of their arguments.

**Example 1.2.3** The DCDS **Bool** $\times$ **Bool** is given by:

- $C_{\mathbf{Bool \times Bool}} = \{\mathtt{b.1}, \mathtt{b.2}\}$
- $V_{\mathbf{Bool \times Bool}} = \{\mathtt{tt}, \mathtt{ff}\}$
- $E_{\mathbf{Bool \times Bool}} = \{\mathtt{b.1} = \mathtt{tt}, \mathtt{b.1} = \mathtt{ff}, \mathtt{b.2} = \mathtt{tt}, \mathtt{b.2} = \mathtt{ff}\}$
- $\vdash_{\mathbf{Bool \times Bool}} \mathtt{b.1}$,
  $\vdash_{\mathbf{Bool \times Bool}} \mathtt{b.2}$

It has 9 states, one of which is $\langle \{\mathtt{b} = \mathtt{tt}\}, \{\mathtt{b} = \mathtt{ff}\} \rangle$, alternatively denoted by $\{\mathtt{b.1} = \mathtt{tt}, \mathtt{b.2} = \mathtt{ff}\}$.
•

## 1.3 Stable and Sequential Functions

We may now define stability and sequentiality of functions between (the collections of states of) DCDSs. The concreteness of DCDSs allows a definition of sequentiality that uses the cells of a concrete data structure in a manner similar to the use of *occurrences* of a syntactic term in a "syntactic" definition of sequentiality [Plo77].

**Definition 1.3.1** Let $M$ and $M'$ be two DCDSs. A continuous function $f$ from $\mathcal{D}(M)$ to $\mathcal{D}(M')$ is *stable* if for any $x \in \mathcal{D}(M)$ and $x' \subseteq_{\mathcal{D}(M')} f(x)$ there exists a state $M(f,x,x') \in \mathcal{D}(M)$, such that, for any $z \subseteq_{\mathcal{D}(M)} x$, $x' \subseteq_{\mathcal{D}(M')} f(z)$ iff $M(f,x,x') \subseteq_{\mathcal{D}(M)} z$.

That is, there must exist a minimal state $M(f,x,x')$ below $x$ on which $f$ attains or surpasses $x'$. •

**Definition 1.3.2** Let $M$ and $M'$ be two DCDSs, and $x \in \mathcal{D}(M)$. A continuous function $f$ from $\mathcal{D}(M)$ to $\mathcal{D}(M')$ is *sequential* at $x$ if, for any $c' \in \mathrm{A}(f(x))$, one of the following holds:

(1) Either $\mathrm{A}(x) = \emptyset$, and thus $x$ has no super-state;[4]

(2) Or there exists some $c \in \mathrm{A}(x)$ that must be filled in any $y$ that increases $x$ such that $c'$ is filled in $f(y)$, that is–

$$\exists c \in \mathrm{A}(x) \;.\; \forall y \in \mathcal{D}(M) \;.\; x \subseteq_{\mathcal{D}} y \;\&\; c' \in \mathrm{F}(f(y)) \Rightarrow c \in \mathrm{F}(y)$$

If case (2) holds, then a $c \in \mathrm{A}(x)$ as described there is a *sequentiality index* of $f$ at $x$ for $c'$. Such a cell $c$ is a *strict sequentiality index* if there does exist a $y$ that increases $x$ with $c'$ filled in $f(y)$. •

**Definition 1.3.3** A function $f : \mathcal{D}(M) \to \mathcal{D}(M')$ is

- *sequential* if it is continuous and it is sequential at all $x \in \mathcal{D}(M)$.
- *strictly sequential* if it is sequential and, for any $x \in \mathcal{D}(M)$ and $c' \in \mathrm{A}(f(x))$, $f$ has a strict index at $x$ for $c'$, that is– it is always possible to increase $x$ to $y$ such that $c' \in \mathrm{F}(f(y))$.
- *strongly sequential* if it is sequential and, for any $x \in \mathcal{D}(M)$ and $c' \in \mathrm{A}(f(x))$ such that $f$ has a strict index at $x$ for $c'$, this index is unique.

•

**Example 1.3.4** Define the strict or function by:

$$sor : \mathcal{D}(\mathbf{Bool} \times \mathbf{Bool}) \to \mathcal{D}(\mathbf{Bool})$$

$$\begin{aligned} sor(\{\{\mathtt{b} = \mathtt{tt}\},\{\mathtt{b} = \mathtt{tt}\}\}) &= \{\mathtt{b} = \mathtt{tt}\} \\ sor(\{\{\mathtt{b} = \mathtt{tt}\},\{\mathtt{b} = \mathtt{ff}\}\}) &= \{\mathtt{b} = \mathtt{tt}\} \\ sor(\{\{\mathtt{b} = \mathtt{ff}\},\{\mathtt{b} = \mathtt{tt}\}\}) &= \{\mathtt{b} = \mathtt{tt}\} \\ sor(\{\{\mathtt{b} = \mathtt{ff}\},\{\mathtt{b} = \mathtt{ff}\}\}) &= \{\mathtt{b} = \mathtt{ff}\} \end{aligned}$$

---

[4] The definition in [Cur86] uses (1′) instead:

(1′) $c'$ is not filled in $f(y)$ for any $y$ above $x$, that is–

$$\forall y \in \mathcal{D}(M) \;.\; x \subseteq_{\mathcal{D}} y \Rightarrow c' \notin \mathrm{F}(f(y))$$

The overall definitions $(1,2)$ and $(1',2)$ are equivalent, but we prefer to use (1), since it is disjoint from (2).

where, for any omitted cases, *sor* returns $\emptyset$.

*sor* is stable and sequential. It has both `b.1` and `b.2` as sequentiality indices at $\emptyset$ for `b`. •

**Example 1.3.5** Define the parallel or function by:

$$por : \mathcal{D}(\mathbf{Bool} \times \mathbf{Bool}) \to \mathcal{D}(\mathbf{Bool})$$

$$\begin{aligned} por(\langle \quad \emptyset \quad , \{\mathtt{b} = \mathtt{tt}\}) &= \{\mathtt{b} = \mathtt{tt}\} \\ por(\langle \{\mathtt{b} = \mathtt{tt}\}, \quad \emptyset \quad \rangle) &= \{\mathtt{b} = \mathtt{tt}\} \\ por(\langle \{\mathtt{b} = \mathtt{ff}\}, \{\mathtt{b} = \mathtt{ff}\}\rangle) &= \{\mathtt{b} = \mathtt{ff}\} \end{aligned}$$

where any omitted cases may either be inferred by monotonicity, or else are taken to return $\emptyset$.

*por* is neither stable nor sequential — it has no sequentiality index at $\emptyset$ for `b`; and there is no unique minimal state of $\mathbf{Bool} \times \mathbf{Bool}$ below $\langle \{\mathtt{b} = \mathtt{tt}\}, \{\mathtt{b} = \mathtt{tt}\} \rangle$ for which *por* attains $\{\mathtt{b} = \mathtt{tt}\}$, thus no state of $\mathbf{Bool} \times \mathbf{Bool}$ can serve as $M(por, \langle \{\mathtt{b} = \mathtt{tt}\}, \{\mathtt{b} = \mathtt{tt}\} \rangle, \{\mathtt{b} = \mathtt{tt}\})$. •

Sequentiality of a function implies stability. The converse, however, does not hold.

**Example 1.3.6** We define *gf*, a variant of "Gustave's function" (attributed to Berry [Ber78] by Huet [Hue86]) by:

$$gf : \mathcal{D}(\mathbf{Bool} \times \mathbf{Bool} \times \mathbf{Bool}) \to \mathcal{D}(\mathbf{Bool})$$

$$\begin{aligned} gf(\langle \{\mathtt{b} = \mathtt{tt}\}, \langle \{\mathtt{b} = \mathtt{ff}\}, \quad \emptyset \quad \rangle\rangle) &= \{\mathtt{b} = \mathtt{tt}\} \\ gf(\langle \quad \emptyset \quad , \langle \{\mathtt{b} = \mathtt{tt}\}, \{\mathtt{b} = \mathtt{ff}\}\rangle\rangle) &= \{\mathtt{b} = \mathtt{tt}\} \\ gf(\langle \{\mathtt{b} = \mathtt{ff}\}, \langle \quad \emptyset \quad , \{\mathtt{b} = \mathtt{tt}\}\rangle\rangle) &= \{\mathtt{b} = \mathtt{tt}\} \\ gf(\langle \{\mathtt{b} = \mathtt{ff}\}, \langle \{\mathtt{b} = \mathtt{ff}\}, \{\mathtt{b} = \mathtt{ff}\}\rangle\rangle) &= \{\mathtt{b} = \mathtt{ff}\} \end{aligned}$$

where any omitted cases may either be inferred by monotonicity, or else are taken to return $\emptyset$.

Note that *gf* is stable, but not sequential — it has no sequentiality index at $\emptyset$ for `b`. •

The DCDSs and sequential functions form a cartesian category (i.e. product-closed), but it is not cartesian closed (because the collection of all sequential functions from a DCDS to another need not form a DCDS). The same is true for DCDSs and stable functions, and for DCDSs and continuous functions.

## 1.4 Sequential Exponentiation of DCDSs

Having defined sequential functions between DCDSs, we now proceed to define sequential algorithms between DCDSs, which may be viewed abstractly as a sequential function plus a computation strategy for that function. The function is embodied in the algorithm's input-output behavior, and the computation strategy is embodied in the choice of the sequentiality index to be computed.

Operationally, the computation is demand driven so that, for instance, an external observer's information about the result of an application may be gradually increased by filling out the cells of the result state, with each demand for the value of a cell spawning a new computation (which may or may not terminate). The algorithm's events then associate with each current input state $x$ and demand for computation $c'$ a command which either outputs a value $v'$ for the demand, or attempts to increase the information known about the input by issuing a demand for computation of some cell $c$ in the input state. This $c$, naturally enough, is a sequentiality index of the sequential function at $x$, so that the choice of $c$ among all sequentiality indices at $x$ (if not unique) determines the computation strategy. The sub-computation of $c$ in the input state, or in the argument expression in a more general setting, proceeds in the same manner: hence the overall coroutine flavor.

More formally:

**Definition 1.4.1** Let $M$ and $M'$ be DCDSs. We define their *sequential exponentiation* $M \to_{seq} M'$ as follows, where we let $M_{\_\to}$ abbreviate $M \to_{seq} M'$.

**Cells** $C_{M_{\_\to}} = \mathcal{D}_{fin}(M) \times C_{M'}$

A cell consists of an input state $x \in \mathcal{D}_{fin}(M)$, describing the currently known information about the argument, and a request for the computation of a value for a result cell $c' \in C_{M'}$. We may denote a cell $(x, c') \in C_{M_{\_\to}}$ as $xc'$.

**Values** $V_{M_{\_\to}} = \{\textbf{valof } c | c \in C_M\} \cup \{\textbf{output } v' | v' \in V_{M'}\}$

Values are commands, either an output command, or a valof command meant to increase the information available about the argument.

**Events** $E_{M_{\_\to}} = \{(xc', \textbf{valof } c) \in C_{M_{\_\to}} \times V_{M_{\_\to}} | c \in A(x)\} \cup \{(xc', \textbf{output } v') \in C_{M_{\_\to}} \times V_{M_{\_\to}} | (c', v') \in E_{M'}\}$

Output events correspond to events in $M'$.

A valof event is an attempt to increase the current input state at an accessible cell.

**Enablings**

**Valof:** $(xc', \textbf{valof } c) \vdash_{M_{\_\to}} yc'$ iff $y = x \cup_{\mathcal{F}} \{(c, v)\}$ for some $v$.

In a *valof enabling* the current input state is increased by filling $c$.

**Output:** $\{(x_j c'_j, \textbf{output } v'_j)\}_{j=1}^{k} \vdash_{M_{\_\to}} xc'$ iff $\{(c'_j, v'_j)\}_{j=1}^{k} \vdash_{M'} c'$ and $x = \cup_{\mathcal{D}_{j=1}^{k}} x_j$.

An *output enabling* corresponds to an enabling in $M'$, where the enabled cell incorporates all the current state information of the enabling cells.

