Towards a Theory of Parallel Algorithms on Concrete Data Structures

Stephen Brookes Shai Geva

July 1991

CMU-CS-91-157 2

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in *Theoretical Computer Science*
Supersedes Technical Report CMU-CS-90-170

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597 and in part by NSF/DARPA Grant CCR-8906483.

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

Abstract

Building on Kahn and Plotkin's theory of concrete data structures and sequential functions, Berry and Curien defined an intensional model of sequential algorithms between concrete data structures. In this paper we report on an attempt to develop a similar intensional model of concurrent computation. We present a notion of parallel algorithm between concrete data structures, together with suitable application and currying operations. We define an *intensional strictness* ordering on parallel algorithms, with respect to which application is well behaved (at first order types). We define the *input-output function* computed by a parallel algorithm, and we show that every parallel algorithm computes a continuous function. Thus, a parallel algorithm may be viewed as a continuous function together with a parallel computation strategy. In contrast, a Berry-Curien sequential algorithm may be viewed as a sequential function together with a sequential computation strategy. The intensional strictness ordering on parallel algorithms corresponds to the pointwise ordering on the functions they compute, in the same sense that the set inclusion ordering used by Berry and Curien on sequential algorithms corresponds to the stable ordering on the functions they compute. We believe that the ideas and results presented here constitute a first step towards a fuller understanding of the intensional semantics of parallelism, even though the model presented here is not yet general enough to provide a satisfactory account of higher order algorithms, and lacks a notion of composition for algorithms. We present some ideas for overcoming these deficiencies, and some directions for further research.

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 its standard continuous functions semantic model is not fully abstract because 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].

Milner [Mil77], Sazonov [Saz75], and Vuillemin [Vui73] proposed notions of sequential functions; however, their constructions make essential use of the number of arguments to a function but do not adequately reflect the internal structure of these arguments, so that their notions of sequentiality are not general enough. Kahn and Plotkin [KP78] defined *concrete data structures*, or CDSs, together with their order-theoretic counterparts, *concrete domains*, which made possible a more general definition of *sequentiality* of functions. Berry [Ber78] introduced the notion of *stability*, a property of functions intermediate between sequentiality and continuity. 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. These negative results paved the way for the development of an intensional model, since no suitable extensional models were found.

Berry and Curien were able to define an exponentiation for concrete data structures, by replacing functions by a notion of *sequential algorithms*. The resulting category of *deterministic* concrete data structures (DCDSs) and sequential algorithms is cartesian closed. Furthermore, a notation for elements of DCDSs is a basis for a functional language CDS0 [BC85], for which the sequential algorithms model provides a semantics with several interesting properties: The semantics is fully abstract with respect to a notion of observability that is sensitive to computation strategy; the model is intensional rather than extensional; sequential algorithms, ordered by set inclusion, form a concrete domain; a sequential algorithm may be viewed as a sequential input-output function paired with a computation strategy. The operational semantics is based on an extension of Kahn-MacQueen's coroutine mechanism [KM77], employing lazy evaluation.

Although it does not solve the original full abstraction problem for PCF, the Berry–Curien model of sequential algorithms is interesting in its own right. It provides deep insights into the nature of deterministic sequential computation. We propose here a generalization of Berry and Curien's notion of algorithm that incorporates *deterministic concurrency* into the framework. We believe that there are fundamental insights into the semantic treatment of parallelism to be gained by doing this. Like Berry and Curien, we restrict attention to deterministic computation[1], although we do allow non-determinism in the scheduling of parallel computations.

In section 2, based on [Cur86], we summarize the background material on DCDSs, sequential algorithms, and stable and sequential functions.

In section 3 we present our notion of parallel algorithm between deterministic concrete data structures. We explain how our construction arises out of an attempt to generalize the Berry-Curien concepts. The key idea is to replace the "valof" command of a sequential exponentiation

[1]Berry and Curien also discussed briefly an attempt to introduce non-determinism into their model [Cur86, section 2.7], but they were unable to obtain a cartesian closed category for non-deterministic sequential computation.

with a "query" command that spawns parallel sub-computations; the formal treatment of this and its consequences leads naturally to the use of a powerdomain. We present a variety of example algorithms, and we define currying and uncurrying operations for parallel algorithms.

In section 4 we formalize what it means to *execute* a parallel algorithm by defining a suitable *application* operation. We show that our notion of parallel application is intuitively right by discussing the applicative behavior of several example algorithms. We explain how our notion of application generalizes the sequential application of Berry and Curien. We define the input-output function computed by a parallel algorithm.

Application for parallel algorithms, unlike its sequential counterpart, is not continuous with respect to set inclusion. This is not a defect of our model or of our definition of application, but rather shows that set inclusion is not an appropriate ordering on parallel algorithms. In section 5 we identify the causes of this failing and introduce a more appropriate ordering, which we call the *intensional strictness* order. Informally, an algorithm a' is above another algorithm a in this order if a' needs less information, at an earlier stage of the computation, to achieve at least the same output as a. We regard intensional strictness as a natural generalization to the intensional setting of the standard extensional ordering on continuous functions. In contrast, the set inclusion ordering on algorithms used by Berry and Curien corresponds to the stable ordering [Ber78] on sequential functions. We show that, at first order types, with suitable countability assumptions, the intensional strictness order is a directed-complete ω-algebraic pre-order on parallel algorithms. We show that application and currying are continuous with respect to the new ordering. This implies that the input-output function computed by an algorithm is continuous, suggesting that parallel algorithms can be viewed as continuous functions paired with parallel computation strategies, by analogy with the result of Berry and Curien that their sequential algorithms correspond to sequential functions paired with sequential computation strategies.

In section 6 we point out some limitations of our model and outline how we intend to overcome them in future work. We discuss a number of topics for further investigation.

2 Background

2.1 Concrete Data Structures

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*, and an *enabling* relation $\vdash$ between finite sets of events and cells. Events are denoted either (c, v) or $c = v$.

For a CDS $M = (\mathrm{C}_M, \mathrm{V}_M, \mathrm{E}_M, \vdash_M)$, $x, y \subseteq \mathrm{E}_M$, and $c \in \mathrm{C}_M$, if $y \vdash_M c$ we say that y is an *enabling* of c. If $y \vdash_M c$ and $y \subseteq x$ we say that y is an enabling of c in x and write $y \vdash_x c$. If $\emptyset \vdash_M c$ we say that c is *initial*.

We define $\mathrm{F}(y)$, the cells *filled in* y, to be the collection of cells in the events of y. $\mathrm{E}(y)$, the cells *enabled in* y, is the collection of cells that have an enabling in y. $\mathrm{A}(y)$, the cells *accessible in* y, is the collection of cells which are enabled in y but not filled; that is, $\mathrm{A}(y) = \mathrm{E}(y) \setminus \mathrm{F}(y)$.

For $c, c' \in \mathrm{C}_M$, we say that c *immediately precedes* c', denoted $c \ll_M c'$, iff there is an enabling $y \vdash_M c'$ such that $c \in \mathrm{F}(y)$. If, moreover, $y \subseteq x$ we say that c immediately precedes c' in x, denoted $c \ll_x c'$. Taking the reflexive and transitive closure of $\ll_M$, we say that c *precedes* c' iff $c \ll_M^* c'$, and analogously $\ll_x^*$ defines precedence in x. M is *well founded* iff $\ll_M$ is well founded.

For a well founded CDS M, we say that $y \subseteq \mathrm{E}_M$ is *functional*[2] iff any cell is filled in y with at

[2]Berry and Curien use the term *consistent* instead of functional.

most one value; let $\mathcal{F}(M)$ be the collection of functional sets of events. If $\mathrm{F}(y) \subseteq \mathrm{E}(y)$ we say that y is *safe*, and y is a *state* of M iff it is functional and safe. Let $\mathcal{D}(M)$ be the collection of states of M. We add a subscript to indicate finiteness, *e.g.*, $\mathcal{D}_{\mathrm{fin}}(M)$ for the collection of finite states. $(\mathcal{D}(M), \subseteq)$ is a concrete domain[3].

A well founded CDS is *stable* iff for any state x and cell c enabled in x, c has a unique enabling in x. A CDS is a *deterministic* CDS, or DCDS for short, iff it is well founded and stable. We will work from now on exclusively with DCDSs, although some of the development could be carried out more generally. Throughout the paper, M, M', M_1 and so on range over DCDSs.

Example 2.1 The DCDS **Null** has no cells, values, events, or enablings; its only state is the empty state $\emptyset$.

The DCDS **Bool** has a single initial cell $\mathtt{b}$, which may be filled with either of the values $\mathtt{tt}$ or $\mathtt{ff}$, representing the boolean truth values; its states are $\emptyset$, $\{\mathtt{b} = \mathtt{tt}\}$ and $\{\mathtt{b} = \mathtt{ff}\}$, and thus $(\mathcal{D}(\mathbf{Bool}), \subseteq)$ is isomorphic to the conventional flat boolean cpo.

The DCDS **Nat** has a single initial cell $\mathtt{n}$, which may be filled with a natural number; its states are $\emptyset$ and $\{\mathtt{n} = k\}$ for $k \in \mathbb{N}$, so that $(\mathcal{D}(\mathbf{Nat}), \subseteq)$ is isomorphic to the conventional flat natural numbers cpo.

The DCDS **LNat** has cells $\{\mathtt{b}_n \mid n \geq 0\}$, values 0 and 1, and enabling relation given by the rules $\emptyset \vdash_{\mathbf{LNat}} \mathtt{b}_0$ and $\{\mathtt{b}_i = 1\} \vdash_{\mathbf{LNat}} \mathtt{b}_{i+1}$, for $i \geq 0$. Thus, the cells are accessed in increasing order of index. We denote the states as follows: $S^n(\bot) = \{\mathtt{b}_i = 1 \mid i < n\}$ and $S^n(0) = \{\mathtt{b}_i = 1 \mid i < n\} \cup \{\mathtt{b}_n = 0\}$, for $n \geq 0$; and $S^\omega(\bot) = \{\mathtt{b}_i = 1 \mid i \geq 0\}$. Thus $(\mathcal{D}(\mathbf{LNat}), \subseteq)$ is isomorphic to the *lazy natural numbers* cpo, described for example in [Col89]. ●

2.2 Product of DCDSs

If c is a cell and i is a tag or label, we write $c.i$ for the the labelled cell (c, i). This notation extends to sets of cells and sets of events: for $C \subseteq \mathrm{C}_M$ and $y \subseteq \mathrm{E}_M$, $C.i = \{c.i \mid c \in C\}$ and $y.i = \{(c.i, v) \mid (c, v) \in y\}$. In defining products we use the labels 1 and 2.

The product of M_1 and M_2, $M_1 \times M_2$, is the DCDS obtained by taking a "disjoint union" of M_1 and M_2, in that cells are labelled by 1 or 2 to indicate where a cell, event or enabling originated; $\mathrm{C}_{M_1 \times M_2} = \mathrm{C}_{M_1}.1 \cup \mathrm{C}_{M_2}.2$, $\mathrm{V}_{M_1 \times M_2} = \mathrm{V}_{M_1} \cup \mathrm{V}_{M_2}$, $\mathrm{E}_{M_1 \times M_2} = \mathrm{E}_{M_1}.1 \cup \mathrm{E}_{M_2}.2$, and for $i = 1, 2$, $y.i \vdash_{M_1 \times M_2} c.i$ iff $y \vdash_{M_i} c$.

Pairs of sets of events are obtained similarly: $\langle z_1, z_2 \rangle = z_1.1 \cup z_2.2$. Projections are easily defined to satisfy $\mathrm{fst}(\langle z_1, z_2 \rangle) = z_1$ and $\mathrm{snd}(\langle z_1, z_2 \rangle) = z_2$. We use $\bar{x}$, $\bar{y}$, etc. to denote pairs.

The product trivially preserves well foundedness and stability, and pairing and the projections preserve functionality, safety and finiteness. $\mathcal{F}(M_1 \times M_2) = \{\langle z_1, z_2 \rangle \mid z_1 \in \mathcal{F}(M_1), z_2 \in \mathcal{F}(M_2)\}$, and set inclusion on $\mathcal{F}(M_1 \times M_2)$ coincides with componentwise set inclusion.

Example 2.2 **Bool** $\times$ **Bool** has two initial cells, $\mathtt{b.1}$ and $\mathtt{b.2}$, each of which may be filled with a value of $\mathtt{tt}$ or $\mathtt{ff}$. It has 9 states, one of which is $\{\mathtt{b.1} = \mathtt{tt}, \mathtt{b.2} = \mathtt{ff}\}$, alternatively denoted by $\langle\{\mathtt{b} = \mathtt{tt}\}, \{\mathtt{b} = \mathtt{ff}\}\rangle$. ●

2.3 Stable and Sequential Functions

We now define stability and sequentiality of functions from $\mathcal{D}(M)$ to $\mathcal{D}(M')$. The definition of sequentiality 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].

[3]When suitable countability requirements are imposed. See [KP78] and [Cur86, section 2.2] for details.

A continuous function $f : \mathcal{D}(M) \to \mathcal{D}(M')$ is *stable* if for any $x \in \mathcal{D}(M)$ and $x' \in \mathcal{D}(M')$ below $f(x)$ there exists a least state $M(f, x, x') \in \mathcal{D}(M)$ below x on which f attains or surpasses x', *i.e.*, for any $z \subseteq x$, $x' \subseteq f(z)$ iff $M(f, x, x') \subseteq z$.

A continuous function $f : \mathcal{D}(M) \to \mathcal{D}(M')$ is *sequential at* $x \in \mathcal{D}(M)$ 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–

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

In case (2), a cell $c \in \mathrm{A}(x)$ as described there is called a *sequentiality index* of f at x for c'.