We call a state of $M \to_{seq} M'$ a *sequential algorithm.*

For $a \in \mathcal{D}(M \to_{seq} M')$ and $x \in \mathcal{D}(M)$, let the *sequential application* of $a$ to $x$, denoted $a \cdot_{seq} x$, be given by

$$a \cdot_{seq} x = \{(c', v') | \exists y.\ (yc', \textbf{output } v') \in a \ \&\ y \subseteq_{\mathcal{D}} x\}.$$

•

Sequential exponentiation preserves well foundedness and stability, and sequential application is well defined. The category of DCDS and sequential algorithms is cartesian closed.

In presenting examples of algorithms, we will make use of a few abbreviations and notational conventions to avoid the proliferation of parentheses and commas. This notation is chosen to display the structure of algorithms in what we hope is a more readable manner. Later we will generalize the notation to the parallel setting. Recall that a sequential algorithm is defined as a state, i.e. a set of events; the events are cells paired with commands; cells are (finite) input states paired with result cells; there are two types of command: **valof** and **output** commands. An event $((x, c'), u)$ of an algorithm $a$ may be abbreviated to $xc' = u$; we may choose to list the member events of $a$ vertically. We also use the display notation in a recursive fashion: within each event $xc' = u$ of $a$ we list the elements of the state component $x$ vertically. Here now are some examples.

**Example 1.4.2** Figures 2 and 3 present two sequential algorithms to compute the strict or function. They have the same input-output function, namely *sor* (example 1.3.4), but differ in their choice of which sequentiality index to compute first.

Figures 4 and 5 present two sequential algorithms for non-strict or. They compute respectively the *lor*– non-strict left or function, and *ror*– non-strict right or function. We demonstrate that they have different input-output functions:

$$\texttt{lsor} \in \mathcal{D}(\textbf{Bool} \times \textbf{Bool} \rightarrow_{seq} \textbf{Bool})$$

$$\texttt{lsor} = \left\{ \begin{array}{ll} \emptyset & \texttt{b=valof b.1} \\ \{\texttt{b.1=tt}\} & \texttt{b=valof b.2} \\ \left\{\begin{array}{l}\texttt{b.1=tt}\\ \texttt{b.2=tt}\end{array}\right\} & \texttt{b=output tt} \\ \left\{\begin{array}{l}\texttt{b.1=tt}\\ \texttt{b.2=ff}\end{array}\right\} & \texttt{b=output tt} \\ \{\texttt{b.1=ff}\} & \texttt{b=valof b.2} \\ \left\{\begin{array}{l}\texttt{b.1=ff}\\ \texttt{b.2=tt}\end{array}\right\} & \texttt{b=output tt} \\ \left\{\begin{array}{l}\texttt{b.1=ff}\\ \texttt{b.2=ff}\end{array}\right\} & \texttt{b=output ff} \end{array} \right\}$$

Figure 2: The left strict or sequential algorithm

$$\texttt{rsor} \in \mathcal{D}(\textbf{Bool} \times \textbf{Bool} \rightarrow_{seq} \textbf{Bool})$$

$$\texttt{rsor} = \left\{ \begin{array}{ll} \emptyset & \texttt{b=valof b.2} \\ \{\texttt{b.2=tt}\} & \texttt{b=valof b.1} \\ \left\{\begin{array}{l}\texttt{b.1=tt}\\ \texttt{b.2=tt}\end{array}\right\} & \texttt{b=output tt} \\ \left\{\begin{array}{l}\texttt{b.1=ff}\\ \texttt{b.2=tt}\end{array}\right\} & \texttt{b=output tt} \\ \{\texttt{b.2=ff}\} & \texttt{b=valof b.1} \\ \left\{\begin{array}{l}\texttt{b.1=tt}\\ \texttt{b.2=ff}\end{array}\right\} & \texttt{b=output tt} \\ \left\{\begin{array}{l}\texttt{b.1=ff}\\ \texttt{b.2=ff}\end{array}\right\} & \texttt{b=output ff} \end{array} \right\}$$

Figure 3: The right strict or sequential algorithm

$$\texttt{lor} \in \mathcal{D}(\textbf{Bool} \times \textbf{Bool} \rightarrow_{seq} \textbf{Bool})$$

$$\texttt{lor} = \left\{ \begin{array}{ll} \emptyset & \texttt{b=valof b.1} \\ \{\texttt{b.1=tt}\} & \texttt{b=output tt} \\ \{\texttt{b.1=ff}\} & \texttt{b=valof b.2} \\ \left\{\begin{array}{l}\texttt{b.1=ff}\\ \texttt{b.2=tt}\end{array}\right\} & \texttt{b=output tt} \\ \left\{\begin{array}{l}\texttt{b.1=ff}\\ \texttt{b.2=ff}\end{array}\right\} & \texttt{b=output ff} \end{array} \right\}$$

Figure 4: The non-strict left or sequential algorithm

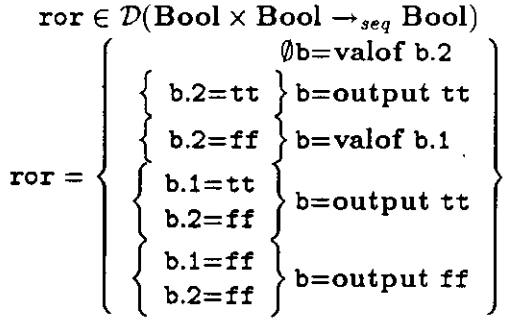



Figure 5: The non-strict right or sequential algorithm

$$\begin{aligned} \mathtt{lor} \cdot_{seq} \{\mathtt{b.1} = \mathtt{tt}\} &= \{\mathtt{b} = \mathtt{tt}\} \\ \mathtt{ror} \cdot_{seq} \{\mathtt{b.1} = \mathtt{tt}\} &= \emptyset \end{aligned}$$

•

And finally we present Berry and Curien's formalization of the view of a sequential algorithm as a pair consisting of a sequential function together with a way of computing it:

**Proposition 1.4.3** *For $M$ and $M'$ DCDSs,*

- *Let* $\mathrm{SEQ}(M, M')$ *be the sequential functions from* $\mathcal{D}(M)$ *to* $\mathcal{D}(M')$.
- *Let* $\leq^s$ *be the stable ordering of stable functions, i.e. for any stable functions* $f$ *and* $g$ *from* $(D, \leq)$ *to* $(D', \leq')$, $f \leq^s g$ *iff*

  $$\forall x \in D \;.\; f(x) \leq' g(x)$$

  *and*

  $$\forall x \in D \;.\; x' \leq' f(x) \Rightarrow M(f, x, x') = M(g, x, x').$$

- *And let* $=^e$ *be extensional equality of sequential algorithms, i.e. for all* $a, a' \in \mathcal{D}(M \to_{seq} M')$, $a =^e a'$ *iff* $a \cdot_{seq} x = a' \cdot_{seq} x$ *for all* $x \in \mathcal{D}(M)$.

*Then* $(\mathcal{D}(M \to_{seq} M')/ =^e, \subseteq_{\mathcal{D}} / =^e)$ *and* $(\mathrm{SEQ}(M, M'), \leq^s)$ *are isomorphic, and, in particular, application is sequential in its second argument.*

So far we have summarized enough of the definitions and results of Berry and Curien's work on sequentiality to establish a coherent background from which to develop our ideas on parallelism. We will begin to explain the genesis of our ideas in the next section.

# 2 Parallel Exponentiation of DCDSs

We would now like to generalize the sequential exponentiation so as to be able to express algorithms for non-sequential functions, such as *por*, while retaining as far as possible suitable analogues to the other properties of sequential exponentiation. Why, then, are sequential algorithms sequential? From an operational viewpoint, the reason seems to be that a valof command may only start

one sub-computation, and only after it returns may the main computation proceed. If that sub-computation happens to be non-terminating then the whole computation is non-terminating. This is precisely why the cell named by a valof is an index of sequentiality.

A natural first step towards a generalization, then, would be to have the valof command start a number of sub-computations, with the understanding that not all of them have to terminate before the main computation may resume. Suppose, then, we allow commands of the form **valof** $s$, where $s$ is a set of cells of the relevant DCDS. For the *por* function, using this informal extension of the notation for sequential algorithms, this would result in the event

$$\emptyset\texttt{b=}\mathbf{valof}\{\texttt{b.1},\texttt{b.2}\}\ ,$$

describing the intention to begin two sub-computations, one for each of the cells b.1 and b.2. A problem now seems to be that there is no way to determine what responses for the values of b.1 and b.2 would let the main computation proceed without, perhaps, waiting for the termination of all the sub-computations. For instance, in a por algorithm, we would like to allow the main computation to proceed (issue the output value tt) once either of the two sub-computations terminates with the value tt; there is no need to wait for termination of both sub-computations when this happens. Thus, in the por example, we would like to enable the input states $\{\texttt{b.1} = \texttt{tt}\}$, $\{\texttt{b.2} = \texttt{tt}\}$, and $\{\texttt{b.1} = \texttt{ff}, \texttt{b.2} = \texttt{ff}\}$, obtaining an algorithm that looks like

$$\left\{\begin{array}{l} \emptyset\texttt{b=}\mathbf{valof}\ \{\texttt{b.1},\texttt{b.2}\} \\ \left\{\begin{array}{l}\texttt{b.1=tt}\\ \texttt{b.2=tt}\end{array}\right\}\ \texttt{b=}\mathbf{output}\ \texttt{tt} \\ \left\{\begin{array}{l}\texttt{b.1=ff}\\ \texttt{b.2=ff}\end{array}\right\}\ \texttt{b=}\mathbf{output}\ \texttt{ff} \end{array}\right\}$$

But there are other states which may also claim to be enabled, if we were to continue to use the notion of enabling that was presented earlier for sequential algorithms. Some of these states are superfluous, in the sense that, for instance, addition of the event

$$\left\{\begin{array}{l}\texttt{b.1=tt}\\ \texttt{b.2=ff}\end{array}\right\}\ \texttt{b=}\mathbf{output}\ \texttt{tt}$$

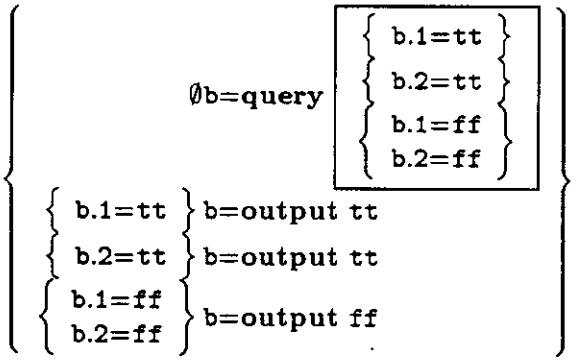
to the algorithm above yields a different algorithm with the same intended operational semantics, possibly impairing full-abstraction. The extra event is superfluous in that it contains information (about b.2) that is not crucial to the operational behavior given the rest of the information that is known (here, $\texttt{b.1} = \texttt{tt}$); the algorithm already describes what should happen next when this much is known.

Similarly, we argue that "partial" states should not be deemed to be enabled, where by partiality we mean that the main computation may not yet proceed, but must continue to wait for additional sub-computations to terminate. In our example these states would be $\{\texttt{b.1} = \texttt{ff}\}$ and $\{\texttt{b.2} = \texttt{ff}\}$; neither of these states is sufficient for triggering a command in the por algorithm. By not regarding such states as enabled in the algorithm, we do not have to include events like $\{\texttt{b.1} = \texttt{ff}\}\texttt{b} = \mathbf{valof}\ \texttt{b.2}$ in the description of the por algorithm.

In any case, using valof commands simply involving sets of cells is not adequate because it does not include all of the necessary information. The solution we adopt is a further generalization of the valof command. Rather than having it ask about values of cells or even sets of cells, we have it proposing alternative *functional sets of events*, or *branches*, of which one corresponding to the input is to be chosen. The operational reading should still be that sub-computations are started for

each of the cells mentioned, but now the point at which the main computation may be resumed and the remaining uncompleted sub-computations may be discarded is precisely defined: each branch specifies a sufficient condition for such a resumption. For our example we need three branches, $\{\mathtt{b.1} = \mathtt{tt}\}$, $\{\mathtt{b.2} = \mathtt{tt}\}$, and $\{\mathtt{b.1} = \mathtt{ff}, \mathtt{b.2} = \mathtt{ff}\}$, corresponding to the input states we would like to enable.

Our tentative por algorithm now takes the form




where the box encloses the set of branches, or *query*. We also change the name of the command from **valof** to **query** in order to emphasize the distinction. The query command in its final form will turn out to be a combination of generalized valof and output commands.