$f : \mathcal{D}(M) \to \mathcal{D}(M')$ is *sequential* if it is continuous and it is sequential at every $x \in \mathcal{D}(M)$.

A sequential function is stable. The converse, however, does not hold.

Example 2.3 The doubly-strict-or function $sor : \mathcal{D}(\mathbf{Bool} \times \mathbf{Bool}) \to \mathcal{D}(\mathbf{Bool})$ is the least monotone function satisfying:

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

sor is stable and sequential. Both `b.1` and `b.2` are sequentiality indices at $\emptyset$ for `b`.

The left-strict-or function $lor : \mathcal{D}(\mathbf{Bool} \times \mathbf{Bool}) \to \mathcal{D}(\mathbf{Bool})$ is the least monotone function satisfying:

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

lor is stable and sequential, with `b.1` as sequentiality index at $\emptyset$ for `b`.

The right-strict-or function $ror : \mathcal{D}(\mathbf{Bool} \times \mathbf{Bool}) \to \mathcal{D}(\mathbf{Bool})$ is defined analogously, and has `b.2` as sequentiality index at $\emptyset$ for `b`.

The parallel-or function $por : \mathcal{D}(\mathbf{Bool} \times \mathbf{Bool}) \to \mathcal{D}(\mathbf{Bool})$ is the least monotone function satisfying:

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

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 $\{\mathtt{b} = \mathtt{tt}\}$.

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

(1′) c' is not filled in $f(y)$ for any y above x, that is– $\forall y \in \mathcal{D}(M) \,.\, x \subseteq y \Rightarrow c' \notin \mathrm{F}(f(y))$.

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

Let $gf : \mathcal{D}((\mathbf{Bool} \times \mathbf{Bool}) \times \mathbf{Bool}) \to \mathcal{D}(\mathbf{Bool})$ be the least monotone function satisfying:

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

This is a variant of "Gustave's function" (attributed to Berry [Ber78] by Huet [Hue86]); *gf* is stable, but not sequential — it has no sequentiality index at $\emptyset$ for $\mathtt{b}$.

Let *min* : $\mathcal{D}(\mathbf{LNat} \times \mathbf{LNat}) \to \mathcal{D}(\mathbf{LNat})$ be the least continuous function such that, for all $x, y \in \mathcal{D}(\mathbf{LNat})$,

$$\begin{aligned}
min(\langle \ x \ , \ 0 \ \rangle) &= 0, \\
min(\langle \ 0 \ , \ x \ \rangle) &= 0, \\
min(\langle S(x), S(y)\rangle) &= S(min(x,y)).
\end{aligned}$$

For all $m, n \geq 0$, $min(\langle S^m(\perp), S^n(\perp)\rangle) = S^k(\perp)$, and $min(\langle S^m(0), S^n(0)\rangle) = S^k(0)$, where k is the minimum of m and n. In a fairly obvious sense *min* generalizes the parallel-or function by iteration, and it computes the minimum of two numbers presented in unary form. The function has no sequentiality index at $\langle\perp,\perp\rangle$ for $\mathtt{b}_0$. In fact, for each $n \geq 0$ it has no sequentiality index at $\langle S^n(\perp), S^n(\perp)\rangle$ for $\mathtt{b}_n$. ●

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

2.4 Sequential Exponentiation of DCDSs

The *sequential exponentiation* $M \to_{seq} M'$ is the DCDS $(\mathrm{C}, \mathrm{V}, \mathrm{E}, \vdash)$ defined as follows:

$\mathrm{C} = \mathcal{D}_{\mathrm{fin}}(M) \times \mathrm{C}_{M'}$. We denote a cell $(x, c') \in \mathrm{C}$ as xc'.

$\mathrm{V} = \{\mathbf{valof}\ c \mid c \in \mathrm{C}_M\} \cup \{\mathbf{output}\ v' \mid v' \in \mathrm{V}_{M'}\}$.

$\mathrm{E} = \{(xc', \mathbf{valof}\ c) \in \mathrm{C} \times \mathrm{V} \mid c \in \mathcal{A}(x)\} \cup \{(xc', \mathbf{output}\ v') \in \mathrm{C} \times \mathrm{V} \mid (c', v') \in \mathrm{E}_{M'}\}$.

$(xc', \mathbf{valof}\ c) \vdash yc'$ iff $y = x \cup \{(c, v)\}$ for some $v \in \mathrm{V}_M$.

$\{(x_j c'_j, \mathbf{output}\ v'_j)\}_{j=1}^{l} \vdash xc'$ iff $\{(c'_j, v'_j)\}_{j=1}^{l} \vdash_{M'} c'$ and $x = \cup\{x_j\}_{j=1}^{l}$.

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

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

$$a \cdot_{seq} x = \{(c', v') \mid \exists y \subseteq x \ . \ (yc', \mathbf{output}\ v') \in a\}.$$

A sequential algorithm between DCDSs may be viewed as a sequential function plus a computation strategy for that function. The function is embodied in the algorithm's input-output behavior; we say that $a \in \mathcal{D}(M \to_{seq} M')$ computes the input-output function $\lambda x \in \mathcal{D}(M) \ . \ a \cdot_{seq} x$. The computation strategy is embodied in the choice of the sequentiality index to be computed.

Intuitively, when a sequential algorithm is executed, computation is demand driven. For instance, an external observer's information about the result of applying an algorithm to an input state may be gradually increased by filling the cells of the result state, with each demand for the value of a result cell spawning a new computation. A cell of the exponentiation consists of a finite state x, describing the information currently known about the input, and a request for computation of a value for a cell c' in the output. The events of an algorithm associate with such a cell xc' a command: either an **output** v' command that terminates the computation and determines that (c', v') is in the output, or a **valof** c command that attempts to increase the current input state x at c. This c, naturally enough, is a sequentiality index (of the algorithm's input-output function) at x, so that the choice of c among all sequentiality indices at x (if not unique) determines the computation strategy. If the sub-computation for c terminates with the value v, the main computation resumes with the enabled cell $(x \cup \{(c, v)\})c'$, and so on until a value is output for c'. The sub-computation for c proceeds in the same manner: hence the overall coroutine-like flavor. Note that if one of the sub-computations fails to terminate, so does the main computation.

Sequential exponentiation preserves well foundedness and stability, and sequential application is well defined and continuous with respect to set inclusion. The category of DCDS and sequential algorithms is cartesian closed.

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

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

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

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

Figure 1: The sequential algorithms `lsor` and `lor`

Example 2.4 To display sequential algorithms we use vertical stacking to list elements of sets, *e.g.*, the events of a state.

There are two sequential algorithms that compute the doubly-strict-or function *sor*: `lsor`, shown in figure 1, which evaluates the two sequentiality indices in left-right order; and `rsor` (not shown) which evaluates in right-left order. `lor` in figure 1 is the unique sequential algorithm that computes the left-strict-or function *lor*. There is a similar unique sequential algorithm `ror` for the right-strict-or function *ror*. No sequential algorithm computes *por*. •

We have now summarized enough of Berry and Curien's work on sequentiality to establish a coherent background from which to develop our ideas on parallelism.

3 Parallel Algorithms between DCDSs

We want to be able to express algorithms for non-sequential functions, such as *por*, while retaining as far as possible suitable analogues to the semantic properties of sequential algorithms.

Sequential algorithms operate sequentially because a valof command may only start one sub-computation, and only after that sub-computation returns may the main computation proceed. A natural first step towards a generalization, then, would be to allow a valof command to start a number of sub-computations in parallel, and to specify a number of conditions, each based on the results of a finite subset of these sub-computations, under which the main computation may be resumed (without waiting for the completion of the remaining parallel sub-computations). For example, a parallel-or algorithm should, when nothing is yet known about its input, start sub-computations for the input cells b.1 and b.2, and the main computation may resume once the information about the input has been increased to either of $\{\texttt{b.1} = \texttt{tt}\}$, $\{\texttt{b.2} = \texttt{tt}\}$ or $\{\texttt{b.1} = \texttt{ff}, \texttt{b.2} = \texttt{ff}\}$. We call this generalization of the valof a *query* command.

We can represent a query value q as a set of finite functional sets of events: each element y of q represents a sufficient condition for resumption. A state x is said to *satisfy* a query q iff there exists $y \in q$ such that $y \subseteq x$. Given this interpretation it is natural to identify q with its upwards-closure: if $y \in q$ and $y \subseteq y'$ then every state satisfying q because of y' also satisfies q because of y. Moreover, if q_1 and q_2 are queries such that $q_1 \supseteq q_2$, every state satisfying q_2 will also satisfy q_1; intuitively, it may require less input information to satisfy q_1 than to satisfy q_2. This leads us to model queries as members of the Smyth powerdomain [Smy78] over a poset of finite functional sets of events (ordered by inclusion). Before we continue, we summarize some relevant details concerning the powerdomain.

Definition 3.1 The Smyth powerdomain $(\mathcal{P}_s(D), \sqsubseteq)$ of a poset $(D, \leq)$ is the set of all non-empty, upwards-closed subsets of D, ordered by reverse set inclusion. That is, for all $p \subseteq D$, $p \in \mathcal{P}_s(D)$ iff $\forall x, x' \in D.(x \in p \;\&\; x \leq x' \Rightarrow x' \in p)$; and, for all $p_1, p_2 \in \mathcal{P}_s(D)$, $p_1 \sqsubseteq p_2$ iff $p_1 \supseteq p_2$. •

A subset P of a Smyth powerdomain is consistent (denoted $\Uparrow P$) iff it has a non-empty intersection, in which case the least upper bound $\sqcup P$ is $\cap P$. We write $p_1 \Uparrow p_2$ when p_1 and p_2 are consistent. The union of a non-empty subset P of a Smyth powerdomain is its glb in the powerdomain, $\sqcap P = \cup P$. The least element of the powerdomain is the underlying set D.

Definition 3.2 A query q over a DCDS M is a non-trivial element of the Smyth powerdomain $(\mathcal{P}_s(\mathcal{F}_{\text{fin}}(M)), \sqsubseteq)$ over the poset $(\mathcal{F}_{\text{fin}}(M), \subseteq)$. •

The non-triviality condition is imposed since a query is meant to represent a non-trivial increment in information. It amounts to requiring that $\emptyset \notin q$ for any query q. Note that for all M, $(\mathcal{F}_{\text{fin}}(M), \subseteq)$ is a well founded poset. It follows that each query can be identified with its set of *minimal* elements, which we may call its *branches*. We write $\text{trim}(q)$ for the set of minimal elements of q, and $\text{up}(q)$ for the upwards closure of q. For all queries q we have $q = \text{up}(\text{trim}(q))$.

In order to ensure that our parallel algorithms compute deterministically, we need to guarantee that an algorithm issue the same output command for a given output cell whenever it is applied to consistent input states. For instance, the parallel-or algorithm associates the same command **output** tt with both of the input states $\{\texttt{b.1} = \texttt{tt}\}$, and $\{\texttt{b.2} = \texttt{tt}\}$, and the result is therefore unambiguous when the algorithm is applied to input $\{\texttt{b.1} = \texttt{tt}, \texttt{b.2} = \texttt{tt}\}$. We enforce determinism by using *sets* of states rather than single states to approximate the input, and by ensuring that consistent states are grouped together. For instance, the set of states

$\{\{\mathtt{b.1} = \mathtt{tt}\}, \{\mathtt{b.2} = \mathtt{tt}\}, \{\mathtt{b.1} = \mathtt{ff}, \mathtt{b.2} = \mathtt{ff}\}\}$ should be partitioned into $\{\{\mathtt{b.1} = \mathtt{tt}\}, \{\mathtt{b.2} = \mathtt{tt}\}\}$ and $\{\{\mathtt{b.1} = \mathtt{ff}, \mathtt{b.2} = \mathtt{ff}\}\}$.

More generally, the considerations that led us to use the Smyth powerdomain for queries lead us to use the Smyth powerdomain again, this time over the poset of finite states ordered by inclusion; and we give the following definition.

Definition 3.3 Given a DCDS M and subset p of $\mathcal{D}_{\mathsf{fin}}(M)$, define a relation of equivalence over p as follows: for all $y, y' \in p$, $y \approx y'$ iff there is a finite sequence of states in p that includes both y and y' such that each pair of consecutive states is consistent in $(\mathcal{D}_{\mathsf{fin}}(M), \subseteq)$. Write $p/_{\approx}$ for the set of equivalence classes of p.

A *class* over M is an element p of $\mathcal{P}_s(\mathcal{D}_{\mathsf{fin}}(M))$ such that $p/_{\approx} = \{p\}$. •

Clearly $\approx$ partitions any $p \in \mathcal{P}_s(\mathcal{D}_{\mathsf{fin}}(M))$ into classes with the property that states in distinct classes are inconsistent, as needed in order to guarantee determinism. Moreover, it produces the finest partitioning with this property, so that expressivity is not lost.

Whereas a sequential algorithm associated a command with cells of the form xc', a parallel algorithm will associate commands with cells of form pc, where p is a class. Intuitively, the elements of a class are states that an algorithm is forced, by determinism, to treat the same.

Up to this point it might seem that we are going to build the DCDS $M \to M'$ by using classes of M instead of single states and by replacing valof commands by queries over M. Indeed, such a simple generalization would be adequate for defining a parallel-or algorithm of type $\mathbf{Bool} \times \mathbf{Bool} \to \mathbf{Bool}$. However, this example is not general enough. Consider, for instance, the curried type $\mathbf{Bool} \to (\mathbf{Bool} \to \mathbf{Bool})$. Our determinism requirement would prevent any non-strict algorithm of this type from having both strict and non-strict results[5]. But a curried parallel-or algorithm should produce a strict result when applied to the empty input state, and a non-strict result when applied to $\{\mathtt{b} = \mathtt{tt}\}$, and therefore cannot be expressed using the framework described so far.

To permit a more general treatment we let algorithms issue queries that involve not only their *immediate* input state, but also the *successive* (or *residual*) arguments to which the algorithm may be applied. For the curried parallel-or example, an input of $\emptyset$ with a residual $\{\mathtt{b} = \mathtt{tt}\}$ or an input of $\{\mathtt{b} = \mathtt{tt}\}$ with a residual $\emptyset$ both lead to a *ground* result $\{\mathtt{b} = \mathtt{tt}\}$, once fully applied, while an input $\{\mathtt{b} = \mathtt{ff}\}$ with a residual $\{\mathtt{b} = \mathtt{ff}\}$ is inconsistent with both previous alternatives, and leads to a ground result of $\{\mathtt{b} = \mathtt{ff}\}$.

While this structuring idea does permit us to express curried algorithms, it could be argued that our solution is somewhat *ad hoc*. Indeed, as a result of this structure currying and uncurrying operations are "built in" and become simple operations on the internal structure of algorithms. We will return briefly at the end of the paper to the advantages and disadvantages of this approach.

We formalize these ideas by associating to each DCDS name M a *representation* DCDS rep(M) and a *base* DCDS base(M). We assume that DCDS names are built from a given collection of *atomic* DCDSs that contains at least **Null**, using the binary operators $\times$ (product) and $\to$ (arrow). We blur the distinction between a DCDS name and the DCDS it is intended to denote. We assume that atomic DCDSs mentioned earlier and the product of DCDSs are interpreted as given above.

Definition 3.4 A DCDS name is *basic* iff its outermost constructor is not $\to$.

If M is basic let rep(M) = **Null** and base(M) = M.

[5]The same output command that is associated with the empty input state would need to be associated with the other possible input states.

For $M \to M'$, let

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

•

We let both $\times$ and $\to$ associate to the *right* so as to correspond to the argument structure of an algorithm; for instance, if M_0 is basic, the DCDS $M_k \to \cdots \to M_1 \to M_0$ has $M_k \times \cdots \times M_1 \times \mathbf{Null}$ for its representation and M_0 for its base. Note that base(M) is always basic.

The classes used in constructing $M \to M'$ will be sets of finite states of $M \times \text{rep}(M')$; the M component embodies an approximation of the input, and the rep(M') component, or residual, will "make sense" in building a result of type M'. The cells of $M \to M'$ will be formed by pairing such classes with cells of base(M'), which represent the demands for computation of a result at base type. Similarly, the queries used in building algorithms of type $M \to M'$ will be sets of finite functional sets of events of $M \times \text{rep}(M')$.

Now that we use a representation, our query command generalizes both the valof and the output commands of the sequential exponentiation; operationally, a query only starts sub-computations for cells of the input type M; and the residuals may contribute to query events in the output algorithm. Again this is illustrated by the curried parallel-or algorithm. Its query may, obviously, only start one sub-computation, corresponding to the single cell of its argument; when the algorithm is applied to the input state $\emptyset$, the corresponding residual $\{\mathtt{b} = \mathtt{tt}\}$ will become (part of) a query of the result algorithm.

We extend the notions of a cell being filled, enabled and accessible in a natural way.

Definition 3.5 For $q \in \mathcal{P}_s(\mathcal{F}(M))$, a cell is *filled* in q iff it is filled in any of q's branches; $\text{F}(q) = \cup_{y \in \text{trim}(q)} \text{F}(y)$. A cell is *enabled* in q iff it is enabled in all of q's branches; $\text{E}(q) = \cap_{y \in \text{trim}(q)} \text{E}(y)$. A cell is *accessible* in q iff it is enabled in q and not filled in q; $\text{A}(q) = \text{E}(q) \setminus \text{F}(q)$. Equivalently, a cell is accessible in q iff it is accessible in all of q's branches; $\text{A}(q) = \cap_{y \in \text{trim}(q)} \text{A}(y)$. •

Definition 3.6 Let M and M' be DCDSs. Then $M \to M'$ is the DCDS $(\text{C}, \text{V}, \text{E}, \vdash)$ defined as follows. Let $M_\times$ abbreviate rep($M \to M'$) and let M_0 abbreviate base($M \to M'$).

$\text{C} = \mathcal{P}_s(\mathcal{D}_{\text{fin}}(M_\times)) \times \text{C}_{M_0}$. We denote a cell (p, c) of C as pc.

$\text{V} = \{\mathbf{query}\ q \mid q \in \mathcal{P}_s(\mathcal{F}_{\text{fin}}(M_\times))\ \&\ \emptyset \notin q\} \cup \{\mathbf{output}\ v \mid v \in \text{V}_{M_0}\}$

$\text{E} = \{(pc, \mathbf{query}\ q) \in \text{C} \times \text{V} \mid \text{F}(q) \subseteq \text{A}(p)\} \cup \{(pc, \mathbf{output}\ v) \in \text{C} \times \text{V} \mid (c, v) \in \text{E}_{M_0}\}$

$(p_1 c, \mathbf{query}\ q) \vdash pc$ iff $p \in (p_1 \sqcup q)/_{\approx}$.

$\{(p_j c_j, \mathbf{output}\ v_j)\}_{j=1}^{l} \vdash pc$ iff $\{(c_j, v_j)\}_{j=1}^{l} \vdash_{M_0} c$, $\Uparrow \{p_j\}_{j=1}^{l}$ and $p \in (\sqcup\{p_j\}_{j=1}^{l})/_{\approx}$.

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 up($\{\emptyset\}$)c, with c an initial cell of M_0. Note also that the construction guarantees that for each cell pc enabled in an algorithm p is indeed a class.

There are several obvious points that show how we have generalized the sequential definition. It is straightforward to define an embedding of sequential algorithms into the parallel algorithms that preserves operational behavior, producing a parallel algorithm that issues queries about a single

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

$$\texttt{por} = \left\{ \begin{array}{rl} \boxed{[\langle\emptyset,\emptyset\rangle]}\,\texttt{b=}\mathbf{query} & \boxed{\begin{array}{c} [\langle\{\texttt{ b=tt }\},\ \emptyset\rangle] \\ [\langle\emptyset\ ,\{\texttt{ b=tt }\}\rangle] \\ [\langle\{\texttt{ b=ff }\},\{\texttt{ b=ff }\}\rangle] \end{array}} \\ \boxed{\begin{array}{c} [\langle\{\texttt{ b=tt }\},\ \emptyset\rangle] \\ [\langle\emptyset\ ,\{\texttt{ b=tt }\}\rangle] \end{array}}\,\texttt{b=}\mathbf{output}\ \texttt{tt} & \\ \boxed{[\langle\{\texttt{ b=ff }\},\{\texttt{ b=ff }\}\rangle]}\,\texttt{b=}\mathbf{output}\ \texttt{ff} & \end{array} \right\}$$

Figure 2: The algorithm **por** for *por*

cell at a time. A sequential **valof** c command corresponds to a query whose branches are of the form $\{(c,v)\}$ (with an empty residual). The condition that a **query** q command can only be issued from cell pc_0 if $\mathrm{F}(q) \subseteq \mathrm{A}(p)$ corresponds to the requirement that a **valof** c command can only be issued from cell xc' if $c \in \mathrm{A}(x)$.

Example 3.7 In addition to the notation used for sequential algorithms, for parallel algorithms we use the following conventions. Classes and queries are framed in boxes, and we list only their minimal elements – branches. The branches themselves are enclosed in square brackets, using a shorthand notation for pairs: $\emptyset \in \mathcal{D}(\mathbf{Null})$ is denoted as $[]$, and $\langle y_0, [y_1, \ldots, y_d]\rangle$ is denoted as $[y_0, y_1, \ldots, y_d]$ for $d \geq 0$.

The unique algorithm for the parallel-or function is presented as **por** in Figure 2.

$$\texttt{lor} \in \mathcal{D}(\mathbf{Bool} \times \mathbf{Bool} \to \mathbf{Bool})$$

$$\texttt{lor} = \left\{ \begin{array}{rl} \boxed{[\langle\emptyset,\emptyset\rangle]}\,\texttt{b=}\mathbf{query} & \boxed{\begin{array}{c} [\langle\{\texttt{ b=tt }\},\emptyset\rangle] \\ [\langle\{\texttt{ b=ff }\},\emptyset\rangle] \end{array}} \\ \boxed{[\langle\{\texttt{ b=tt }\},\emptyset\rangle]}\,\texttt{b=}\mathbf{output}\ \texttt{tt} & \\ \boxed{[\langle\{\texttt{ b=ff }\},\emptyset\rangle]}\,\texttt{b=}\mathbf{query} & \boxed{\begin{array}{c} [\langle\emptyset,\{\texttt{ b=tt }\}\rangle] \\ [\langle\emptyset,\{\texttt{ b=ff }\}\rangle] \end{array}} \\ \boxed{[\langle\{\texttt{ b=ff }\},\{\texttt{ b=tt }\}\rangle]}\,\texttt{b=}\mathbf{output}\ \texttt{tt} & \\ \boxed{[\langle\{\texttt{ b=ff }\},\{\texttt{ b=ff }\}\rangle]}\,\texttt{b=}\mathbf{output}\ \texttt{ff} & \end{array} \right\}$$

Figure 3: The algorithm **lor** for *lor*

The (parallel) algorithms corresponding to the sequential algorithms **lor** and **lsor** from Figure 1 are shown in Figures 3 and 5. A second algorithm **plor**, for the function *lor*, presented in Figure 4, initiates computations for *both* input cells together. These three algorithms have corresponding

$$\texttt{plor} \in \mathcal{D}(\textbf{Bool} \times \textbf{Bool} \to \textbf{Bool})$$

$$\texttt{plor} = \left\{ \begin{array}{rl} \boxed{[\langle\emptyset,\emptyset\rangle]} & \texttt{b=query}\ \boxed{\begin{array}{l}[\langle\{\ \texttt{b=tt}\ \},\quad \emptyset\rangle]\\ [\langle\{\ \texttt{b=ff}\ \},\{\ \texttt{b=tt}\ \}\rangle]\\ [\langle\{\ \texttt{b=ff}\ \},\{\ \texttt{b=ff}\ \}\rangle]\end{array}} \\ \boxed{[\langle\{\ \texttt{b=tt}\ \},\emptyset\rangle]} & \texttt{b=output tt} \\ \boxed{[\langle\{\ \texttt{b=ff}\ \},\{\ \texttt{b=tt}\ \}\rangle]} & \texttt{b=output tt} \\ \boxed{[\langle\{\ \texttt{b=ff}\ \},\{\ \texttt{b=ff}\ \}\rangle]} & \texttt{b=output ff} \end{array} \right\}$$

Figure 4: The algorithm plor for *lor*

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

$$\texttt{lsor} = \left\{ \begin{array}{rl} \boxed{[\langle\emptyset,\emptyset\rangle]} & \texttt{b=query}\ \boxed{\begin{array}{l}[\langle\{\ \texttt{b=tt}\ \},\emptyset\rangle]\\ [\langle\{\ \texttt{b=ff}\ \},\emptyset\rangle]\end{array}} \\ \boxed{[\langle\{\ \texttt{b=tt}\ \},\emptyset\rangle]} & \texttt{b=query}\ \boxed{\begin{array}{l}[\langle\emptyset,\{\ \texttt{b=tt}\ \}\rangle]\\ [\langle\emptyset,\{\ \texttt{b=ff}\ \}\rangle]\end{array}} \\ \boxed{[\langle\{\ \texttt{b=tt}\ \},\{\ \texttt{b=tt}\ \}\rangle]} & \texttt{b=output tt} \\ \boxed{[\langle\{\ \texttt{b=tt}\ \},\{\ \texttt{b=ff}\ \}\rangle]} & \texttt{b=output tt} \\ \boxed{[\langle\{\ \texttt{b=ff}\ \},\emptyset\rangle]} & \texttt{b=query}\ \boxed{\begin{array}{l}[\langle\emptyset,\{\ \texttt{b=tt}\ \}\rangle]\\ [\langle\emptyset,\{\ \texttt{b=ff}\ \}\rangle]\end{array}} \\ \boxed{[\langle\{\ \texttt{b=ff}\ \},\{\ \texttt{b=tt}\ \}\rangle]} & \texttt{b=output tt} \\ \boxed{[\langle\{\ \texttt{b=ff}\ \},\{\ \texttt{b=ff}\ \}\rangle]} & \texttt{b=output ff} \end{array} \right\}$$

Figure 5: The algorithm lsor for *sor*

$$\mathtt{psor} \in \mathcal{D}(\mathbf{Bool} \times \mathbf{Bool} \to \mathbf{Bool})$$

$$\mathtt{psor} = \left\{ \begin{array}{l} \boxed{[\langle\emptyset,\emptyset\rangle]}\,\mathtt{b{=}query}\ \boxed{\begin{array}{l} [\langle\{\ \mathtt{b{=}tt}\ \},\{\ \mathtt{b{=}tt}\ \}\rangle] \\ [\langle\{\ \mathtt{b{=}tt}\ \},\{\ \mathtt{b{=}ff}\ \}\rangle] \\ [\langle\{\ \mathtt{b{=}ff}\ \},\{\ \mathtt{b{=}tt}\ \}\rangle] \\ [\langle\{\ \mathtt{b{=}ff}\ \},\{\ \mathtt{b{=}ff}\ \}\rangle] \end{array}} \\ \boxed{[\langle\{\ \mathtt{b{=}tt}\ \},\{\ \mathtt{b{=}tt}\ \}\rangle]}\,\mathtt{b{=}output\ tt} \\ \boxed{[\langle\{\ \mathtt{b{=}tt}\ \},\{\ \mathtt{b{=}ff}\ \}\rangle]}\,\mathtt{b{=}output\ tt} \\ \boxed{[\langle\{\ \mathtt{b{=}ff}\ \},\{\ \mathtt{b{=}tt}\ \}\rangle]}\,\mathtt{b{=}output\ tt} \\ \boxed{[\langle\{\ \mathtt{b{=}ff}\ \},\{\ \mathtt{b{=}ff}\ \}\rangle]}\,\mathtt{b{=}output\ ff} \end{array} \right\}$$