We impose the restriction that a query contain no additional superfluous branches, such as $\{\mathtt{b.1} = \mathtt{tt}, \mathtt{b.2} = \mathtt{ff}\}$, for our example. Such branches make no sense operationally, and, as pointed out above, may impair full abstraction. The superfluous branches are precisely those which are supersets of other branches, and so we require of a query that it have no two branches that are related by set inclusion. We call queries satisfying this restriction *trim*, and formulate later the notion of a *trim powerdomain* of functional sets of events, to which the queries belong.

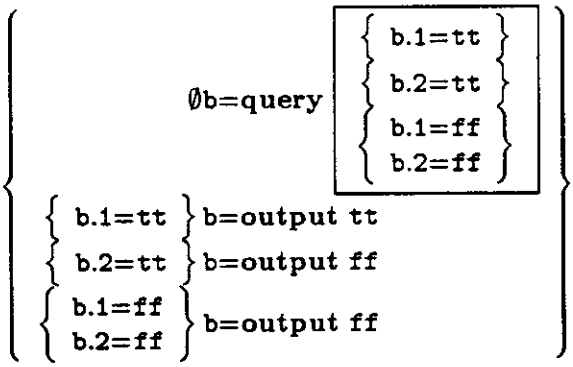
We will also impose another intuitively natural restriction on queries: that the empty branch is not allowed, since a query containing the empty branch (and by trimness this would be the only branch) need not be issued at all.

We now address the problem of selecting a branch. Obviously, there is no problem for branches which are not consistent[5], since only one of them may be satisfied by the input state. But several consistent branches may match a given input state. If each such branch enables a distinct cell, then we could associate with it a distinct command, and thus obtain a non-deterministic relational semantics[6]. Consider, for example, the following variation on the tentative por algorithm (which is no longer intended to compute *por* proper):

---

[5]Consistency of branches is taken to mean consistency in $(\mathcal{F}_{fin}(M), \subseteq_{\mathcal{F}})$ for the appropriate $M$. That is, two branches are consistent iff their union is functional.

[6]Assuming that the selection of the branch taken is non-deterministic.

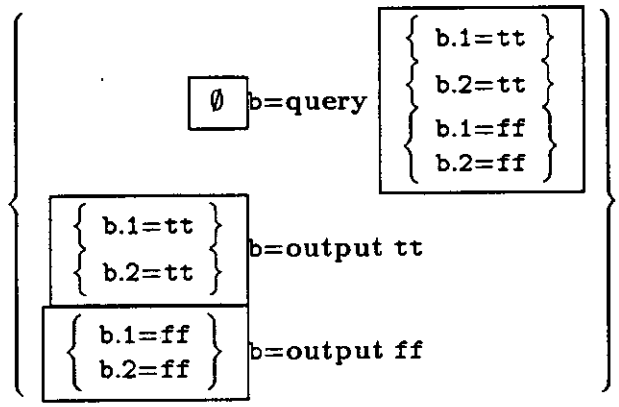



For an input state {b.1 = tt, b.2 = tt} the resulting output state could be either {b = tt}, or {b = ff}, depending on which of the two relevant branches was chosen.

We choose, however, to limit ourselves to a deterministic, functional semantics. We do this by not having each *branch* enable a cell, but, rather, by having each *equivalence class* of branches of a query enable a cell, with *equivalence* defined to be the transitive closure of the (restriction to the query of the) consistency relation. Transitivity is needed to ensure that no matter which two consistent branches satisfy the input, they will be bundled together for the purpose of enabling a common cell.

Therefore the exponentiation cells, rather than consisting of a *state*, representing the current input state, and a result cell, now consist of a *class* and a result cell; a class consists of *several* (representations of) current input states, each containing the information of a branch from the chosen equivalence class and of a previous input state representation, from the previously enabled cell. This conglomeration of equivalent current input states makes it impossible (for the algorithm, or any external observer of its behavior) to determine which specific branch was taken, and thus we achieve a deterministic functional semantics, without making any assumptions about the selection of a branch among several consistent branches. We believe that this is a crucial aspect, since, in more general terms, if the algorithm, or computational agent, has a means of telling apart consistent possibilities which are satisfied by the environment, then it may reflect non-determinism in scheduling by the environment as non-deterministic behavior on its part. We would argue that it is necessary to equate all such equivalent possibilities in order to get a properly deterministic notion of behavior.

Here, then, is our next tentative version of por, where the two consistent branches {b.1 = tt} and {b.2 = tt} enable the same cell:




Classes are also enclosed in boxes.

Analogously to queries, and for the same reasons, we require that classes belong to a *trim powerdomain* of states.

The introduction of classes with multiple states, though, introduces a new consideration, related to *currying*. The elements of a class of an uncurried algorithm will be compound states of a product of DCDSs. When currying such an algorithm we must take care to preserve the relationship between components of these compound states, so that, for instance, uncurrying would yield back the original algorithm. The currying must take place at the level of the class element. Each element of a class must therefore include, in a *suitable representation*, all the information content that may be necessary for currying. That is, class elements must encompass state components from all the possible arguments. Our definition of exponentiation must then provide such a representation, and currying and uncurrying will be simple transformations of the representation.

We now proceed to define formally the concepts behind the above intuitions.

## 2.1 Trim Powerdomains

As mentioned above, we will not allow just any collection of states or functional sets of events to serve as classes and queries, respectively. We impose a requirement that, for such a collection to be valid, it must be non-empty, and no two elements of it may be related (by set inclusion, in our concrete case).

To motivate an ordering for our powerdomain construction, consider classes as assertions about the input state, with each class a disjunction of conjunctions, so that the information about the input state may be increased by reducing the number of disjuncts and/or increasing the number of conjuncts in one or more of the disjuncts.

Here is the formalization, for a general poset:

**Definition 2.1.1** For $(D, \leq)$ a poset, define its *trim powerdomain*, denoted by $(\mathcal{P}_t(D), \sqsubseteq)$, as follows:

- $p \in \mathcal{P}_t(D)$ iff
  - $p \subseteq D$ &
  - $p$ is non-empty &
  - **(Trim)**: No two elements of $p$ are related by $\leq$, that is–

    $$x_1, x_2 \in p \;\&\; x_1 \leq x_2 \Rightarrow x_1 = x_2$$
- $p_1 \sqsubseteq p_2$ iff $\forall x_2 \in p_2 \;.\; \exists x_1 \in p_1 \;.\; x_1 \leq x_2$

•

The $\sqsubseteq$ relation is the Smyth preorder on $\mathcal{P}(D)$, and it is a partial order on $\mathcal{P}_t(D)$. If $\perp_D$ is a least element of $(D, \leq)$ then $(\mathcal{P}_t(D), \sqsubseteq)$ has a least element $\{\perp_D\}$ .

Our construction of the trim powerdomain is essentially identical to the Smyth powerdomain [Smy78], but rather than dealing directly with the equivalence classes, we take canonical representatives, chosen for their trimness. It is important to note, however, that we allow infinite subsets of $D$ in our powerdomain, because natural algorithms such as the identity algorithm on Nat involve infinite queries.

**Definition 2.1.2** For $(D, \leq)$ a poset and $Y \subseteq D$, $Y$ is *consistent* in $D$, denoted $\Uparrow_D Y$, iff it has an *upper bound* in $D$. This also defines a binary *consistency* relation $\Uparrow_D$, on $D$, namely, $y_1 \Uparrow_D y_2$ iff $\Uparrow_D \{y_1, y_2\}$. •

**Definition 2.1.3** For $S$ a set and $P \subseteq \mathcal{P}(S)$ a collection of subsets of $S$, define

$$\Psi P = \{X \subseteq \cup P | \forall p \in P \ . \ p \cap X \neq \emptyset\}$$

$\Psi P$ is the collection of *cross-sections*, or possible representative choices, of the family $P$. •

Consistency for a subset of a trim powerdomain coincides with the property of having a consistent cross-section, in the following sense:

**Lemma 2.1.4 (Consistency Lemma)** *For any* $P \subseteq \mathcal{P}_t(D)$,

$$\Uparrow_{\mathcal{P}_t(D)} P \quad \textit{iff} \quad \exists X \in \Psi P \ . \ \Uparrow_D X.$$

We define a useful map that, for $p \in \mathcal{P}(D)$, returns the set of minimal elements of $p$.

**Definition 2.1.5** For $(D, \leq)$ a poset, define

$$\text{trim} : \mathcal{P}(D) \rightarrow \mathcal{P}_t(D) \cup \{\emptyset\}$$

by

$$\forall p \subseteq D \ . \ \text{trim}(p) = \{x \in p | \forall x' \in p \ . \ x' \leq x \Rightarrow x' = x\}$$

•

Note that if $(D, \leq)$ is well founded then, for any non-empty subset $P$ of $D$, $\text{trim}(P) \in \mathcal{P}_t(D)$.

We now use this map to get an explicit formulation of lubs and glbs in the trim powerdomain. We limit ourselves to well founded posets.

**Proposition 2.1.6** *For* $(D, \leq)$ *a well founded poset, and any non-empty subset* $P$ *of the trim powerdomain* $\mathcal{P}_t(D)$, $\sqcap P$ *exists, and* $\sqcap P = \text{trim}(\cup P)$.

**Proposition 2.1.7** *If* $(D, \leq)$ *is a consistently complete and well founded poset then* $(\mathcal{P}_t(D), \sqsubseteq)$ *is consistently complete. Moreover, for any consistent subset* $P \subseteq \mathcal{P}_t(D)$,

$$\sqcup P = \text{trim}(\{\vee X | X \in \Psi P \ \& \ \Uparrow_D X\}),$$

*where* $\vee$ *is the lub in* $(D, \leq)$.

Note that for $P = \emptyset$ the above reduces to $\sqcup\emptyset = \{\vee\emptyset\} = \{\perp_D\} = \perp_{\mathcal{P}_t(D)}$.

## 2.2 A Few Preliminaries

For the purposes of defining exponentiation we will mainly be interested in the trim powerdomains formed over $(\mathcal{F}_{fin}(M), \subseteq_{\mathcal{F}})$ and $(\mathcal{D}_{fin}(M), \subseteq_{\mathcal{D}})$ for a DCDS $M$. It is from these trim powerdomains, denoted $(\mathcal{P}_t(\mathcal{F}_{fin}(M)), \sqsubseteq_{\mathcal{F}})$ and $(\mathcal{P}_t(\mathcal{D}_{fin}(M)), \sqsubseteq_{\mathcal{D}})$, that we will draw our queries and classes, respectively. We use similarly subscripted versions of $\sqcup$ for the lubs in these trim powerdomains. We also use additional trim powerdomains, such as $(\mathcal{P}_t(\mathcal{E}(M)), \sqsubseteq)$, to give more general definitions.

Note that for any DCDS $M$, the posets $(\mathcal{E}_{fin}(M), \subseteq)$, $(\mathcal{F}_{fin}(M), \subseteq_{\mathcal{F}})$ and $(\mathcal{D}_{fin}(M), \subseteq_{\mathcal{D}})$ are well founded, due to finiteness, and thus we can use the results relating to lub and glb in the trim powerdomains. When they exist, the lubs for the trim powerdomains built over these three posets coincide (just observe that the lubs in the underlying posets are all set unions).

We give here some preliminaries relating to these trim power domains.

We extend the notions of a cell being filled, enabled and accessible in a set of events to collections of sets of events in a natural way.

**Definition 2.2.1** For $M$ a CDS and $q \in \mathcal{P}_t(\mathcal{E}(M))$:

- $\mathrm{F}(q) = \cup_{y \in q} \mathrm{F}(y)$. A cell is *filled* in $q$ iff it is filled in any of $q$'s elements.
- $\mathrm{E}(q) = \cap_{y \in q} \mathrm{E}(y)$. A cell is *enabled* in $q$ iff it is enabled in all of $q$'s elements.
- $\mathrm{A}(q) = \mathrm{E}(q) \setminus \mathrm{F}(q)$. A cell is *accessible* in $q$ iff it is enabled in $q$, and is not filled in $q$. An equivalent definition is $\mathrm{A}(q) = \cap_{y \in q} \mathrm{A}(y)$. A cell is accessible in $q$ iff it is accessible in all of $q$'s elements.

We define, in general, equivalence over a subset of a poset, meant to capture the above mentioned equivalence of branches in a query, the query being regarded as a subset of the poset $(\mathcal{F}_{fin}(M), \subseteq_{\mathcal{F}})$ for the appropriate $M$.

**Definition 2.2.2** For $(D, \leq)$ a poset, $Y \subseteq D$, let $\Uparrow_Y$ be the restriction of the binary consistency relation $\Uparrow_D$ to $Y$, i.e., $y_1 \Uparrow_Y y_2$ iff $y_1 \Uparrow_D y_2$ and $y_1, y_2 \in Y$.

Define the relation of *equivalence* on $Y$, denoted $\approx_Y$, to be $\Uparrow_Y^*$, the transitive closure of $\Uparrow_Y$. Thus, $y \approx_Y y'$ iff there is an $n \geq 0$ and a chain in $Y$ such that

$$y = y_0 \Uparrow_Y y_1 \Uparrow_Y \cdots \Uparrow_Y y_n = y'.$$

This relation $\approx_Y$ is indeed an equivalence relation on $Y$. We denote by $[y]_Y$ the *equivalence class* of $y$ in $Y$, and by $Y/_{\approx}$ the collection of all such equivalence classes.

Now define a *covering* relation that is intended to capture the enabling of a cell by an equivalence class of branches in a query, as proposed in the discussion above.

**Definition 2.2.3** For $M$ a DCDS, and $q_1, q_2, q \in \mathcal{P}_t(\mathcal{F}_{fin}(M))$, we say that $q_2$ *$q$-covers* $q_1$, or that $q_1$ is *$q$-covered* by $q_2$, denoted $q_1 \prec_q q_2$, iff

$$\exists y \in q \;.\; q_2 = q_1 \sqcup_{\mathcal{F}} [y]_q$$

where $[y]_q$ is the $\approx_q$ equivalence class of $y$ in $q$, as defined above.

We will use the covering relation solely in cases where $q_1$ and $q_2$ are actually classes. The following proposition states that if $p_1 \prec_q p_2$ with $p_1$ a class and $q$ a query about cells that are accessible from $p_1$, then $p_2$ is itself a class, and, further, we have a simplified explicit formulation of $p_2$ that does not make use of the trim mapping.

**Proposition 2.2.4** *If $p_1 \in \mathcal{P}_t(\mathcal{D}_{fin}(M))$, $q \in \mathcal{P}_t(\mathcal{F}_{fin}(M))$, $\mathrm{F}(q) \subseteq \mathrm{A}(p_1)$, and $p_1 \prec_q p_2$, then $p_2 \in \mathcal{P}_t(\mathcal{D}_{fin}(M))$, and there is a branch $y \in q$ such that*

$$p_2 = \{x_1 \sqcup_{\mathcal{F}} y' | x_1 \in p_1, y' \in [y]_q\}.$$

**Example 2.2.5** For the above tentative example for `por`, the query, call it $q$, has two equivalence classes,

$$\{\{\mathtt{b.1} = \mathtt{tt}\}, \{\mathtt{b.2} = \mathtt{tt}\}\}$$

and

$$\{\{\mathtt{b.1} = \mathtt{ff}, \mathtt{b.2} = \mathtt{ff}\}\}$$

participating, respectively, in the following covering relations:

$$\{\emptyset\} \prec_q \{\{\mathtt{b.1} = \mathtt{tt}\}, \{\mathtt{b.2} = \mathtt{tt}\}\}$$

and

$$\{\emptyset\} \prec_q \{\{\mathtt{b.1} = \mathtt{ff}, \mathtt{b.2} = \mathtt{ff}\}\}$$

And finally, before turning to the exponentiation itself, we define the *representation* and *base* maps, needed to facilitate currying.

**Definition 2.2.6** We define two maps, rep and base, from CDSs to CDSs. If a CDS $M$ is not an exponentiation, we call it a *basic* CDS, and let $\mathrm{rep}(M) = \mathbf{Null}$ and $\mathrm{base}(M) = M$. For an exponentiation $M \to M'$, we let

$$\begin{aligned} \mathrm{rep}(M \to M') &= M \times \mathrm{rep}(M') \\ \mathrm{base}(M \to M') &= \mathrm{base}(M'). \end{aligned}$$

Strictly speaking, we should define an algebra of CDS names to serve as a type system, so that the above maps may be defined by structural induction. Different interpretations of $\to$ would then let us talk about sequential and parallel exponentiation. But, having pointed this out, we proceed on the assumption that DCDSs are finitely generated, and that $\mathrm{rep}(M')$ and $\mathrm{base}(M')$ may be taken as inductively determined for the purpose of forming $M \to M'$.

It is the definition of rep that calls for $\times$ associating to the right. In turn, we define rep this way to correspond to the argument structure of the exponentiation, in that for instance if $M_0$ is basic, the DCDS $M_k \to \cdots \to M_1 \to M_0$ has for its representation the DCDS $M_k \times \cdots M_1 \times \mathbf{Null}$ and $M_0$ for its base.

## 2.3 Exponentiation

We may now give the generalized definition of exponentiation.

**Definition 2.3.1** Let $M$ and $M'$ be DCDSs. Let $M_{\to}$ abbreviate $M \to M'$, let $M_{\times}$ abbreviate $\mathrm{rep}(M \to M')$ and let $M_0$ abbreviate $\mathrm{base}(M \to M')$. Define the DCDS $M \to M'$ as follows:

**Cells** $\mathrm{C}_{M_{\to}} = \mathcal{P}_t(\mathcal{D}_{fin}(M_{\times})) \times \mathrm{C}_{M_0}$

A cell of the exponentiation consists of a trim and non-empty class of finite states of the representation DCDS, and a basic cell.

*Notation:* We may denote an element $(p, c)$ of $\mathrm{C}_{M_{\to}}$ as $pc$, and an element $(\bar{x}, c)$ of $\mathcal{D}_{fin}(M_{\times}) \times \mathrm{C}_{M_0}$ as $\bar{x}c$.

**Values** $V_{M_{\to}} = \{\mathbf{query}\ q | q \in \mathcal{P}_t(\mathcal{F}_{fin}(M_{\times})) \ \&\ \emptyset \notin q\} \cup \{\mathbf{output}\ v | v \in V_{M_0}\}$

Values are commands, either an output command involving a basic value, or a query command involving finite functional sets of events — branches — of the representation DCDS. A query must be trim and non-empty, and may not contain the empty branch.

**Events** $E_{M_-} = \{(pc, \textbf{query } q) \in C_{M_-} \times V_{M_-} | F(q) \subseteq A(p)\}$
$\cup \{(pc, \textbf{output } v) \in C_{M_-} \times V_{M_-} | (c, v) \in E_{M_0}\}$

Output events correspond to events in the base DCDS.

A query event for a cell $pc$ may only issue a query $q$ that concerns cells that are accessible from $p$; this is a request to increase the information known about the input state.

**Enablings**

**Query:** $(pc, \textbf{query } q) \vdash_{M_-} p'c$ iff $p \prec_q p'$.

A *query enabling* is determined by the covering relation. The enabled cell increases the enabling cell by an equivalence class of the query.

**Output:** $\{(p_j c_j, \textbf{output } v_j)\}_{j=1}^k \vdash_{M_-} pc$ iff $\{(c_j, v_j)\}_{j=1}^k \vdash_{M_0} c$ and $p = \sqcup_{\mathcal{D}} {}_{j=1}^k p_j$.

An *output enabling* corresponds to an enabling in the base DCDS. The enabled cell incorporates all the information of the enabling cells.

We call a state of $M \to M'$ a *parallel algorithm*, or just an *algorithm*.

Note that an initial cell of $M \to M'$ is of the form $\{\emptyset\}c$, with $c$ an initial cell of $M_0$.

Our usage of a representation DCDS is necessitated by our demand to represent together information about all possible arguments, so that base($M$) must never be an exponentiation, for any $M$. There is, in general, a difference between $M \to M'$ and rep($M \to M'$) $\to$ base($M \to M'$), although they are isomorphic, and will be related by the currying transformations.

Exponentiation preserves well foundedness:

**Proposition 2.3.2** *For any DCDSs $M$ and $M'$, $M \to M'$ is a well founded CDS.*

The proof relies on well foundedness of $(\mathcal{P}_t(\mathcal{D}_{fin}(\text{rep}(M \to M'))), \sqsubseteq_{\mathcal{D}})$, and this relies on the finiteness of the states involved. (The possible existence of infinite sets in a trim powerdomain is irrelevant to this consideration.)

Before proceeding to show that exponentiation preserves stability, we give a few examples.

**Example 2.3.3** Here are the representations and bases for several DCDSs:

$$\begin{aligned}
\text{rep}(\textbf{Bool}) &= \textbf{Null} \\
\text{base}(\textbf{Bool}) &= \textbf{Bool} \\
\\
\text{rep}(\textbf{Bool} \to \textbf{Bool}) &= \textbf{Bool} \times \textbf{Null} \\
\text{base}(\textbf{Bool} \to \textbf{Bool}) &= \textbf{Bool} \\
\\
\text{rep}(\textbf{Bool} \to \textbf{Bool} \to \textbf{Bool}) &= \textbf{Bool} \times \textbf{Bool} \times \textbf{Null} \\
&= \textbf{Bool} \times (\textbf{Bool} \times \textbf{Null}) \\
\text{base}(\textbf{Bool} \to \textbf{Bool} \to \textbf{Bool}) &= \textbf{Bool} \\
\\
\text{rep}(\textbf{Bool} \times \textbf{Bool} \to \textbf{Bool}) &= (\textbf{Bool} \times \textbf{Bool}) \times \textbf{Null} \\
\text{base}(\textbf{Bool} \times \textbf{Bool} \to \textbf{Bool}) &= \textbf{Bool}
\end{aligned}$$

**Example 2.3.4** Figure 6 finally presents the parallel or algorithm.

Here we extend the display notation as follows. Classes and queries are framed in boxes, with their elements, compound sets of events, stacked vertically. We use a shorthand notation for pairs,

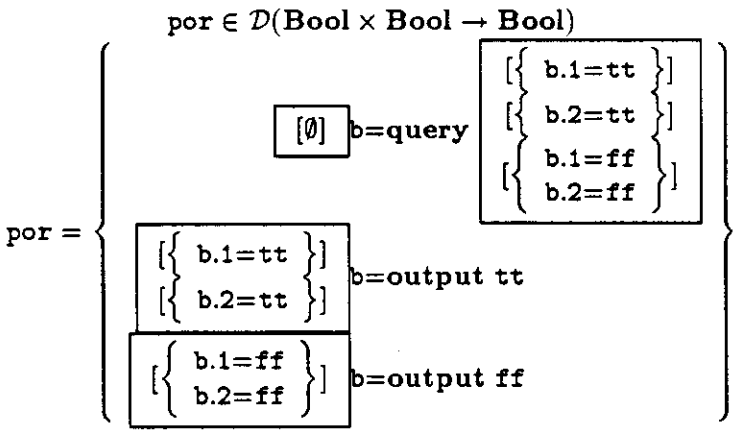
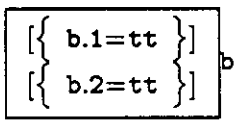



Figure 6: The parallel or algorithm

denoting $\emptyset \in \mathcal{E}(\mathbf{Null})$ as [], and denoting $\langle y_0, [y_1, \ldots y_d] \rangle$ as $[y_0, y_1, \ldots y_d]$ for $d \geq 0$. We will only use this notation at the topmost level– elements of classes and queries– so as to stress our use of a representation, and to differentiate between pairs in $\mathcal{D}_{fin}(\text{rep}(M \rightarrow M'))$ and pairs that arise when $M$ itself is a product.

Note that [{ b.1=tt }] and [{ b.2=tt }] are consistent and hence equivalent branches in the query. Thus they are "coerced" into enabling the same cell




which ensures that the algorithm cannot determine which of the branches was satisfied, and must act deterministically. In particular, we have no way of expressing the non-deterministic algorithm suggested on page 14. •

**Example 2.3.5** Figure 7 presents the algorithm for the function *gf*. For convenience of presentation, we use here a shorthand notation for cells of **Bool** × **Bool** × **Bool**, with b1 for b.1, b2 for b.1.2 and b3 for b.2.2.