Figure 6: The algorithm psor for *sor*

$$\mathtt{plsor} \in \mathcal{D}(\mathbf{Bool} \times \mathbf{Bool} \to \mathbf{Bool})$$

$$\mathtt{plsor} = \left\{ \begin{array}{l} \boxed{[\langle\emptyset,\emptyset\rangle]}\,\mathtt{b{=}query}\ \boxed{\begin{array}{l} [\langle\{\ \mathtt{b{=}tt}\ \},\quad \emptyset\rangle] \\ [\langle\{\ \mathtt{b{=}ff}\ \},\{\ \mathtt{b{=}tt}\ \}\rangle] \\ [\langle\{\ \mathtt{b{=}ff}\ \},\{\ \mathtt{b{=}ff}\ \}\rangle] \end{array}} \\ \boxed{[\langle\{\ \mathtt{b{=}tt}\ \},\emptyset\rangle]}\,\mathtt{b{=}query}\ \boxed{\begin{array}{l} [\langle\emptyset,\{\ \mathtt{b{=}tt}\ \}\rangle] \\ [\langle\emptyset,\{\ \mathtt{b{=}ff}\ \}\rangle] \end{array}} \\ \boxed{[\langle\{\ \mathtt{b{=}tt}\ \},\{\ \mathtt{b{=}tt}\ \}\rangle]}\,\mathtt{b{=}output\ tt} \\ \boxed{[\langle\{\ \mathtt{b{=}tt}\ \},\{\ \mathtt{b{=}ff}\ \}\rangle]}\,\mathtt{b{=}output\ tt} \\ \boxed{[\langle\{\ \mathtt{b{=}ff}\ \},\{\ \mathtt{b{=}tt}\ \}\rangle]}\,\mathtt{b{=}output\ tt} \\ \boxed{[\langle\{\ \mathtt{b{=}ff}\ \},\{\ \mathtt{b{=}ff}\ \}\rangle]}\,\mathtt{b{=}output\ ff} \end{array} \right\}$$

Figure 7: The algorithm plsor for *sor*

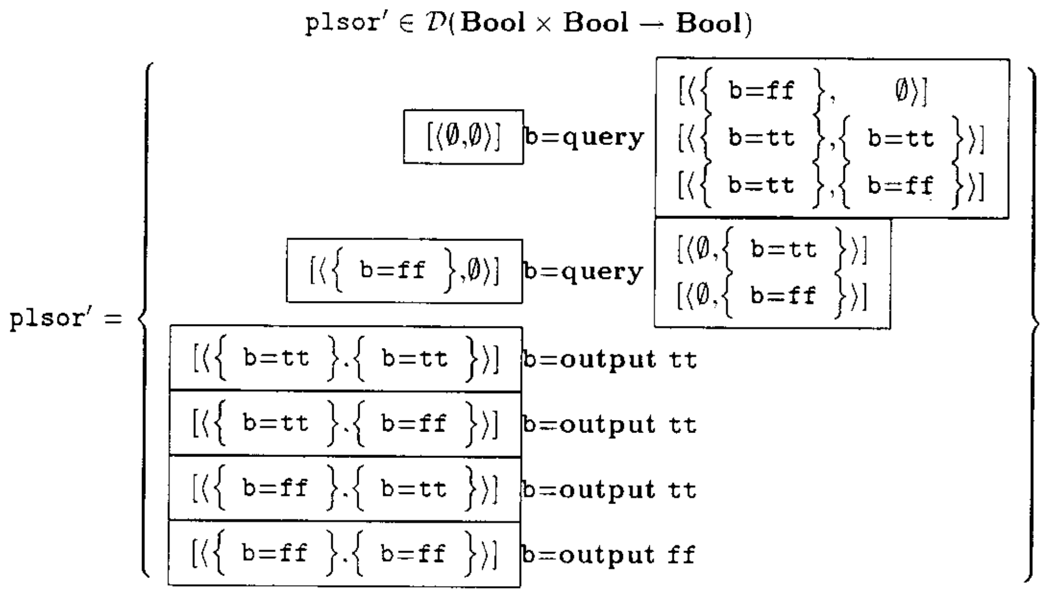

plsor′ ∈ 𝒟(Bool × Bool → Bool)
plsor′ =
[⟨∅,∅⟩] b=query
[⟨{ b=ff }, ∅⟩]
[⟨{ b=tt },{ b=tt }⟩]
[⟨{ b=tt },{ b=ff }⟩]
[⟨{ b=ff },∅⟩] b=query
[⟨∅,{ b=tt }⟩]
[⟨∅,{ b=ff }⟩]
[⟨{ b=tt },{ b=tt }⟩] b=output tt
[⟨{ b=tt },{ b=ff }⟩] b=output tt
[⟨{ b=ff },{ b=tt }⟩] b=output tt
[⟨{ b=ff },{ b=ff }⟩] b=output ff

Figure 8: The algorithm **plsor′** for *sor*

right-handed versions: **ror**, **rsor** and **pror**, respectively (not shown).

For the doubly-strict-or function *sor*, there are several algorithms which employ a parallel computation strategy, initiating computations for both input cells together. Figure 6 presents the algorithm **psor**, in an obvious sense the "most eager" algorithm for *sor*; additional algorithms for *sor* that compute in parallel are **plsor** and **plsor′**, presented in figures 7 and 8, and the corresponding right-handed versions **prsor** and **prsor′** (not shown). •

Example 3.8 Figure 9 presents an algorithm **gf** for the function *gf*. Note that every class of **gf** has a least element. A variant for which this is not true is the algorithm **gf′** (Figure 10) for the function $gf' : \mathcal{D}((\mathbf{Bool} \times \mathbf{Bool}) \times \mathbf{Bool}) \to \mathcal{D}(\mathbf{Bool})$, defined to be the least monotone function satisfying:

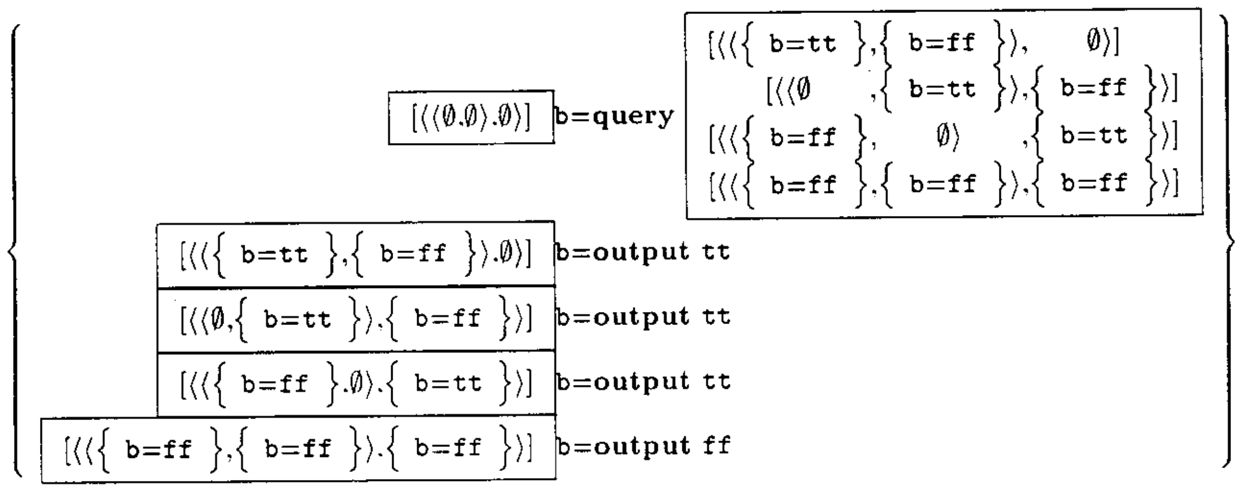
$$
\begin{aligned}
gf'(\langle\langle\{\mathtt{b}=\mathtt{tt}\},\{\mathtt{b}=\mathtt{ff}\}\rangle,\ \emptyset\ \rangle) &= \{\mathtt{b}=\mathtt{tt}\}\\
gf'(\langle\langle\ \emptyset\ ,\{\mathtt{b}=\mathtt{tt}\}\rangle,\{\mathtt{b}=\mathtt{ff}\}\rangle) &= \{\mathtt{b}=\mathtt{tt}\}\\
gf'(\langle\langle\{\mathtt{b}=\mathtt{ff}\},\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

Like *gf*, *gf′* has no sequentiality index at $\emptyset$. In contrast to *gf*, *gf′* is also not stable — there is no unique minimal state below $\langle\langle\{\mathtt{b}=\mathtt{tt}\},\{\mathtt{b}=\mathtt{ff}\}\rangle,\{\mathtt{b}=\mathtt{ff}\}\rangle$ for which *gf′* attains $\{\mathtt{b}=\mathtt{tt}\}$; correspondingly, not all classes of **gf′** have a least element. •

Example 3.9 Figure 11 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. •

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

$\mathbf{gf} =$



[⟨⟨∅,∅⟩,∅⟩] b=query
[⟨⟨{ b=tt },{ b=ff }⟩, ∅⟩]
[⟨⟨∅ ,{ b=tt }⟩,{ b=ff }⟩]
[⟨⟨{ b=ff }, ∅⟩ ,{ b=tt }⟩]
[⟨⟨{ b=ff },{ b=ff }⟩,{ b=ff }⟩]
[⟨⟨{ b=tt },{ b=ff }⟩,∅⟩] b=output tt
[⟨⟨∅,{ b=tt }⟩,{ b=ff }⟩] b=output tt
[⟨⟨{ b=ff },∅⟩,{ b=tt }⟩] b=output tt
[⟨⟨{ b=ff },{ b=ff }⟩,{ b=ff }⟩] b=output ff

Figure 9: The algorithm **gf** for gf

Example 3.10 The parallel algorithm $\mathtt{min} \in \mathcal{D}(\mathbf{LNat} \times \mathbf{LNat} \to \mathbf{LNat})$ for computing the function min on pairs of lazy natural numbers is given in Figure 12.

●

3.1 Elementary Properties of $M \to M'$

We now prove some simple properties of $M \to M'$ leading to the proof that $M \to M'$ is well defined: whenever M and M' are DCDSs so is $M \to M'$.

Proposition 3.11 *If $y \vdash_{M\to M'} p_1c$, $y \vdash_{M\to M'} p_2c$ and $p_1 \Uparrow p_2$, then $p_1 = p_2$.*

Proof: Intersecting equivalence classes are equal. ■

Proposition 3.12 *If $(pc, \mathbf{query}\ q) \vdash_{M\to M'} p'c$ then $p \sqsubset p'$ and for every $\bar{x}' \in p'$ there exists $\bar{x} \in p$ such that $\bar{x} \subset \bar{x}'$.*

Proof: If $\bar{x}' \in p'$ then $\bar{x}' \in p \sqcup q$. For some $\bar{x} \in \mathrm{trim}(p)$ and $\bar{y} \in \mathrm{trim}(q)$, $\bar{x} \cup \bar{y} \subseteq \bar{x}'$. Since $\bar{y} \neq \emptyset$ and $\mathrm{F}(\bar{y}) \subseteq \mathrm{A}(\bar{x})$, it follows that $\bar{x} \subset \bar{x}'$. ■

Proposition 3.13 *$M \to M'$ is a well founded CDS.*

Proof: Define the relation $\ll_{M\to M'}$ over $\mathcal{D}_{\mathrm{fin}}(M_\times) \times \mathrm{C}_{M_0}$ as follows:

$$\bar{x}c \ll_{M\to M'} \bar{x}'c' \text{ iff either } (\bar{x} \subseteq \bar{x}' \ \&\ c \ll_{M_0} c') \text{ or } (\bar{x} \subset \bar{x}' \ \&\ c = c').$$

It is easy to establish the following implications:

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

$$\mathbf{gf}' =$$

$$\left\{ \begin{array}{l}
\boxed{[\langle\langle\emptyset,\emptyset\rangle,\emptyset\rangle]}\ \mathtt{b}{=}\mathbf{query}\ \boxed{\begin{array}{l}
[\langle\langle\{\ \mathtt{b{=}tt}\ \},\{\ \mathtt{b{=}ff}\ \}\rangle,\ \emptyset\rangle] \\
[\langle\langle\emptyset,\{\ \mathtt{b{=}tt}\ \}\rangle,\{\ \mathtt{b{=}ff}\ \}\rangle] \\
[\langle\langle\{\ \mathtt{b{=}ff}\ \},\ \emptyset\rangle,\{\ \mathtt{b{=}tt}\ \}\rangle] \\
[\langle\langle\{\ \mathtt{b{=}ff}\ \},\{\ \mathtt{b{=}tt}\ \}\rangle,\ \emptyset\rangle] \\
[\langle\langle\emptyset,\{\ \mathtt{b{=}ff}\ \}\rangle,\{\ \mathtt{b{=}tt}\ \}\rangle] \\
[\langle\langle\{\ \mathtt{b{=}tt}\ \},\ \emptyset\rangle,\{\ \mathtt{b{=}ff}\ \}\rangle] \\
[\langle\langle\{\ \mathtt{b{=}ff}\ \},\{\ \mathtt{b{=}ff}\ \}\rangle,\{\ \mathtt{b{=}ff}\ \}\rangle]
\end{array}} \\
\boxed{\begin{array}{l}
[\langle\langle\{\ \mathtt{b{=}tt}\ \},\{\ \mathtt{b{=}ff}\ \}\rangle,\ \emptyset\rangle] \\
[\langle\langle\emptyset,\{\ \mathtt{b{=}tt}\ \}\rangle,\{\ \mathtt{b{=}ff}\ \}\rangle] \\
[\langle\langle\{\ \mathtt{b{=}ff}\ \},\ \emptyset\rangle,\{\ \mathtt{b{=}tt}\ \}\rangle] \\
[\langle\langle\{\ \mathtt{b{=}ff}\ \},\{\ \mathtt{b{=}tt}\ \}\rangle,\ \emptyset\rangle] \\
[\langle\langle\emptyset,\{\ \mathtt{b{=}ff}\ \}\rangle,\{\ \mathtt{b{=}tt}\ \}\rangle] \\
[\langle\langle\{\ \mathtt{b{=}tt}\ \},\ \emptyset\rangle,\{\ \mathtt{b{=}ff}\ \}\rangle]
\end{array}}\ \mathtt{b}{=}\mathbf{output\ tt} \\
\boxed{[\langle\langle\{\ \mathtt{b{=}ff}\ \},\{\ \mathtt{b{=}ff}\ \}\rangle,\{\ \mathtt{b{=}ff}\ \}\rangle]}\ \mathtt{b}{=}\mathbf{output\ ff}
\end{array} \right\}$$

Figure 10: The algorithm **gf′** for *gf′*

$$\mathtt{id}_{\mathtt{Nat}} \in \mathcal{D}(\mathbf{Nat} \rightarrow \mathbf{Nat})$$

$$\begin{array}{rcl}
\mathtt{id}_{\mathtt{Nat}} & = & \left\{\ \boxed{[\emptyset]}\ \mathtt{n}{=}\mathbf{query}\ \cup_{k\in\mathbb{N}} \boxed{[\{\ \mathtt{n}{=}k\ \}]}\ \right\} \\
& & \cup(\cup_{k\in\mathbb{N}} \left\{\ \boxed{[\{\ \mathtt{n}{=}k\ \}]}\ \mathtt{n}{=}\mathbf{output}\ k\ \right\})
\end{array}$$

Figure 11: The identity algorithm on **Nat**

$$\mathtt{min} \in \mathcal{D}(\mathbf{LNat} \times \mathbf{LNat} \rightarrow \mathbf{LNat})$$

$$\mathtt{min} = \cup_{i=0}^{\infty}\mathtt{min}_i, \text{ where, for each } i \geq 0,$$

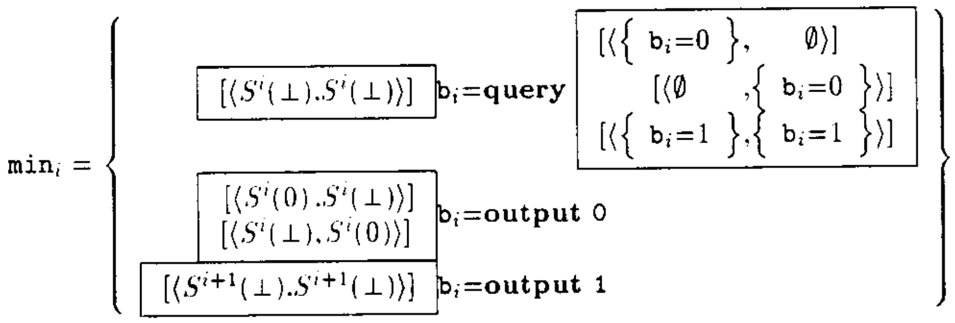

min_i =
[⟨S^i(⊥),S^i(⊥)⟩] b_i=query
[⟨{ b_i=0 }, ∅⟩]
[⟨∅ ,{ b_i=0 }⟩]
[⟨{ b_i=1 },{ b_i=1 }⟩]
[⟨S^i(0),S^i(⊥)⟩]
[⟨S^i(⊥),S^i(0)⟩] b_i=output 0
[⟨S^{i+1}(⊥),S^{i+1}(⊥)⟩] b_i=output 1

Figure 12: The **min** algorithm

(1) If $\ll_{M_0}$ has an infinite descending chain, then so does $\ll_{M \rightarrow M'}$. This is because if $c \ll_{M_0} c'$ then $\text{up}(\{\emptyset\})c \ll_{M \rightarrow M'} \text{up}(\{\emptyset\})c'$.

(2) If $\ll_{M \rightarrow M'}$ has an infinite descending chain, then so does $\lll_{M \rightarrow M'}$. This is because, when $pc \ll_{M \rightarrow M'} p'c'$, for each $\bar{x}' \in p'$ there exists $\bar{x} \in p$ such that $\bar{x}c \lll_{M \rightarrow M'} \bar{x}'c'$ (using proposition 3.12).

(3) If $\lll_{M \rightarrow M'}$ has an infinite descending chain, then so does $\ll_{M_0}$. This follows from the finiteness of the states involved.

By these implications, well foundedness of any CDS coincides with well foundedness of its base, and hence $M \rightarrow M'$ is well founded iff M' is. ∎

We now prove the Tree Lemma, a technical result corresponding to an analogous lemma proven by Berry and Curien for sequential algorithms. Our proof is similar to theirs. This lemma is the basis for a tree-like notation for algorithms and is useful in reasoning about the structure of algorithms. As an added benefit, the tree lemma establishes stability of $M \rightarrow M'$.

Lemma 3.14 (Tree Lemma) *Let a be a state of $M \rightarrow M'$.*

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

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

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

(2) Every cell $pc \in \text{E}(a)$ has a unique enabling in a.

Proof: By induction on c in $\ll_{M_0}$, where we take $M_0 = \text{base}(M \rightarrow M')$.

Let $pc, p'c \in \text{E}(a)$, such that $p \Uparrow p'$. Let $\hat{p} = p \sqcup p'$. Examine the last few enablings in a leading to pc (respectively $p'c$), starting with the last output enabling. There must be such an output enabling, by well foundedness of $M \rightarrow M'$. Let us name the constituents of these enablings as follows:

$$\{(r_jd_j.\mathbf{output}\ v_j)\}_{j=1}^{l} \vdash_a (p_0c.\mathbf{query}\ q_1) \vdash_a \cdots \vdash_a (p_{k-1}c.\mathbf{query}\ q_k) \vdash_a p_kc = pc$$

$$\{(r'_j d'_j, \textbf{output } v'_j)\}_{j=1}^{l'} \vdash_a (p'_0 c, \textbf{query } q'_1) \vdash_a \cdots \vdash_a (p'_{k'-1}c, \textbf{query } q'_{k'}) \vdash_a p'_{k'}c = p'c.$$

Assume that $k \leq k'$. We show by induction on m that, for all $m \leq k$, $p_m = p'_m$.

- For the base case, we show that $p_0 = p'_0$.
  Let $x = \{(d, v) \mid d \ll^{+}_{M_0} c \;\&\; \exists r \,.\, (rd, \textbf{output } v) \in a \;\&\; r \sqsubseteq \hat{p}\}$. Clearly, x is a set of events of M_0. We show that $x \in \mathcal{D}(M_0)$:
  Safety: If $(d, v) \in x$ then $rd \in \mathrm{F}(a)$ for some $r \sqsubseteq \hat{p}$. Let $\{(s_j c_j, \textbf{output } w_j)\}_{j=1}^{m} \vdash_a r'd$, with $r' \sqsubseteq r$. Such an enabling exists, because rd has a proof in a, and that proof must have a last output enabling. Therefore $\{(c_j, w_j)\}_{j=1}^{m} \vdash_x d$.
  Functionality: If $(d, v), (d, v') \in x$ then $(rd, \textbf{output } v), (r'd, \textbf{output } v') \in a$ for some $r, r' \sqsubseteq \hat{p}$, so that, by induction hypothesis (1b), $r = r'$, and $v = v'$.
  The state x contains both $\{(d_j, v_j)\}_{j=1}^{l}$ and $\{(d'_j, v'_j)\}_{j=1}^{l'}$, two enablings of c, which must be equal by stability of M_0; so $l = l'$, and, without loss of generality, $\forall j \leq l \,.\, d_j = d'_j$.
  Now, for any $j \leq l$, $d_j = d'_j \ll^{+}_{M_0} c$, and $r_j \Uparrow r'_j$, and by induction hypothesis (1b) we have $r_j = r'_j$. Consequently, by proposition 3.11, $p_0 = p'_0$.
- For the inductive step, assuming $m + 1 \leq k$ and $p_m = p'_m$, we get $q_{m+1} = q'_{m+1}$ by functionality. It follows by proposition 3.11 that $p_{m+1} = p'_{m+1}$.

From the above, it follows that (assuming $k \leq k'$) $p_k = p'_k$, so $pc \ll^{*}_{a} p'c$, and there exists a query chain in a from pc to $p'c$ (of length $k' - k \geq 0$). If $k' \leq k$, we can similarly show that $p'c \ll^{*}_{a} pc$. Therefore (1a) holds.

Assume that $(pc, \textbf{output } v), (p'c, \textbf{output } v') \in a$. If $k < k'$ then $(pc, \textbf{query } q'_{k+1}) \in a$, contradicting functionality. So $k' \leq k$. By symmetry, $k \leq k'$; thus $k = k'$ and $p = p'$, and (1b) holds.

Finally, to show uniqueness of the enabling for pc, take $p = p'$ in the above argument for (1a) and suppose there are two enablings for pc in a. Since $\ll$ is well founded, we must get $k = k'$, and the argument shows that the enablings are equal. ∎

Corollary 3.15 *$M \multimap M'$ is a DCDS.*

3.2 Currying

Currying and uncurrying operators on algorithms are easy to define, given our use of rep in structuring the components from which an algorithm is built. Recall that

$$\begin{aligned} \mathrm{rep}(M_1 \times M_2 \multimap M') &= (M_1 \times M_2) \times \mathrm{rep}(M'), \\ \mathrm{rep}(M_1 \multimap M_2 \multimap M') &= M_1 \times (M_2 \times \mathrm{rep}(M')). \end{aligned}$$

Definition 3.16 Define the map $\mathrm{curry} : \mathcal{F}(\mathrm{rep}(M_1 \times M_2 \multimap M')) \to \mathcal{F}(\mathrm{rep}(M_1 \multimap M_2 \multimap M'))$ by

$$\mathrm{curry}(\langle\langle y_1, y_2\rangle, \vec{y}'\rangle) = \langle y_1, \langle y_2, \vec{y}'\rangle\rangle.$$

This function extends to queries, cells, commands, and algorithms as follows:

$$\mathrm{curry}(q) = \{\mathrm{curry}(\vec{y}) \mid \vec{y} \in q\}. \qquad \mathrm{curry}(pc) = \mathrm{curry}(p)c,$$

$$\mathrm{curry}(\textbf{query } q) = \textbf{query } \mathrm{curry}(q). \qquad \mathrm{curry}(\textbf{output } v) = \textbf{output } v.$$

$$\mathrm{curry}(a) = \{(\mathrm{curry}(p)c, \mathrm{curry}(u)) \mid (pc, u) \in a\}.$$

The uncurrying function may be defined similarly.

Proposition 3.17 *The map* curry : $\mathcal{D}(M_1 \times M_2 \to M') \to \mathcal{D}(M_1 \to M_2 \to M')$ *is an isomorphism and preserves enablings.*

Proof: Straightforward. ∎

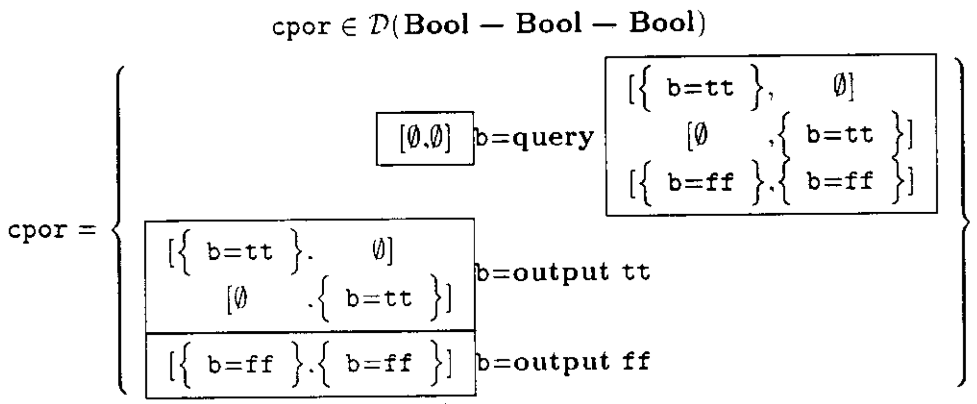


Figure 13: The curried parallel-or algorithm, **cpor** = curry(**por**)

Example 3.18 Figure 13 presents **cpor** = curry(**por**), the curried version of **por**. Figure 14 presents the fully curried version of **gf**$'$. •

Note that currying the parallel-or function *por* to *cpor* reduces parallelism, in an informal sense, shown by a comparison of the **por** and **cpor** algorithms. **por**'s query initiates two parallel sub-computations, while **cpor**'s query initiates a single sub-computation. Even though **cpor** does not compute in parallel, the *cpor* function is not sequential (as defined in section 2.3) since it is not even monotone with respect to set inclusion – contrast $cpor(\emptyset)$ and $cpor(\{\mathtt{b} = \mathtt{tt}\})$. This observation is a premonition of problems we will encounter with application.