Note that equivalence is discrete on the query of **gf**, i.e. all pairs of branches in the query are inconsistent. Here is a variant of *gf* for which this is no longer the case:

$$gf' : \mathcal{D}(\mathbf{Bool} \times \mathbf{Bool} \times \mathbf{Bool}) \rightarrow \mathcal{D}(\mathbf{Bool})$$

$$\begin{aligned}
gf'(\langle\{\mathtt{b}=\mathtt{tt}\},\langle\{\mathtt{b}=\mathtt{ff}\}, \quad \emptyset \quad \rangle\rangle) &= \{\mathtt{b}=\mathtt{tt}\} \\
gf'(\langle \quad \emptyset \quad ,\langle\{\mathtt{b}=\mathtt{tt}\},\{\mathtt{b}=\mathtt{ff}\}\rangle\rangle) &= \{\mathtt{b}=\mathtt{tt}\} \\
gf'(\langle\{\mathtt{b}=\mathtt{ff}\},\langle \quad \emptyset \quad ,\{\mathtt{b}=\mathtt{tt}\}\rangle\rangle) &= \{\mathtt{b}=\mathtt{tt}\} \\
gf'(\langle\{\mathtt{b}=\mathtt{ff}\},\langle\{\mathtt{b}=\mathtt{tt}\}, \quad \emptyset \quad \rangle\rangle) &= \{\mathtt{b}=\mathtt{tt}\} \\
gf'(\langle \quad \emptyset \quad ,\langle\{\mathtt{b}=\mathtt{ff}\},\{\mathtt{b}=\mathtt{tt}\}\rangle\rangle) &= \{\mathtt{b}=\mathtt{tt}\} \\
gf'(\langle\{\mathtt{b}=\mathtt{tt}\},\langle \quad \emptyset \quad ,\{\mathtt{b}=\mathtt{ff}\}\rangle\rangle) &= \{\mathtt{b}=\mathtt{tt}\} \\
gf'(\langle\{\mathtt{b}=\mathtt{ff}\},\langle\{\mathtt{b}=\mathtt{ff}\},\{\mathtt{b}=\mathtt{ff}\}\rangle\rangle) &= \{\mathtt{b}=\mathtt{ff}\}
\end{aligned}$$

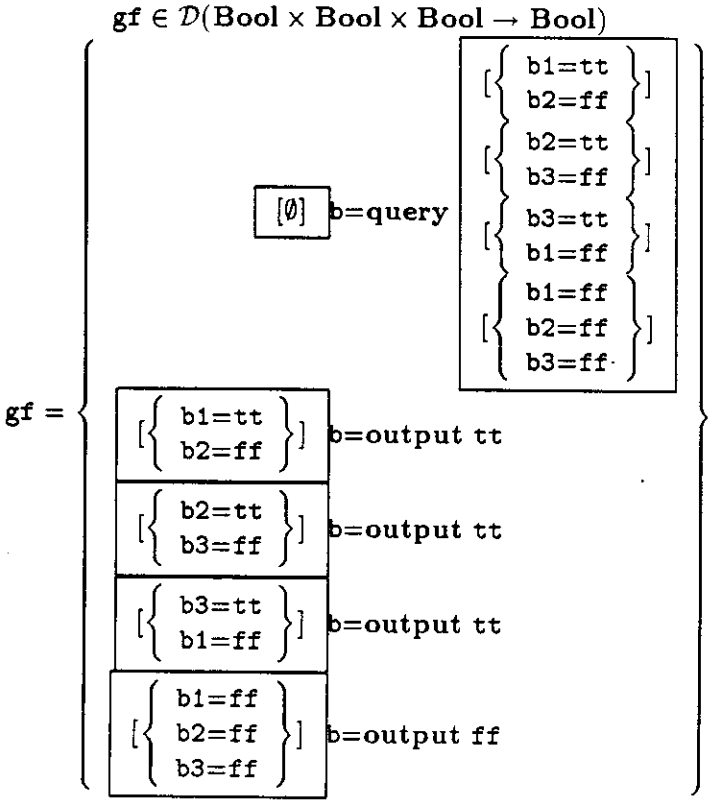



Figure 7: Algorithm for *gf*

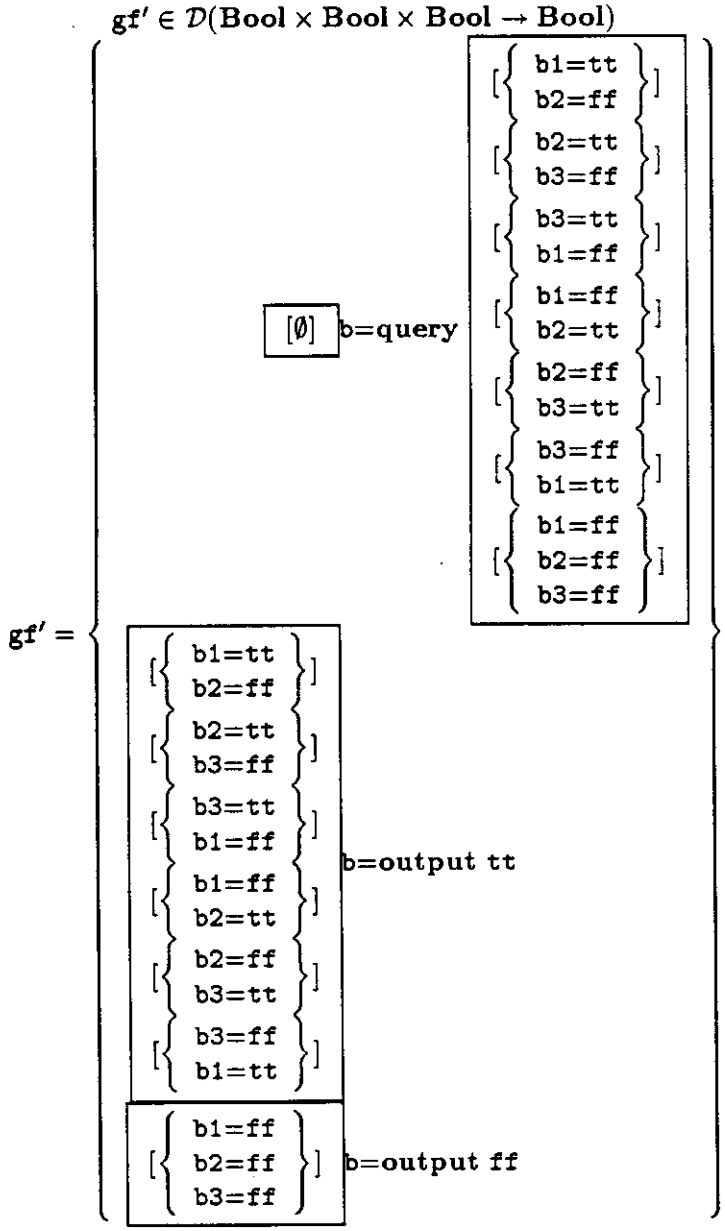



Figure 8: Algorithm for $gf'$

where any omitted cases may either be inferred by monotonicity, or else are taken to return $\emptyset$. Figure 8 presents the algorithm for $gf'$. Note that here we have branches which are equivalent, but not consistent. The first branch is consistent with the fifth and sixth; the second branch is consistent with the fourth and sixth; the third with the fourth and fifth; and hence, the first six branches of the query are equivalent. Note that $gf'$ is not sequential — just like $gf$ it has no sequentiality index at $\emptyset$ — but, in contrast to $gf$, it is also not stable — there is no unique minimal state below $\langle\{\mathtt{b} = \mathtt{tt}\}, \{\mathtt{b} = \mathtt{ff}\}\rangle, \{\mathtt{b} = \mathtt{ff}\}\rangle$ for which $gf'$ attains $\{\mathtt{b} = \mathtt{tt}\}$. This observation is not accidental, as we will show when characterizing stability of an algorithm's input-output function in future work.

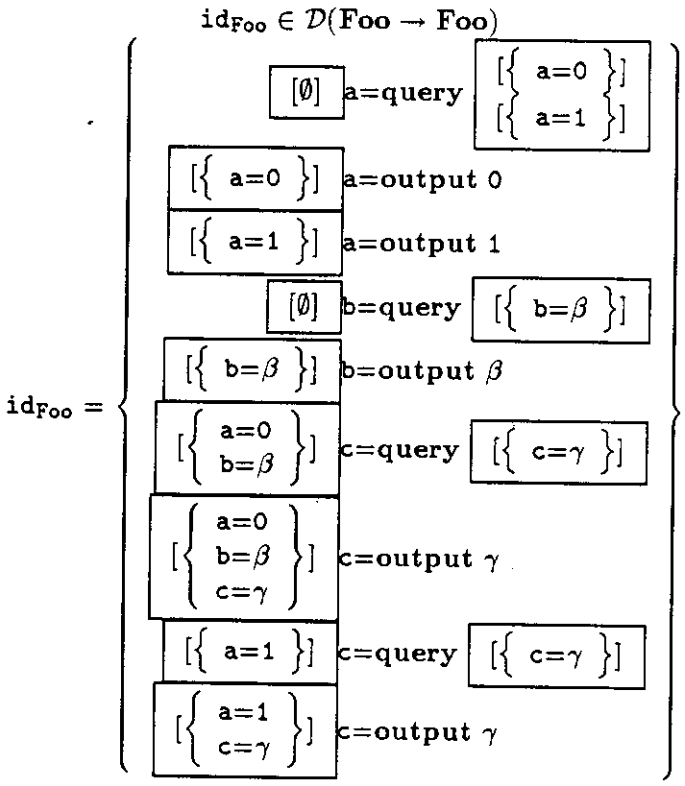



Figure 9: The identity algorithm on **Foo**

**Example 2.3.6** Figure 9 presents the identity algorithm on the DCDS **Foo** (see example 1.1.11).

**Example 2.3.7** Figure 10 presents the identity algorithm on the DCDS **Nat**. Note that this involves a query containing an infinite number of (mutually inconsistent) branches, and an infinite number of output events.

## 2.4 The Tree Lemma

An algorithm $a \in \mathcal{D}(M \to M')$ has a well defined structure when considered as a directed graph, which we call the *precedence graph* of $a$, with enabled cells as nodes and the immediate precedence

$$\mathtt{id_{Nat}} \in \mathcal{D}(\mathbf{Nat} \to \mathbf{Nat})$$

$$\mathtt{id_{Nat}} = \left\{ \boxed{[\emptyset]}\ \mathtt{n{=}query}\ \left\{ [\{\ \mathtt{n}{=}k\ \}] \mid k \in \mathbb{N} \right\} \right\} \cup (\cup_{k\in\mathbb{N}} \left\{ \boxed{[\{\ \mathtt{n}{=}k\ \}]}\ \mathtt{n{=}output}\ k \right\})$$

Figure 10: The identity algorithm on **Nat**

relation $pc \ll_{M\to M'} p'c'$ giving a directed edge from $pc$ to $p'c'$, if they are enabled in $a$. If $M \to M'$ is well founded the graph is acyclic, hence a *precedence DAG*. By functionality of $a$ we may take for a filled cell the unique event associated with it.

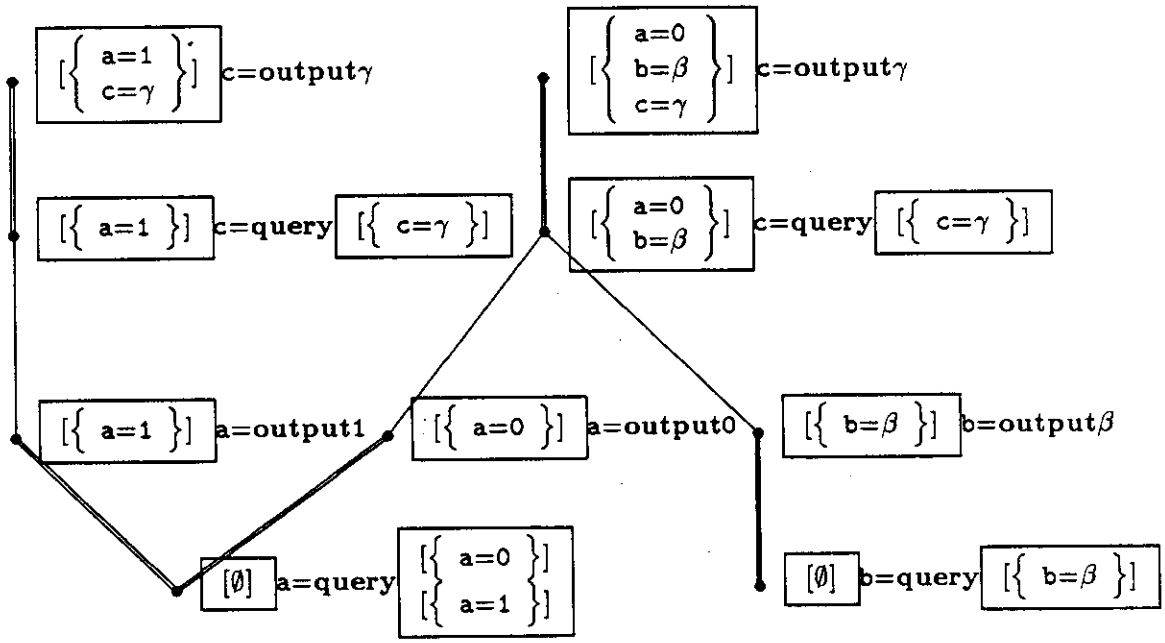



Figure 11: The precedence DAG of $\mathtt{id_{Foo}}$

**Example 2.4.1** Figure 11 presents the precedence DAG of the identity algorithm on the DCDS **Foo**. Arcs corresponding to query enablings are depicted by double lines. Nodes are labelled by events; in this case all enabled cells are filled. •

The precedence graph of an algorithm contains some interesting sub-graphs. In particular, it is worth looking at the structure of the sub-graph obtained when considering only cells sharing the same basic cell $c \in \mathrm{C}_{\mathtt{base}(M\to M')}$. This subgraph is, in general, a forest, and any two distinct trees in the forest have inconsistent classes at their root cells. The precedence edges internal to such forests are all induced by query enablings, while precedence edges between forests are induced by output enablings, and thus from enablings in base($M \to M'$). We may talk of a *query precedence forest* or *query precedence tree* for a basic cell $c$ in $a$.

Each of the algorithms given in the examples with a base DCDS **Bool** corresponds to a single tree. The identity algorithm on **Foo** in examples 2.3.6 and 2.4.1 corresponds to three forests, two with a single tree and one (for the basic cell c) containing two trees.

We now give the tree lemma, an important technical lemma which states formally the properties mentioned above. A corresponding lemma has been shown by Berry and Curien for sequential algorithms. It is the basis for a tree-like notation for algorithms and, in general, for reasoning about the structure of algorithms. As an added benefit, we will use the tree lemma to show that exponentiation preserves stability.

**Proposition 2.4.2** *Let $M$ be a DCDS, $p, p_1, p_2, q \in \mathcal{P}_t(\mathcal{F}(M))$. If $p \prec_q p_1$, $p \prec_q p_2$, and $p_1 \Uparrow p_2$ then $p_1 = p_2$.*

**Lemma 2.4.3 (Tree Lemma)** *For $M$ and $M'$ DCDSs, and an algorithm $a \in \mathcal{D}(M \to M')$,*

*(1) If $pc, p'c \in \mathrm{E}(a)$ and $p \Uparrow p'$ then:*

*(1a) Either $pc \ll^* p'c$, or $p'c \ll^* pc$.*

*If $pc \ll^* p'c$ then there exists $n \geq 0$ and a chain*

$$p = p_0 \prec_{q_1} p_1 \prec_{q_2} p_2 \cdots \prec_{q_n} p_n = p'$$

*such that $\forall i < n \;.\; (p_i c, \mathbf{query}\ q_{i+1}) \in a$.*

*(1b) And if $(pc, \mathbf{output}\ v), (p'c, \mathbf{output}\ v') \in a$ for some $v, v'$, then $p = p'$ (and $v = v'$).*

*(2) Any cell $pc$ enabled in $a$ has only one enabling in $a$.*

**Corollary 2.4.4** *For any DCDSs $M$ and $M'$, $M \to M'$ is a DCDS.*

**Corollary 2.4.5** *If $M \to M'$ is a DCDS, $a \in \mathcal{D}(M \to M')$, $p \Uparrow p'$ and $(pc, u), (p'c, u) \in a$ for some $u$, then $p = p'$.*

## 2.5 Currying

We now define currying and uncurrying operators. They turn out to be simple manipulations of states of the representation DCDS.

**Definition 2.5.1** Let $M_1$, $M_2$ and $M'$ be DCDSs. Let $M_u$ and $M_c$ be the DCDSs

$$\begin{aligned} M_u &= M_1 \times M_2 \to M' \\ M_c &= M_1 \to M_2 \to M' \end{aligned}$$

so that $\mathrm{rep}(M_u) = (M_1 \times M_2) \times \mathrm{rep}(M')$ and $\mathrm{rep}(M_c) = M_1 \times (M_2 \times \mathrm{rep}(M'))$. We define

$$\begin{aligned} \mathrm{curry} &: \mathcal{E}(\mathrm{rep}(M_u)) \to \mathcal{E}(\mathrm{rep}(M_c)) \\ \mathrm{uncurry} &: \mathcal{E}(\mathrm{rep}(M_c)) \to \mathcal{E}(\mathrm{rep}(M_u)) \end{aligned}$$

by

$$\begin{aligned} \forall \langle\langle y_1, y_2\rangle, \bar{y}'\rangle \in \mathcal{E}(\mathrm{rep}(M_u)) \;. \quad \mathrm{curry}(\langle\langle y_1, y_2\rangle, \bar{y}'\rangle) &= \langle y_1, \langle y_2, \bar{y}'\rangle\rangle \\ \forall \langle y_1, \langle y_2, \bar{y}'\rangle\rangle \in \mathcal{E}(\mathrm{rep}(M_c)) \;. \quad \mathrm{uncurry}(\langle y_1, \langle y_2, \bar{y}'\rangle\rangle) &= \langle\langle y_1, y_2\rangle, \bar{y}'\rangle \end{aligned}$$

Extend curry to algorithms as follows, and extend uncurry in a completely analogous way.

$$\begin{aligned} &\forall q \in \mathcal{P}_t(\mathcal{E}(\mathrm{rep}(M_u))) \;. \\ &\quad \mathrm{curry}(q) = \{\mathrm{curry}(\bar{y}) | \bar{y} \in q\} \\ &\forall a \in \mathcal{D}(M_u) \;. \\ &\quad \mathrm{curry}(a) = \{(\mathrm{curry}(p)c, \mathbf{query}\ \mathrm{curry}(q)) | (pc, \mathbf{query}\ q) \in a\} \\ &\qquad\qquad \cup \{(\mathrm{curry}(p)c, \mathbf{output}\ v) | (pc, \mathbf{output}\ v) \in a\} \end{aligned}$$

**Proposition 2.5.2** *The maps* curry *and* uncurry *are well defined, that is, they produce states of $M_c$, $M_u$, when applied to states of $M_u$, $M_c$, respectively. Moreover,* curry *is an isomorphism from $\mathcal{D}(M_u)$ to $\mathcal{D}(M_c)$, and* uncurry *is its inverse. The two maps preserve enablings.*

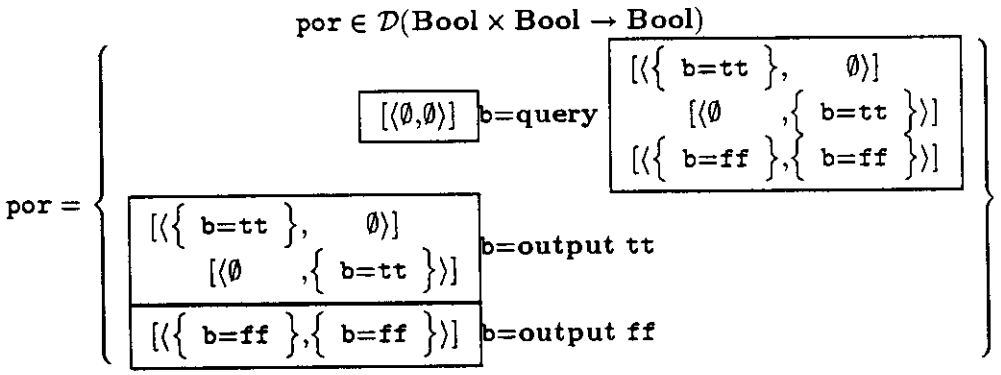



Figure 12: An alternative presentation of the parallel or algorithm

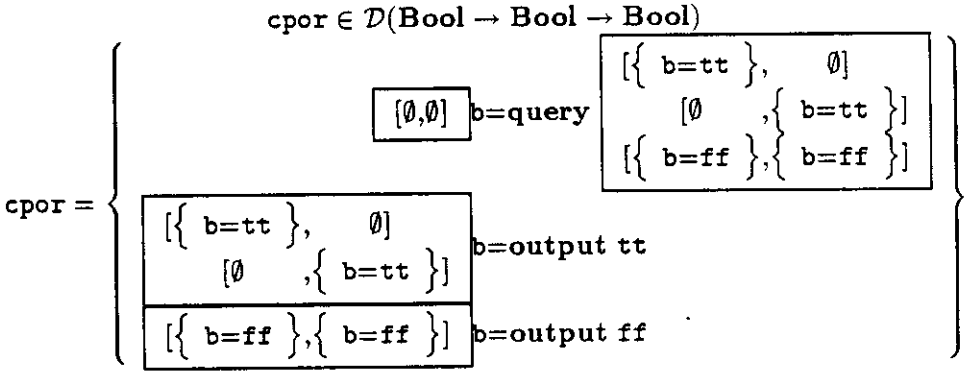



Figure 13: The curried parallel or algorithm, cpor = curry(por)

**Example 2.5.3** Figure 13 presents cpor = curry(por), the curried version of por. Contrast it with an alternative presentation of por in figure 12, using a compound state notation for states of **Bool** × **Bool**. •

**Example 2.5.4** Figure 14 presents $\mathbf{cgf}' = \mathrm{curry}(\mathrm{curry}(\mathbf{gf}'))$, the fully curried $\mathbf{gf}'$ algorithm. •

# 3 Application

Recall that for a sequential algorithm $a$ of $M \rightarrow_{seq} M'$ and a state $x$ of $M$, Berry and Curien defined the application of $a$ to $x$ by

$$\mathtt{cgf}' \in \mathcal{D}(\mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool})$$

$\mathtt{cgf}' =$

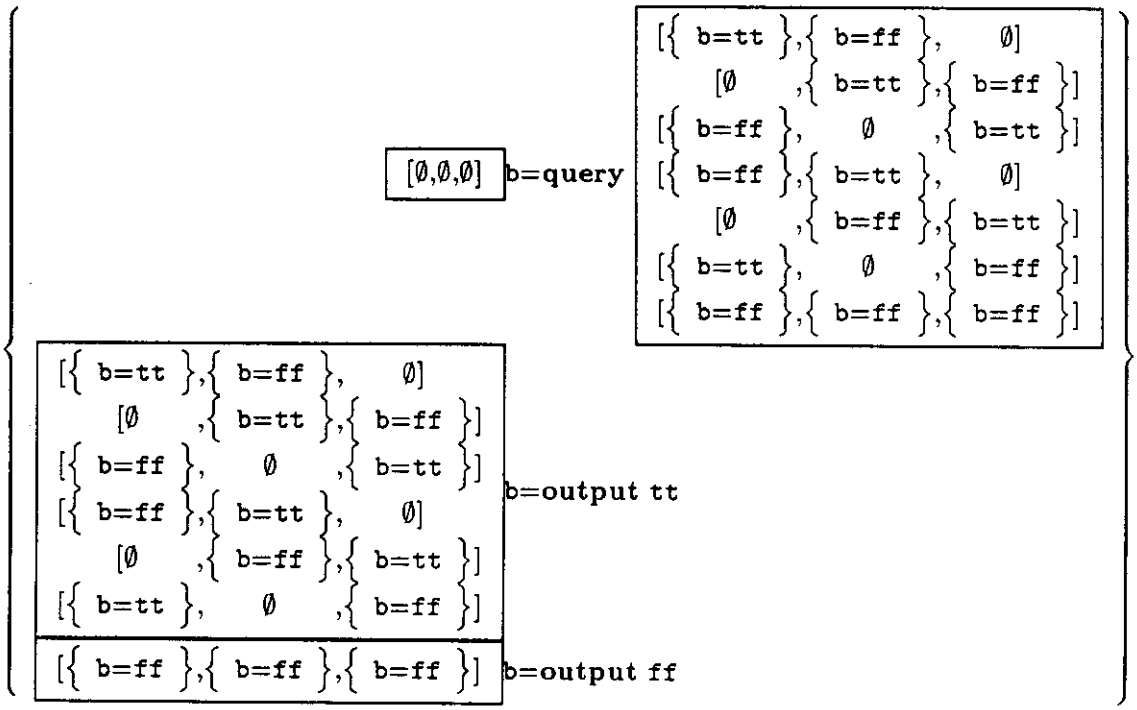



Figure 14: The curried algorithm for $gf'$, $\mathtt{cgf}' = \mathrm{curry}(\mathrm{curry}(\mathtt{gf}'))$

$$a \cdot_{seq} x = \{(c', v') | \exists y.\ (yc', \mathbf{output}\ v') \in a\ \&\ y \subseteq_{\mathcal{D}} x\}.$$

One might read this as saying that the events $(c', v')$ of $a \cdot_{seq} x$ are obtained by "projection" from events $(yc', \mathbf{output}\ v')$ of $a$ whose state component $y$ conforms with $x$ in that $y \subseteq_{\mathcal{D}} x$.