4 Application

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

$$a \cdot_{seq} x = \{(c', v') \mid \exists y \subseteq x\ .\ (yc'.\mathbf{output}\ v') \in a\}.$$

One might read this as saying that the events (c', v') of $a \cdot_{seq} x$ are obtained by "projection" from output events $(yc', \mathbf{output}\ v')$ of a whose state component y is below x, and thus may be an accurate partial description of the input x.

Consider the application of a parallel algorithm $a \in \mathcal{D}(M \to M')$ to $x \in \mathcal{D}(M)$, producing a result which we will write as $a \cdot x$. Intuitively, there ought to be an operational correspondence between the events of a and the events of $a \cdot x$, in the rough sense that for each event $(pc, u) \in a$ there are some events of $a \cdot x$ which are responsible for $a \cdot x$ exhibiting the same behavior that (pc, u) entails when the argument to a is known to be x. Given our use of residuals in constructing the events of $M \to M'$, p is a set of finite states of $M \times \text{rep}(M')$ and each query q in a is a set of finite

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

$$\mathbf{cgf}' =$$

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

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

functional sets of events of $M \times \mathrm{rep}(M')$. By analogy with the sequential case, given a class p and an input state x we will be interested in the set of residuals derived from elements of p whose input component approximates x; and similarly for a query q. We therefore define a projection operator π_x on queries (and classes) as follows.

Definition 4.1 For $x \in \mathcal{D}(M)$, and $q \in \mathcal{P}_s(\mathcal{F}_{\mathrm{fin}}(\mathrm{rep}(M \to M')))$, define

$$\pi_x(q) = \{\bar{y} \mid \exists y \subseteq x \ . \ \langle y, \bar{y} \rangle \in q\}.$$

$\pi_x(q)$ is either empty, or in $\mathcal{P}_s(\mathcal{F}_{\mathrm{fin}}(\mathrm{rep}(M')))$. •

An event (pc, u) of a for which $\pi_x(p) = \emptyset$ is irrelevant when a is applied to input state x, because x is not approximated by any element of p. Even when $\pi_x(p)$ is not empty it need not be a single equivalence class of states: the residuals obtained from equivalent but inconsistent states of p need not remain equivalent in $\pi_x(p)$. When this happens we must split $\pi_x(p)$ into its equivalence classes; in this way, a single cell of a may project onto more than one cell of $a \cdot x$.

Now consider a query event $(pc, \mathbf{query}\ q)$ of a, and suppose that $\pi_x(p)$ is not empty. There are three possibilities: either $\pi_x(q)$ is itself a query over $\mathrm{rep}(M')$; or $\pi_x(q)$ contains the empty set; or else $\pi_x(q)$ is the empty set. If $\pi_x(q)$ is a proper query, then we should obtain an event $(p'c, \mathbf{query}\ \pi_x(q))$ in $a \cdot x$ for each equivalence class p' of $\pi_x(p)$. If $\emptyset \in \pi_x(q)$, then when applying a to x the query q is satisfied by the input state alone, and no further query needs to be issued concerning the residual arguments. However, some events following pc in a may contribute events to $a \cdot x$. Such a query is said to be *fully satisfied* by application of a to x. Finally, if $\pi_x(q) = \emptyset$ then when a is applied to x the computation can progress up to a point where the query q is issued, but cannot go further because q cannot be satisfied; there should therefore be no events in $a \cdot x$ corresponding to $(pc, \mathbf{query}\ q)$ or any event following it in a.

Similarly, an output event $(pc, \mathbf{output}\ v)$ of a projects iff $\pi_x(p)$ is not empty, in which case we obtain an event $(p'c, \mathbf{output}\ v)$ of $a \cdot x$ for each equivalence class p' of $\pi_x(p)$.

We therefore extend π_x to a (partial) map from $V_{M \to M'}$ to $V_{M'}$ as follows, and give a formal definition of application that makes these ideas precise:

$$\pi_x(\mathbf{query}\ q) = \mathbf{query}\ \pi_x(q) \quad \text{and} \quad \pi_x(\mathbf{output}\ v) = \mathbf{output}\ v.$$

Definition 4.2 Let $a \in \mathcal{D}(M \to M')$ and $x \in \mathcal{D}(M)$. The application of a to x is defined by[6]

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

•

The requirement that events of $a \cdot x$ belong to $\mathrm{E}_{M'}$ filters out empty projections and trivial queries.

We remark that when the Berry-Curien algorithms are embedded in the parallel framework, a valof command is either not projected by an application, or else it is fully satisfied, since all residuals are vacuous. Correspondingly, the sequential application need only project the output events.

[6]When M' is basic, this definition of $a \cdot x$ produces not a state of M' but a degenerate "nullary algorithm" built from rep **Null** and base M'. Its events are of the form $(\{\emptyset\}c, \mathbf{output}\ v)$, with (c, v) an event of M', since there are no legal queries over **Null**. Such nullary algorithms are isomorphic to states of M' by replacing each $(\{\emptyset\}c, \mathbf{output}\ v)$ by (c, v); we will omit explicit mention of this isomorphism in the definition of application and related development for simplicity of presentation.

Example 4.3 Consider the application of the curried parallel-or algorithm **cpor** to $\{\mathtt{b} = \mathtt{ff}\}$. The result is the identity algorithm on **Bool**. There is a clear one-to-one correspondence between the events of **cpor** and $\mathbf{cpor} \cdot \{\mathtt{b} = \mathtt{ff}\}$: each event of **cpor** projects onto a unique event of $\mathbf{cpor} \cdot \{\mathtt{b} = \mathtt{ff}\}$.

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

●

Example 4.4 Consider the application of **cpor** to $\emptyset$. The resulting algorithm does not have (or need) an event with an **output ff** command, because projection of the **output ff** event of **cpor** does not produce a valid class or event.

$$\mathtt{cpor}\cdot\emptyset = \left\{\begin{array}{l} \qquad\qquad\boxed{[\emptyset]}\,\mathtt{b{=}}\mathbf{query}\ \boxed{[\{\ \mathtt{b{=}tt}\ \}]} \\ \boxed{[\{\ \mathtt{b{=}tt}\ \}]}\,\mathtt{b{=}}\mathbf{output}\ \mathtt{tt}\end{array}\right\}$$

●

Example 4.5 The query of **cpor** is fully satisfied when **cpor** is applied to $\{\mathtt{b} = \mathtt{tt}\}$, and the result is a non-strict constant algorithm.

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

●

Example 4.6 Splitting occurs when we apply the algorithm $\mathbf{cgf}'$ (figure 14) to $\emptyset$.

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

●

4.1 Elementary Properties of Application

We now show that application is well defined. We begin by introducing two maps $\mathrm{root}_{a,x}$ and $\mathrm{source}_{a,x}$ to make precise the correspondence between events of $a \cdot x$ and a. These maps are not generally surjective, since some events fail to project. They are also not injective, because of possible splitting.

Definition 4.7 For $a \in \mathcal{D}(M - M')$ and $x \in \mathcal{D}(M)$, define $\mathrm{root}_{a,x} : \mathrm{F}(a \cdot x) \to \mathrm{F}(a)$ and $\mathrm{source}_{a,x} : \mathrm{F}(a \cdot x) - \mathrm{F}(a)$ by:

$\mathrm{root}_{a,x}(p'c) = p_0 c$ where $p_0 = \sqcap\{p \mid pc \in \mathrm{E}(a)\ \&\ p' \in \pi_x(p)/_\approx\}$,

$\mathrm{source}_{a,x}(p'c) = p_1 c$ iff $\exists u\ .\ (p_1 c, u) \in a\ \&\ (p'c, \pi_x(u)) \in a \cdot x\ \&\ p' \in \pi_x(p_1)/_\approx$.

●

Proposition 4.8 $\mathrm{root}_{a,x}$ *and* $\mathrm{source}_{a,x}$ *are well defined. Moreover, for any* $p'c \in \mathrm{F}(a \cdot x)$,

(1) $\mathrm{root}_{a,x}(p'c) \ll_a^* \mathrm{source}_{a,x}(p'c)$, *and*

(2) For any $pc \in \mathrm{E}(a)$, $p' \in \pi_x(p)/_\approx$ *iff* $\mathrm{root}_{a,x}(p'c) \ll_a^* pc \ll_a^* \mathrm{source}_{a,x}(p'c)$.

Proof: Let $(p'c, u') \in a \cdot x$, $C = \{pc \in \mathrm{E}(a) \mid p' \in \pi_x(p)/_\approx\}$, $P = \{p \mid pc \in C\}$ and $p_0 = \sqcap P$; C is non-empty, by definition of application.

All classes in P are consistent, so that, by the tree lemma, the cells in C form a $\ll$-chain. By well foundedness of $\ll$, C has $\ll$-minimal element, and that must be $p_0 c$. $\mathrm{root}_{a,x}(p'c)$ is uniquely determined to be $p_0 c$. Moreover, $p_0 c$ is clearly filled, so that $\mathrm{root}_{a,x}$ is well defined.

By definition of application, there exists some $(p_1 c, u) \in a$ such that $p' \in \pi_x(p_1)/_\approx$ and $u' = \pi_x(u)$; obviously, $p_1 c \in C$. Assume that there exists $pc \in C$ such that $p_1 c \ll pc$, *i.e.*, such that $u =$ **query** q and $(p_1 c, \textbf{query}\ q) \vdash_a pc$. But since $p' \in \pi_x(p)/_\approx$, this necessarily implies $\emptyset \in \pi_x(q)$, a contradiction. It follows that $p_1 c$ must be $\ll$-maximal in C. $\mathrm{source}_{a,x}(p'c)$ is uniquely determined to be $p_1 c$, so that $\mathrm{source}_{a,x}$ is well defined.

Moreover, we have shown that $\mathrm{root}_{a,x}(p'c)$ and $\mathrm{source}_{a,x}(p'c)$ are $\ll$-minimal and $\ll$-maximal, respectively, in C, and $C \subseteq \{pc \in \mathrm{E}(a) \mid \mathrm{root}_{a,x}(p'c) \ll_a^* pc \ll_a^* \mathrm{source}_{a,x}(p'c)\}$. The converse inclusion follows from monotonicity of projection. ■

Proposition 4.9 *For* $a \in \mathcal{D}(M - M')$ *and* $x \in \mathcal{D}(M)$, $a \cdot x$ *is a state of* M'.

Proof: Clearly, $a \cdot x \subseteq \mathrm{E}_{M'}$.

To show functionality of $a \cdot x$, note that, for $(p'c, u') \in a \cdot x$, u' is uniquely determined to be $\pi_x(u)$ where $(\mathrm{source}_{a,x}(p'c), u) \in a$.

We now show that $a \cdot x$ is safe. Let $p'c \in \mathrm{F}(a \cdot x)$, $pc = \mathrm{root}_{a,x}(p'c)$ and $\{(p_j c_j, u_j)\}_{j=1}^{l} \vdash_a pc$.

For every $j \leq l$, $p_j \sqsubseteq p$, and by monotonicity of projection, $\pi_x(p_j) \sqsubseteq \pi_x(p)$, so there must exist a unique $p'_j \in \pi_x(p_j)/_\approx$ such that $p'_j \sqsubseteq p'$.

If pc has an output enabling, *i.e.*, each u_j has form **output** v_j, then $p \in (\sqcup\{p_j\}_{j=1}^{l})/_\approx$, and it must be that $p' \in (\sqcup\{p'_j\}_{j=1}^{l})/_\approx$, so that $\{(p'_j c_j, \textbf{output}\ v_j)\}_{j=1}^{l} \vdash_{a \cdot x} p'c$.

If pc has a query enabling then $l = 1$, $u_1 =$ **query** q for some q, and $(p_1 c, \textbf{query}\ q) \vdash_a pc$, *i.e.*, $p \in (p_1 \sqcup q)/_\approx$. Clearly, $\pi_x(q) \neq \emptyset$, and further, by $\ll$-minimality of $\mathrm{root}_{a,x}(p'c)$, $p'_1 \neq p'$ so that $\emptyset \notin \pi_x(q)$, and $(p'_1 c, \textbf{query}\ \pi_x(q)) \in a \cdot x$. It is easy to show that $(p'_1 c, \textbf{query}\ \pi_x(q)) \vdash_{a \cdot x} p'c$, *i.e.*, that $p' \in (p'_1 \sqcup \pi_x(q))/_\approx$. ■

Now that $a \cdot x$ has been shown to be a state, we extend $\mathrm{root}_{a,x} : \mathrm{F}(a \cdot x) \to \mathrm{F}(a)$ to $\mathrm{root}_{a,x} : \mathrm{E}(a \cdot x) - \mathrm{E}(a)$, using the same definition given above, and complement proposition 4.8 as follows:

Corollary 4.10 $\mathrm{root}_{a,x} : \mathrm{E}(a \cdot x) - \mathrm{E}(a)$ *is well defined. Moreover, for any* $p'c \in \mathrm{E}(a \cdot x)$,

$y' \vdash_{a \cdot x} p'c$ *iff* $\mathrm{source}_{a,x}(y') \vdash_a \mathrm{root}_{a,x}(p'c)$.

where $\mathrm{source}_{a,x}(y') = \{(\mathrm{source}_{a,x}(p'c), u) \in a \mid (p'c, \pi_x(u)) \in y'\}$.

5.1 The Intensional Strictness Order

Definition 5.2 The *intensional strictness* pre-order $\leq^i_M$ on $\mathcal{D}(M)$ is defined by induction on M as follows.

For an atomic DCDS M let $\leq^i_M$ be set inclusion.

For a product $M_1 \times M_2$ let $\leq^i_{M_1\times M_2}$ be defined componentwise: $\langle x_1, x_2\rangle \leq^i_{M_1\times M_2} \langle x'_1, x'_2\rangle$ iff $x_1 \leq^i_{M_1} x'_1$ and $x_2 \leq^i_{M_2} x'_2$.