We will in fact start with a technical definition of a generalized (and formal) projection operation fitted to our parallel setting. The motivation for its definition follows, and we will then give the reasoning leading to our definition of *parallel application.*

In this section, let $M$ and $M'$ be DCDSs.

## 3.1 Projection

Let $M_\times$ abbreviate $\mathrm{rep}(M')$, so that $\mathrm{rep}(M \to M') = M \times M_\times$.

**Definition 3.1.1** For $x \in \mathcal{D}(M)$, define the *projection* by $x$

$$\pi_x : \mathcal{P}_t(\mathcal{E}(M \times M_\times)) \to \mathcal{P}_t(\mathcal{E}(M_\times)) \cup \{\emptyset\}$$

by

$$\forall q \in \mathcal{P}_t(\mathcal{E}(M \times M_\times))\ .\ \pi_x(q) = \mathrm{trim}(\{\mathrm{snd}(\bar{z}) | \bar{z} \in q\ \&\ \mathrm{fst}(\bar{z}) \subseteq_{\mathcal{D}} x\}).$$

Extend $\pi_x$ to $V_{M \to M'}$ by setting:

$$\begin{array}{rcl} \pi_x(\mathbf{query}\ q) & = & \mathbf{query}\ \pi_x(q), \\ \pi_x(\mathbf{output}\ v) & = & \mathbf{output}\ v. \end{array}$$

Projection is continuous:

**Proposition 3.1.2** *For any $x \in \mathcal{D}(M)$ and $Q \subseteq \mathcal{P}_t(\mathcal{E}_{fin}(M \times M_\times))$, if $\Uparrow Q$ then*

$$\sqcup\{\pi_x(q) | q \in Q\} = \pi_x(\sqcup Q)$$

*(where the left hand side is to be taken as the empty set in case it is not well defined, that is, when $\pi_x(q) = \emptyset$ for some $q \in Q$.)*

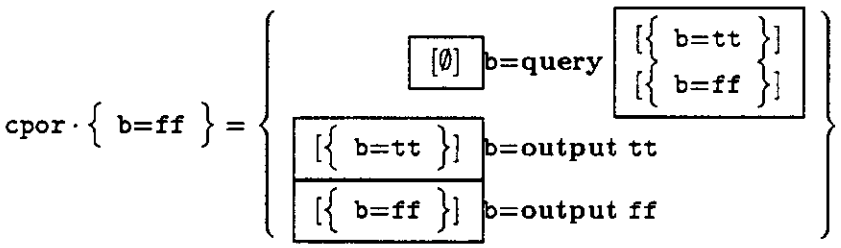
The above proposition is also true for trim powerdomains over functional sets of events, *i.e.* for $Q \subseteq \mathcal{P}_t(\mathcal{F}_{fin}(M \times M_\times))$, and for trim powerdomains over states, *i.e.* for $Q \subseteq \mathcal{P}_t(\mathcal{D}_{fin}(M \times M_\times))$.

## 3.2 Motivation

We discuss now the considerations leading to our formal definition of application. We intersperse the discussion with several examples, and only afterwards do we give the formal definition. All of the examples given conform with the ensuing definition.

Let us now consider the application of an algorithm $a \in \mathcal{D}(M \to M')$ to $x \in \mathcal{D}(M)$. If $M'$ is an exponentiation itself, then the result should be an algorithm $a' \in \mathcal{D}(M')$. Intuitively speaking, there ought to be an operational correspondence between the events of $a$ and the events of $a'$, in the rough sense that for each event $(pc, u) \in a$ there are some events of $a'$ which are responsible for $a'$ exhibiting the same behavior that $(pc, u)$ entails when the first argument to $a$ is known to be $x$. Let us, for the time being, assume that each event of $a$ has at most one corresponding event of $a'$ — call this the *uniqueness* assumption.

**Example 3.2.1** Consider the application of the curried parallel-or algorithm `cpor` to $\{\mathtt{b} = \mathtt{ff}\}$. Intuitively the result should be the identity algorithm on **Bool**; this is in fact the case. There is a clear one-to-one correspondence between the events of the algorithms, and the uniqueness assumption holds.




•

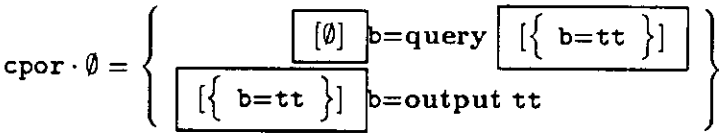
Consider an output event $(pc, \mathbf{output}\ v) \in a$. When $a$ is applied to $x$, a particular $\bar{x} \in p$ may be *the* true description of $a$'s input only if $\mathrm{fst}(\bar{x}) \subseteq_{\mathcal{D}} x$. For such an $\bar{x}$ there must then be a corresponding output event $(p'c, \mathbf{output}\ v) \in a'$ with $\mathrm{snd}(\bar{x}) \in p'$. By the uniqueness assumption, this implies that $p' = \pi_x(p)$ — and this is, in fact, the motivation behind the definition of projection.

If no $\bar{x} \in p$ has $\mathrm{fst}(\bar{x}) \subseteq_{\mathcal{D}} x$, then the output event $(pc, \mathbf{output}\ v)$ will not take place at all when $a$ is applied to $x$. Therefore there need not be a corresponding event in $a'$. We will identify this case by $p' = \pi_x(p) = \emptyset$, which is not a valid class anyway.

For a query event $(pc, \mathbf{query}\ q) \in a$ there should be a corresponding query event $(p'c, \mathbf{query}\ q')$ in $a'$. As for output events, using the uniqueness assumption, we should have $p' = \pi_x(p)$. Similarly, $q'$ need only contain $\mathrm{snd}(\bar{y})$ for the branches $\bar{y} \in q$ for which $\mathrm{fst}(\bar{y}) \subseteq_{\mathcal{F}} x$: no other branch of $q$ will ever be satisfied by $x$. We have then $q' = \pi_x(q)$ — but this is independent of the uniqueness assumption, since we want $q'$ to be identical to the query that is externally visible when $a$ is applied to $x$ and $q$ is issued. (Admittedly, this argument depends heavily on the underlying operational semantics, which we do not present here.)

As with output events, if $p' = \pi_x(p) = \emptyset$ then no element of $p$ is a true description of the input state, when $a$ is applied to $x$, and there need be no corresponding event in $a'$. Quite similarly, if $\mathrm{fst}(\bar{y}) \not\subseteq_{\mathcal{F}} x$ for all branches $\bar{y}$ of $q$, then no branch of the query can be satisfied when $a$ is applied to $x$, and there need not be a corresponding event in $a'$. This case is identified by $q' = \pi_x(q) = \emptyset$, which, again, is not a valid query anyway.

**Example 3.2.2** Consider the application of `cpor` to $\emptyset$.




The resulting algorithm will not have or need an event with an **output** `ff` command, and, in fact, projection of that event of `cpor` produces an invalid empty class. •

We have thus far seen two characteristic situations in which a query event $(pc, \mathbf{query}\ q)$ need not have a corresponding event in $a'$: either because it will not be executed when $a$ is applied to $x$, in which case $\pi_x(p) = \emptyset$; or because, even if it is executed, no branch of its query may possibly be taken. In both cases, no event of $a$ following from $(pc, \mathbf{query}\ q)$ can be executed either, when applying $a$ to $x$.

A third type of a query event in $a$ that need not have a corresponding query event in $a'$ arises when some branch $\bar{y} \in q$ is completely satisfied by $x$, that is, when $\mathrm{fst}(\bar{y}) \subseteq_{\mathcal{F}} x$ and $\mathrm{snd}(\bar{y}) = \emptyset$. It is then the case that $\emptyset \in q' = \pi_x(q)$, and again $q'$ is not a valid query.

In this third case, in contrast to the previous two, some event following $(pc, \mathbf{query}\ q)$ may in fact have a corresponding event in $a'$; but $(pc, \mathbf{query}\ q)$ itself has no corresponding event in $a'$ because $a$ when applied to $x$ may "jump to conclusions" without waiting for any additional arguments. That is, it will act according to the branch that is completely satisfied by $x$, and there is no need to issue any corresponding query; rather, $a'$ will have an event corresponding to some subsequently enabled event that is consistent with this branch. We refer to this phenomenon, involving the loss or "abstracting away" of events, as *abstraction.*

**Example 3.2.3** This third situation occurs when we apply cpor to {b = tt}, obtaining a non-strict constant algorithm, since a branch of the query is completely satisfied by {b = tt}.

$$\mathtt{cpor} \cdot \left\{ \mathtt{b{=}tt} \right\} = \left\{ \boxed{[\emptyset]}\ \mathtt{b{=}output\ tt} \right\}$$

•

So far we have outlined application in terms of our projection operator. Indeed, $\pi_x(a) = \{(\pi_x(p)c, \pi_x(u)) | (pc, u) \in a\}$ is a fairly natural generalization from the Berry-Curien formulation of $a \cdot_{seq} x$. It does yield a very useful approximation of application — in fact, for all examples mentioned so far in this section it yields precisely what we would intend the application to produce. It is probably rather more intuitive than the precise definition we are about to give. But it cannot serve to define application completely since, in general, $\pi_x(a)$ is not a state. The reason is that *splitting* of equivalence classes may occur: equivalent but inconsistent branches in a query $q$ need not remain equivalent when $q$ is projected by $x$. The equivalence classes of $q$ will each be mapped into one or more equivalence classes of $\pi_x(q)$, so that if $(pc, \mathbf{query}\ q) \vdash_{M \to M'} p_1 c$, then $(\pi_x(p)c, \mathbf{query}\ \pi_x(q)) \vdash_{M'} p'c$ with $p' \subseteq \pi_x(p_1)$; and $p' = \pi_x(p_1)$ is true only if the relevant equivalence class is not split.

**Example 3.2.4** For an example of splitting consider the application of the cgf′ algorithm to $\emptyset$.

$$\mathtt{cgf'} \cdot \emptyset = \left\{ \begin{array}{l} \boxed{[\emptyset,\emptyset]}\ \mathtt{b{=}query}\ \boxed{\begin{array}{l}[\{\mathtt{b{=}tt}\},\{\mathtt{b{=}ff}\}] \\ [\{\mathtt{b{=}ff}\},\{\mathtt{b{=}tt}\}]\end{array}} \\ \boxed{[\{\mathtt{b{=}tt}\},\{\mathtt{b{=}ff}\}]}\ \mathtt{b{=}output\ tt} \\ \boxed{[\{\mathtt{b{=}ff}\},\{\mathtt{b{=}tt}\}]}\ \mathtt{b{=}output\ tt} \end{array} \right\}$$

Splitting occurs because the two branches which are projected, though not consistent, are equivalent in the original query by means of branches which are not projected. There is no longer, after projection, an equivalence chain *in the query* relating these two branches.

Contrast this with $\pi_\emptyset(\mathtt{cgf'})$ which is not safe:

$$\pi_\emptyset(\mathtt{cgf'}) = \left\{ \begin{array}{l} \boxed{[\emptyset,\emptyset]}\ \mathtt{b{=}query}\ \boxed{\begin{array}{l}[\{\mathtt{b{=}tt}\},\{\mathtt{b{=}ff}\}] \\ [\{\mathtt{b{=}ff}\},\{\mathtt{b{=}tt}\}]\end{array}} \\ \boxed{\begin{array}{l}[\{\mathtt{b{=}tt}\},\{\mathtt{b{=}ff}\}] \\ [\{\mathtt{b{=}ff}\},\{\mathtt{b{=}tt}\}]\end{array}}\ \mathtt{b{=}output\ tt} \end{array} \right\}$$

In general, as in this example, $\pi_x(a)$ may violate the safety requirement in that its classes are actually unions of enabled classes.

In order to handle splitting correctly we now have to abandon the uniqueness assumption. The definition of application can no longer be local, but must rather be an inductive definition, proceeding by the above intuitions, with safety built into the definition, so that we reconstruct the correct classes.

### 3.3 Definition of Application

Here, now, is the inductive definition of application[7]. We build $a \cdot x$ by induction on its enabling layers[8]; We define $(a \cdot x)_{n+1}$ so that it will turn out to be the maximal sub-state of $a \cdot x$ such that $\mathrm{F}((a \cdot x)_{n+1}) \subseteq \mathrm{E}((a \cdot x)_n)$. The sequence thus obtained is increasing, and $a \cdot x$ is its limit.

**Definition 3.3.1** For $a \in \mathcal{D}(M \to M')$, $x \in \mathcal{D}(M)$, define the *application* of $a$ to $x$, denoted by $a \cdot x$, to be $\cup_{n \geq 0}(a \cdot x)_n$, where the sequence $\{(a \cdot x)_n\}_{n \geq 0}$ is inductively defined as follows:

- $(a \cdot x)_0 = \emptyset$.
- $(a \cdot x)_{n+1} = \{(p'c, \pi_x(u)) \in \mathrm{E}_{M'} | (pc, u) \in a \ \&\ p' \subseteq \pi_x(p) \ \&\ p'c \in \mathrm{E}(a \cdot x)_n\}$.

We define a map from events of $a \cdot x$ to events of $a$ that will make more precise the notion of corresponding events alluded to above. First, we need the following result:

**Proposition 3.3.2** *If $(p'c, u') \in a \cdot x$ then there exists a unique $(pc, u) \in a$ such that $p' \subseteq \pi_x(p)$ and $u' = \pi_x(u)$.*

**Definition 3.3.3** For $a \in \mathcal{D}(M \to M')$ and $x \in \mathcal{D}(M)$, define

$$\mathrm{source}_{a,x} : a \cdot x \to a$$

by setting, for each $(p'c, u') \in a \cdot x$, $\mathrm{source}_{a,x}(p'c, u')$ to be the unique event $(pc, u) \in a$ such that $u' = \pi_x(u)$ & $p' \subseteq \pi_x(p)$.

Note especially the requirement in the definition of application that events of $a \cdot x$ belong to $\mathrm{E}_{M'}$. It is because of this that $\mathrm{source}_{a,x}$ is not surjective. This requirement filters out the undesired by-products of $\pi_x$, identified above as arising from cases where one of the following three situations occur:

- $\pi_x(q) = \emptyset$ for some $(pc, \mathbf{query}\ q) \in a$.

[7]The above discussion assumes that $M'$ is an exponentiation, and $a \cdot x$ is an algorithm. If $M'$ is not an exponentiation, then, proceeding by the above guidelines, one would obtain a trivial algorithm $a'$ of a "unary exponentiation" $\to M'$ that is constructed as an exponentiation out of a representation of Null and a base $M'$. Note that such an algorithm $a'$ will have no query events, since the only possible branch is the empty branch; therefore it is isomorphic to a state of $M'$ by the obvious isomorphism, which maps the event $(\{\emptyset\}c, \mathbf{output}\ v)$ to $(c, v)$. We will omit explicit mention of this isomorphism in the definition and related development for simplicity of presentation.

[8]That is, the induction may be thought of as formulated on the height of the proof for a cell in $a \cdot x$.

- $\pi_x(p) = \emptyset$ for some $(pc, u) \in a$.
- $\emptyset \in \pi_x(q)$ for some $(pc, \mathbf{query}\ q) \in a$.

Note also that $\text{source}_{a,x}$ is not injective, because of the possibility of splitting, as discussed above.

**Proposition 3.3.4** *Application is well defined, that is, for $a \in \mathcal{D}(M \to M')$ and $x \in \mathcal{D}(M)$, $a \cdot x \in \mathcal{D}(M')$.*

Examples of application may be found in the preceding discussion.

Application is monotone and continuous in its first argument, but in general it is not even monotone in its second argument. For an example contrast $\texttt{cpor} \cdot \emptyset$ with $\texttt{cpor} \cdot \{\texttt{b} = \texttt{tt}\}$, where abstraction occurs, or even with $\texttt{cpor} \cdot \{\texttt{b} = \texttt{ff}\}$, where the query gains an additional branch — we will call this *amplification.*

We are currently investigating an "intensional strictness" order on parallel algorithms that treats abstraction and amplification more appropriately, so that application enjoys monotonicity and continuity [BG]. For the time being we show that ground application — maximally iterated application — has these properties even when set inclusion is used.

## 3.4 Ground application

Ground application is just iterated or repeated application of an algorithm until the result is a basic state. Since the result is not an algorithm, we do not have to deal with abstraction and amplification, and may order states by set inclusion without violating monotonicity or continuity.

**Definition 3.4.1** For $a \in \mathcal{D}(M)$ and $\bar{x} \in \mathcal{D}(\text{rep}(M))$, define inductively the *ground application* $a \cdot_g \bar{x}$ of $a$ to $\bar{x}$ to be:

$$a \cdot_g \bar{x} = \begin{cases} a & \text{if } \text{rep}(M) = \mathbf{Null} \\ (a \cdot \text{fst}(\bar{x})) \cdot_g \text{snd}(\bar{x}) & \text{otherwise} \end{cases}$$

Or, equivalently:

$$a \cdot_g \bar{x} = (\ldots(a \cdot x_d) \cdot \ldots) \cdot x_1$$

when $\bar{x} = [x_d, \ldots x_1]$. •

An equivalent direct characterization, bringing out clearly the analogy between ground application and the Berry-Curien definition of the *input-output function* of an algorithm $a$, is:

**Proposition 3.4.2** *For $a \in \mathcal{D}(M)$ and $\bar{x} \in \mathcal{D}(\text{rep}(M))$,*

$$a \cdot_g \bar{x} = \{(c, v) | (pc, \mathbf{output}\ v) \in a\ \&\ p \sqsubseteq_{\mathcal{D}} \{\bar{x}\}\}$$

The link with the sequential definition is made even clearer when we observe that $p \sqsubseteq_{\mathcal{D}} \{\bar{x}\}$ holds if and only if there is a $\bar{y} \in p$ such that $\bar{y} \subseteq \bar{x}$.

**Definition 3.4.3** For $a \in \mathcal{D}(M)$ define the *ground input-output function* associated with $a$, denoted $|a|_g$, to be

$$|a|_g : \mathcal{D}(\text{rep}(M)) \to \mathcal{D}(\text{base}(M))$$

$$\forall \bar{x} \in \mathcal{D}(\text{rep}(M))\ .\ |a|_g(\bar{x}) = a \cdot_g \bar{x}$$

•

**Proposition 3.4.4** *Ground application is monotone and continuous in its second argument, that is, for $a \in \mathcal{D}(M)$, $|a|_g$ is monotone and continuous.*

### 3.5 Application for filiform DCDSs

For filiform DCDSs the general definition of application can be simplified. A CDS is *filiform* iff all its enablings contain at most one event. The class of filiform DCDSs is interesting, since common atomic DCDSs like **Null**, **Bool** and **Nat** are filiform, and product and exponentiation preserve filiformness. In fact, Berry and Curien [Cur86] showed that the category of filiform DCDSs with sequential algorithms is also cartesian-closed.

In an algorithm of a filiform DCDS, a class $p$ of an enabled cell is a single equivalence class; $p/\approx = \{p\}$. Although splitting may still occur, and thus in general $\pi_x(a)$ is not safe, we can exploit this property to give a simpler, local and non-inductive, definition of application, which closely resembles $\pi_x(a)$.

**Proposition 3.5.1** *If $M \to M'$ is a filiform DCDS, $a \in \mathcal{D}(M \to M')$, and $x \in \mathcal{D}(M)$ then*

$$a \cdot x = \{(p'c, \pi_x(u)) \in \mathrm{E}_{M'} | (pc, u) \in a \ \&\ p' \in \pi_x(p)/\approx\}$$

## 4 Conclusion

We have defined a new mathematical model of parallel algorithms and a new parallel exponentiation for deterministic concrete data structures. We have explained our construction intuitively as a natural generalization of the sequential algorithms of Berry and Curien. We have discussed both informally and formally a variety of examples, and we have stated several important properties of our construction. In particular, we defined currying and uncurrying operators with the expected properties, and we formulated precisely what it means to apply a parallel algorithm to an input state from the relevant DCDS.

This presentation has been extensively motivated by appeal to an informally described operational semantics. We included no proofs for our results. We will describe a formal operational semantics, and give the relevant proofs, in a subsequent presentations of this work. In any case, we feel that we have included enough detail to enable the reader to grasp our main ideas and to see that what we have devised is a reasonable attempt to incorporate an appealing treatment of concurrency into the framework of DCDS.

We plan to explore the mathematical structure of our model more deeply and we will attempt to define an appropriate notion of *composition* for parallel algorithms; this will enable us to establish one of our primary aims (and a touchstone for judging the "validity" of our model): that we have indeed built a cartesian closed category in which a satisfactory semantic account of parallel algorithms can be given. However, as we hinted above, it seems that a new order properly taking account of the phenomena of abstraction and amplification is required, and in [BG] we introduce a new "intesional strictness" order that solves the problem for first-order exponentiation. We will establish the appropriate relationship between our model and the standard continuous functions model of PCF. Just as the Berry-Curien sequential algorithms correspond to *sequential* functions paired with a computation strategy, we should be able to show how our parallel algorithms can be viewed as *continuous* functions paired with a strategy. This relationship seems very appropriate, since the standard continuous functions model is fully abstract for PCFP, the extension of PCF to include a parallel conditional. Again, this type of relationship would be further evidence in favor of the naturalness of our model.

We also intend to design a parallel programming language based closely on our model, just as the programming language CDS0 was built on top of the Berry–Curien model. Our overall aim is to achieve semantic properties analogous to those listed for CDS0.

Several interesting possibilities are suggested by our work. First, it seems that we are able to define a generalized version of sequentiality indices, perhaps better called *computation indices* which are applicable to the parallel setting; these are just the queries of the parallel algorithms. The sequential algorithms of Berry and Curien turn out (unsurprisingly) to correspond to algorithms with a minimal degree of parallelism (that have only one cell filled in each query). We can also characterize the class of algorithms which have a stable input-output function, in Berry's sense. We can formulate some intuitively natural new orderings on algorithms: one which reflects the *degree of parallelism* exhibited by an algorithm, and possibly one which measures the *laziness* of an algorithm. There appears to be a natural hierarchy among parallel algorithms, based on our notion of degree of parallelism. We would like to investigate the structure of this hierarchy, and perhaps it might be useful in assessing the relative expressive powers of various parallel primitives. We hope also to achieve a semantics for a parallel language in which the denotations reflect accurately the *efficiency* with which an algorithm is able to compute a function.

## 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.

[Ber78] G. Berry. Stable models of typed $\lambda$-calculi. In *Proc. 5th Coll. on Automata, Languages and Programming. LNCS 62*, pages 72–89, Springer Verlag, Berlin, New-York, July 1978.

[BG] S. Brookes and S. Geva. Towards a theory of parallel algorithms on concrete data structures. Submitted to the Fifth Annual IEEE Symposium on Logic in Computer Science, Philadelphia, June 1990.

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

[Hue86] G. Huet. Formal structures for computation and deduction. Class notes for graduate course at CMU, May 1986. First edition.

[KM77] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 1977*, pages 993–998, North Holland, 1977.

[KP78] Gilles Kahn and Gordon Plotkin. *Domaines Concrets.* Rapport 336, IRIA-LABORIA, 1978.

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

[Smy78] M. B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, February 1978.