For an arrow type $M \to M'$ and $x, x' \in \mathcal{D}(M \to M')$, let $x \leq^i_{M\to M'} x'$ iff there exists a function $f : \mathrm{E}(x) \to \mathrm{E}(x')$ such that the following hold:

(1) If $f(pc) = p'c'$ then $c = c'$ and $p' \sqsubseteq p$.

(2) If $(pc, \textbf{output}\ v) \in x$ then $(f(pc), \textbf{output}\ v) \in x'$.

(3) If $\{(p_jc_j, \textbf{output}\ v_j)\}^l_{j=1} \vdash_x pc$ then $\{(f(p_jc_j), \textbf{output}\ v_j)\}^l_{j=1} \vdash_{x'} f(pc)$. Note that, by taking $l = 0$, f must map initial cells into initial cells.

(4) If $(pc, \textbf{query}\ q) \in x$ then one of the following holds:

(WKN) There exists $q' \sqsubseteq q$ such that $(f(pc), \textbf{query}\ q') \in x'$, and if $(pc, \textbf{query}\ q) \vdash_x p_1c$ then $(f(pc), \textbf{query}\ q') \vdash_{x'} f(p_1c)$.

In such a case we say that f weakens $(pc, \textbf{query}\ q)$.

(ABS) If $(pc, \textbf{query}\ q) \vdash_x p_1c$ then $f(p_1c) = f(pc)$.

In such a case we say that f abstracts $(pc, \textbf{query}\ q)$.

We call such an f a *morphism*. We say that $x \leq^i x'$ by f in cases where we need to mention the morphism explicitly. We will often drop the subscript M from $\leq^i_M$. ●

In other words, a morphism f preserves basic cells, output commands and output enablings, and may either weaken a query or abstract it. Roughly speaking, if $x \leq^i x'$ then x' is less strict than x in the sense that it may require less information about the inputs, and may ask for it at an earlier point of the computation, in order to produce at least the same outputs as x.

Example 5.3 Note that our previous counter-examples to monotonicity (example 5.1) become examples of algorithms related by $\leq^i$, since $\texttt{cpor}\cdot\emptyset \leq^i \texttt{cpor}\cdot\{\texttt{b} = \texttt{ff}\}$ and $\texttt{cpor}\cdot\emptyset \leq^i \texttt{cpor}\cdot\{\texttt{b} = \texttt{tt}\}$. We also have $\texttt{gf} \leq^i \texttt{gf}'$, by weakening. ●

Example 5.4 We further illustrate $\leq^i$ by relating the algorithms introduced in example 3.7. These algorithms differ in strictness of the computed function, and in their computation strategies. We have $\texttt{psor} \leq^i \texttt{plor} \leq^i \texttt{por}$ by weakening, and $\texttt{plsor} \leq^i \texttt{plor}$ by abstraction; $\texttt{plsor}' \leq^i \texttt{lor}$ by weakening; and on the sequential algorithms we have $\texttt{lsor} \leq^i \texttt{lor}$ by abstraction. The remaining relationships may be inferred by left-right symmetry and transitivity. Figure 15 summarizes the relationships between these algorithms. Note that the algorithms for *sor* are pairwise incomparable, and the two algorithms for *lor* are incomparable.

In each of these simple examples a suitable morphism is easy to construct. ●

4.2 Input-Output Functions

Our definition of the input-output function computed by a parallel algorithm is similar to the Berry-Curien definition for sequential algorithms. In fact, the embedding of the sequential algorithms into the parallel algorithms mentioned earlier preserves the function computed by an algorithm. Again this shows that our notion of application is a sensible generalization of the sequential definition.

Definition 4.11 The *input-output function* of an algorithm $a \in \mathcal{D}(M \rightarrow M')$ is the function $\lambda x \in \mathcal{D}(M)\ .\ a \cdot x$, mapping states of M to states of M'. •

Example 4.12 Each of the algorithms discussed in example 3.7 computes the corresponding function: for instance, `por` computes *por*; both `lor` and `plor` compute *lor*; each of `lsor`, `plsor`, `plsor'`, and `psor` computes *sor*. Similarly, the `min` algorithm (example 3.10) computes *min*. •

We can also show now that currying and application interact correctly.

Proposition 4.13 *For any $a \in \mathcal{D}((M_1 \times M_2) \rightarrow M')$, $x_1 \in \mathcal{D}(M_1)$ and $x_2 \in \mathcal{D}(M_2)$,*

$$(\mathrm{curry}(a) \cdot x_1) \cdot x_2 = a \cdot \langle x_1, x_2 \rangle .$$

Proof: Immediate. ▪

In other words, if a computes f, then curry(a) computes curry(f). Again, a corresponding property holds for uncurrying.

5 Ordering Algorithms

Application as defined above is monotone and continuous in its first argument with respect to the set inclusion ordering on algorithms, but not even monotone in its second argument. This is caused by two phenomena, which we call *weakening* and *abstraction* of queries.

Contrast $a \cdot x$ and $a \cdot x'$, for an algorithm a and $x \subseteq x'$. Clearly, increasing the argument from x to x' may increase the set of elements of a query q of a whose input conditions are satisfied, so that $\pi_x(q) \subseteq \pi_{x'}(q)$ (and $\pi_{x'}(q) \sqsubseteq \pi_x(q)$). If $\pi_x(q)$ is a valid query then $\pi_{x'}(q)$ is non-empty, and we need to ask whether x' fully satisfies q.

- If $\pi_{x'}(q)$ is a valid query, *i.e.*, $\emptyset \notin \pi_{x'}(q)$, then we say that the query $\pi_x(q)$ of $a \cdot x$ is weakened into the query $\pi_{x'}(q)$ of $a \cdot x'$.
- If x' fully satisfies q, *i.e.*, $\emptyset \in \pi_{x'}(q)$, then we say that the query $\pi_x(q)$ is abstracted.

Example 5.1 Consider, *e.g.*, $\mathtt{cpor} \cdot \emptyset \not\subseteq \mathtt{cpor} \cdot \{\mathtt{b} = \mathtt{ff}\}$ and $\mathtt{cpor} \cdot \emptyset \not\subseteq \mathtt{cpor} \cdot \{\mathtt{b} = \mathtt{tt}\}$, owing to weakening and abstraction, respectively. •

The counter-examples above cannot be resolved by modifying the definition of application, since they are simple and intuitively correct, and serve as guidelines to which any definition of application must conform. The desire for monotonicity and continuity of application therefore motivates a coarser order than inclusion on states; we define a pre-order $\leq^i$ based on the existence of a *morphism* between algorithms that preserves enabling structure up to weakening and abstraction.

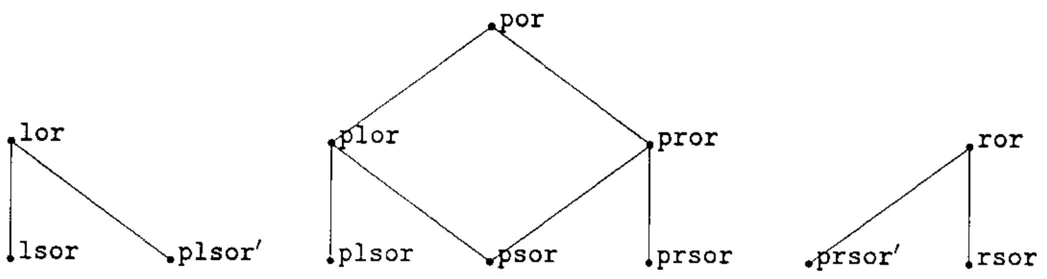


Figure 15: The or-algorithms related by $\leq^i$

5.2 First order DCDSs

Strictly speaking, now that we have determined that set inclusion is not appropriate as the underlying order for our model, we should go back and examine what happens to our construction of $M \rightarrow M'$ when we employ $\leq^i$ instead of set inclusion. However, it is easy to see that this would make no difference in the construction of *first order* DCDSs, defined to be the Ms generated by the following grammar, where A is atomic:

$$M \ ::= \ P \mid P \rightarrow M \qquad P \ ::= \ A \mid P \times P.$$

Algorithms of first order type may return algorithms as results but do not take algorithms as arguments. All examples of algorithms discussed so far have been first order, and the class of first order DCDSs is closed under currying and uncurrying. When M is first order the set inclusion ordering on rep(M) coincides with the intensional strictness ordering, so that the first order algorithm space and the definition of application remain unchanged if we use $\leq^i$ instead of $\subseteq$ as the underlying order. For the rest of this development we focus on first order DCDSs, and we show that our model provides a satisfactory account of first order algorithms. At the end of the paper we will discuss briefly why a more radical solution is needed at higher order types.

5.3 Order-theoretic Properties

Proposition 5.5 *If $a \leq^i a'$ by f then $f(a) = \{(f(pc), u') \in a' \mid pc \in \mathrm{F}(a)\}$ is a state. For any $pc \in \mathrm{E}(a)$, if $f(pc) = pc$ then no event that precedes pc in a is abstracted.*

Proof: Functionality of $f(a)$ is inherited from a'. Safety of $f(a)$ may be shown by induction on the number of abstracted query events below a cell $pc \in \mathrm{E}(a)$. The same proof may be adapted to show that no abstraction may occur below pc if $f(pc) = pc$. ∎

Proposition 5.6 *$\leq^i$ contains the set inclusion relation, and, in particular, is reflexive. The empty set is a least element in $\leq^i$.*

Proof: If $a \subseteq a'$ then the identity embedding of $\mathrm{E}(a)$ into $\mathrm{E}(a')$ is clearly a morphism showing that $a \leq^i a'$. ∎

Note that the intensional order properly contains set inclusion, since (for instance) `lsor` $\leq^i$ `lor` but `lsor` $\not\subseteq$ `lor`.

Proposition 5.7 *If $a \leq^i a'$ by f and $a' \leq^i a''$ by f' then $a \leq^i a''$ by $f' \circ f$.*

Proof: Properties (1), (2) and (3) in definition 5.2 are obviously preserved by composition of morphisms. We check property (4). Let $(pc, \mathbf{query}\ q) \in a$.

- If f abstracts $(pc, \mathbf{query}\ q)$ then so does $f' \circ f$.
- If f weakens $(pc, \mathbf{query}\ q)$ into $(p'c, \mathbf{query}\ q') \in a'$ which is then abstracted by f' then $f' \circ f$ abstracts $(pc, \mathbf{query}\ q)$.
- If f weakens $(pc, \mathbf{query}\ q)$ into $(p'c, \mathbf{query}\ q') \in a'$ which is then weakened by f' into $(p''c, \mathbf{query}\ q'') \in a''$, then $f' \circ f$ weakens $(pc, \mathbf{query}\ q)$ into $(p''c, \mathbf{query}\ q'')$. ■

Thus, $\leq^i$ is reflexive and transitive. However, $\leq^i$ is not anti-symmetric in general. Intuitively, this is because queries that do not have an output event following them may be abstracted and re-introduced at will, thus generating distinct but $\leq^i$-equivalent algorithms.

Example 5.8 Consider $\leq^i_{\mathbf{Bool-Bool}}$. We have

$$\emptyset \leq^i \left\{ \boxed{[\emptyset]}\, \texttt{b=}\mathbf{query}\ \boxed{[\{\ \texttt{b=tt}\ \}]} \right\} \leq^i \emptyset.$$

by inclusion and abstraction respectively. •

However, $\leq^i$ is anti-symmetric, and hence a partial order, on algorithms all of whose queries lead to output events, since in such cases abstraction cannot be "undone". We make this precise as follows.

Definition 5.9 A cell $pc \in \mathrm{E}(a)$ is *observable* in a iff there is an output event $(p_0c, \mathbf{output}\ v) \in a$ such that $pc \ll^*_a p_0c$. An event is observable iff its cell is observable, and an algorithm is observable iff all of its events are observable. •

Proposition 5.10 *$\leq^i$ is anti-symmetric on observable algorithms.*

Proof: Assume a and a' are observable algorithms, $a \leq^i a'$ by f, and $a' \leq^i a$ by g.

For any output event $(pc, \mathbf{output}\ v)$ of a, $(g \circ f(pc), \mathbf{output}\ v)$ is also an output event of a. By the tree lemma, $g \circ f(pc) = f(pc) = pc$. By proposition 5.5, no event preceding $(pc, \mathbf{output}\ v)$ may be abstracted by $g \circ f$. Therefore $g \circ f$ may not abstract any observable event. Since a is observable $g \circ f$ may not abstract at all. It is easy to adapt the case analysis in the proof of proposition 5.7 to deduce that f itself may not abstract.

Let $(pc, \mathbf{query}\ q) \in a$ be a query event that is weakened by $g \circ f$. It is weakened by f to $(f(pc), \mathbf{query}\ q') \in a'$, which is in turn weakened by g to $(g \circ f(pc), \mathbf{query}\ q'') \in a$, with $q'' \sqsubseteq q' \sqsubseteq q$. But by the tree lemma and since $q'' \sqsubseteq q$, $(pc, \mathbf{query}\ q) = (g \circ f(pc), \mathbf{query}\ q'')$, and consequently $(pc, \mathbf{query}\ q) = (f(pc), \mathbf{query}\ q')$.

Therefore f may not abstract any of the events of a, and all weakenings are identities. We thus have $a \subseteq a'$, and, symmetrically, $a' \subseteq a$. ■

Corollary 5.11 *$\leq^i$ is a pre-order, and it is a partial order on observable algorithms.*

Every algorithm has a unique observable algorithm to which it is $\leq^i$-equivalent, by abstraction of the non-observable queries, and by inclusion, respectively. This means that we lose no generality if we concern ourselves mainly with observable algorithms.

5.4 Distinguished Morphisms

There may be several morphisms between two algorithms, as in the following example.

Example 5.12 Let $\mathsf{a}_1, \mathsf{a}_3 \in \mathcal{D}(\mathbf{Bool} \times \mathbf{Bool} - \mathbf{Bool})$ be the following algorithms:

$$\mathsf{a}_1 = \left\{ \begin{array}{r l} \boxed{\emptyset} & \mathtt{b}{=}\mathbf{query}\ \boxed{[\{\ \mathtt{b.1{=}tt}\ \}]} \\ \boxed{[\{\ \mathtt{b.1{=}tt}\ \}]} & \mathtt{b}{=}\mathbf{query}\ \boxed{[\{\ \mathtt{b.2{=}tt}\ \}]} \\ \boxed{[\left\{ \begin{array}{l} \mathtt{b.1{=}tt} \\ \mathtt{b.2{=}tt} \end{array} \right\}]} & \mathtt{b}{=}\mathbf{output}\ \mathtt{tt} \end{array} \right\}$$

$$\mathsf{a}_3 = \left\{ \begin{array}{r l} \boxed{\emptyset} & \mathtt{b}{=}\mathbf{query}\ \boxed{\begin{array}{l} [\{\ \mathtt{b.1{=}tt}\ \}] \\ [\{\ \mathtt{b.2{=}tt}\ \}] \end{array}} \\ \boxed{\begin{array}{l} [\{\ \mathtt{b.1{=}tt}\ \}] \\ [\{\ \mathtt{b.2{=}tt}\ \}] \end{array}} & \mathtt{b}{=}\mathbf{output}\ \mathtt{tt} \end{array} \right\}$$

There are two morphisms showing that $\mathsf{a}_1 \leq^i \mathsf{a}_3$: one morphism weakens the first query and abstracts the second, while the other morphism abstracts the first query and weakens the second query. ●

We may, however, characterize a unique *distinguished morphism* $\delta_{a,a'}$ whenever $a \leq^i a'$. Intuitively, a distinguished morphism is defined inductively so that it always weakens whenever possible. Thus, in the previous example, only the morphism that weakens the initial cell is a distinguished morphism. We make these ideas more precise as follows.

First, let $a, a' \in \mathcal{D}(M - M')$, with a an observable algorithm.

Definition 5.13 Define a partial function $\delta_{a,a'} : \mathrm{E}(a) - \mathrm{E}(a')$ by induction on $pc \in \mathrm{E}(a)$:

$$\delta_{a,a'}(pc) = \begin{cases} p'c & \text{if } \{(p_jc_j.\mathbf{output}\ v_j)\}_{j=1}^{l} \vdash_a pc \ \&\ p' \sqsubseteq p \\ & \quad \&\ \{(\delta_{a,a'}(p_jc_j).\mathbf{output}\ v_j)\}_{j=1}^{l} \vdash_{a'} p'c \\ p'c & \text{if } (p_1c.\mathbf{query}\ q) \vdash_a pc \ \&\ p' \sqsubseteq p \\ & \quad \&\ \exists q' \sqsubseteq q\ .\ (\delta_{a,a'}(p_1c).\mathbf{query}\ q') \vdash_{a'} p'c \\ \delta_{a,a'}(p_1c) & \text{if } (p_1c.\mathbf{query}\ q) \vdash_a pc \ \&\ \neg(\exists q' \sqsubseteq q\ .\ (\delta_{a,a'}(p_1c).\mathbf{query}\ q') \in a') \\ \text{undefined} & \text{otherwise} \end{cases}$$

●

We say that $\delta_{a,a'}$ *preserves* the output event $(pc.\mathbf{output}\ v) \in a$ iff $(\delta_{a,a'}(pc), \mathbf{output}\ v) \in a'$.

Proposition 5.14 *$\delta_{a,a'}$ is well defined as a partial function, and its domain is downwards closed with respect to $\ll_a^*$. If $\delta_{a,a'}$ preserves all output events of a that precede pc then $\delta_{a,a'}$ is defined on pc: if $\delta_{a,a'}$ preserves all output events of a then $\delta_{a,a'}$ is a total function.*

Proof: By induction on $pc \in \mathrm{E}(a)$. ∎

Proposition 5.15 *$\delta_{a,a'}$ preserves all output events of a iff $a \leq^i a'$ by $\delta_{a,a'}$.*

Proof: By definition of $\delta_{a,a'}$. ∎

Proposition 5.16 *If $a \leq^i a'$ by f then, for every $pc \in \mathrm{E}(a)$,*

(1) There exists $\hat{p}c \ll^*_a pc$ such that $f(pc) = \delta_{a,a'}(\hat{p}c)$.

(2) $\delta_{a,a'}$ preserves all output events that precede pc in a.

(3) For any $pc \in \mathrm{E}(a)$, $f(pc) \ll^*_{a'} \delta_{a,a'}(pc)$.

Proof: By induction on $pc \in \mathrm{E}(a)$.

We assume the following immediate properties of $\delta_{a,a'}$:

(a) If $\delta_{a,a'}(pc)$ is defined and $p'c = \delta_{a,a'}(pc)$ then $p' \sqsubseteq p$.

(b) If $\delta_{a,a'}(pc)$ is defined and $\hat{p}c \ll^*_a pc$ then $\delta_{a,a'}(\hat{p}c) \ll^*_{a'} \delta_{a,a'}(pc)$.

Let $pc \in \mathrm{E}(a)$. Note that by induction hypothesis (2) and Proposition 5.14, $\delta_{a,a'}(pc)$ is defined.

(1) If $\{(p_jc_j, \textbf{output } v_j)\}_{j=1}^{l} \vdash_a pc$ then, by induction hypothesis (2), the enabling is preserved in a', so that $\{(\delta_{a,a'}(p_jc_j), \textbf{output } v_j)\}_{j=1}^{l} \vdash_{a'} \delta_{a,a'}(pc)$. Similarly, since f is a morphism, $\{(f(p_jc_j), \textbf{output } v_j)\}_{j=1}^{l} \vdash_{a'} f(pc)$. For $j \leq l$, the classes $\delta_{a,a'}(p_jc_j)$ and $f(p_jc_j)$ are both upper-bounded by p_j, so that, by the tree lemma, $\delta_{a,a'}(p_jc_j) = f(p_jc_j)$ for each j. Also, the classes $\delta_{a,a'}(pc)$ and $f(pc)$ are upper-bounded by p, so by proposition 3.11, $\delta_{a,a'}(pc) = f(pc)$.

If $(p_1c, \textbf{query } q) \vdash_a pc$ then, by induction hypothesis (1), there exists $\hat{p}_1c \ll^*_a p_1c$ such that $f(p_1c) = \delta_{a,a'}(\hat{p}_1c) \ll^*_{a'} \delta_{a,a'}(p_1c)$. If $f(pc) = f(p_1c)$ then we have shown (1). If, on the other hand, $(f(p_1c), \textbf{query } q'') \vdash_{a'} f(pc)$, with $q'' \sqsubseteq q$, then it is also the case that $(\delta_{a,a'}(\hat{p}_1c), \textbf{query } q'') \vdash_{a'} f(pc)$. We cannot assume that $\hat{p}_1c$ is $\delta_{a,a'}$-weakened, but there is certainly such a cell on the query chain from $\hat{p}_1c$ to p_1c (since p_1c itself qualifies). By well-foundedness, there is a first such cell, say $\hat{p}_2c$: it is the first cell on the query chain whose query is above q''. Since $\hat{p}_2c$ is $\delta_{a,a'}$-weakened, then $(\hat{p}_2c, \textbf{query } q_2) \vdash_a \hat{p}c \ll^*_a pc$, with $q'' \sqsubseteq q_2$, so that $(\delta_{a,a'}(\hat{p}_2c), \textbf{query } q'') \vdash_{a'} \delta_{a,a'}(\hat{p}c)$, and $f(pc) = \delta_{a,a'}(\hat{p}c)$, again by proposition 3.11.

(2) Let $(pc, \textbf{output } v) \in a$. Then $(f(pc), \textbf{output } v) \in a'$ and by (1) there exists $\hat{p}c \ll^*_a pc$ such that $f(pc) = \delta_{a,a'}(\hat{p}c)$. It follows that $\delta_{a,a'}(pc) = \delta_{a,a'}(\hat{p}c)$.

(3) Follows from (1) and (b). ∎

Corollary 5.17 *If $a \leq^i a'$ then $a \leq^i a'$ by $\delta_{a,a'}$. Moreover, $\delta_{a,a'}$ is the unique morphism g that weakens whenever possible, i.e., such that whenever $(pc, \textbf{query } q) \in a$, $(g(pc), \textbf{query } q') \in a'$ and $q' \sqsubseteq q$ then g weakens pc.*

Proof: Whenever $a \leq^i a'$, $\delta_{a,a'}$ preserves all output events and thus is a morphism. Note that $\delta_{a,a'}$ weakens whenever possible. It is easy to show by induction on $pc \in \mathrm{E}(a)$, that for any morphism g that weakens whenever possible, $g(pc) = \delta_{a,a'}(pc)$. ∎

The definition of distinguished morphisms $\delta_{a,a'}$ can be extended to the case where a is not an observable algorithm, by making $\delta_{a,a'}$ abstract all non-observable events of a.

The composition of distinguished morphisms is not necessarily distinguished, as in the following example.

Example 5.18 Consider the algorithms a_1 and a_3 in example 5.12 above and the algorithm a_2 given here,

$$\mathsf{a}_2 = \left\{ \begin{array}{l} \boxed{\emptyset}\,\texttt{b=}\mathbf{query}\,\boxed{[\{\ \texttt{b.2=tt}\ \}]} \\ \boxed{[\{\ \texttt{b.2=tt}\ \}]}\,\texttt{b=}\mathbf{output}\ \texttt{tt} \end{array} \right\}.$$

Clearly, $\mathsf{a}_1 \leq^i \mathsf{a}_2 \leq^i \mathsf{a}_3$, and $\delta_{\mathsf{a}_1,\mathsf{a}_3}$ is the morphism which weakens the first query and abstracts the second; but $\delta_{\mathsf{a}_1,\mathsf{a}_3} \neq \delta_{\mathsf{a}_2,\mathsf{a}_3} \circ \delta_{\mathsf{a}_1,\mathsf{a}_2}$, because $\delta_{\mathsf{a}_1,\mathsf{a}_2}$ abstracts the initial query and therefore the composition is "forced" to abstract too early. •

However, the following can be said concerning composition of distinguished morphisms.

Proposition 5.19 *If $a \leq^i a'$ and $a' \leq^i a''$, and $\delta_{a,a'}$ does not abstract at any cell preceding pc, and $\delta_{a',a''}$ does not abstract at any cell preceding $\delta_{a,a'}(pc)$, then $\delta_{a,a''}(pc) = \delta_{a',a''} \circ \delta_{a,a'}(pc)$.*

Proof: By induction on pc. ▪

5.5 Limits of Directed Sets

A subset X of a partial order or pre-order $(D, \leq)$ is *directed* iff it is non-empty and every pair of elements of X has an upper bound in X. $(D, \leq)$ is said to be *directed complete* iff every directed subset has a lub.

We start by defining directed complete partial orders on values and events, which we denote $\leq^i$ again. We then show, using distinguished morphisms, that the intensional strictness order $\leq^i$ on algorithms is directed complete.

Definition 5.20 For values u and u' of $M \multimap M'$, let $u \leq^i u'$ iff $u = u' = \mathbf{output}\ v$, or $u = \mathbf{query}\ q$ and $u' = \mathbf{output}\ v$, or $u = \mathbf{query}\ q$ and $u' = \mathbf{query}\ q'$ with $q' \sqsubseteq q$.

For events (pc, u) and $(p'c', u')$ of $M \multimap M'$, let $(pc, u) \leq^i (p'c', u')$ iff $p' \sqsubseteq p$, $c = c'$ and $u \leq^i u'$. •

Proposition 5.21 *For all M and M', $\leq^i$ is a directed complete partial order on values and events of $M \multimap M'$.*

Proof: Clearly, $\leq^i$ is a partial order on values and on events.

The lub of a directed set of values U is given by

$$\vee^i U = \begin{cases} \mathbf{output}\ v & \text{if } \mathbf{output}\ v \in U, \\ \mathbf{query}\ \sqcap \{q \mid \mathbf{query}\ q \in U\} & \text{otherwise.} \end{cases}$$

The lub is well defined by directedness of U.

The lub of a directed set of events E is (pc, u) where c is the unique basic cell mentioned in E, $p = \sqcap\{p' \mid p'c \in \mathrm{F}(E)\}$, and $u = \vee^i U$ for $U = \{u' \mid \exists(p'c, u') \in E\}$. Directedness of U follows from directedness of E. (pc, u) is a valid event if u is an output. If u is **query** q and $c' \in \mathrm{F}(q)$ then, by directedness, c' is filled in all queries of E from some point on, so that it is accessible in all classes of E from some point on, and therefore $c' \in \mathrm{A}(p)$. ▪

Throughout this section, let A be a directed set of algorithms. For $a \in A$, let A_a be the subset $\{a' \in A \mid a \leq^i a'\}$.

The key concept in constructing limits is *persistence*. A cell is persistent if it, all cells preceding it, and their images by distinguished morphisms in A, are never abstracted.

Definition 5.22 A cell pc is *persistent* from a if it is filled in a and for every $p'c' \ll_a^* pc$, $a' \in A_a$ and $a'' \in A_{a'}$, $\delta_{a',a''}$ does not abstract at $\delta_{a,a'}(p'c')$.

A cell pc is *persistently enabled* from a if it has an enabling $y \vdash_a pc$ such that all cells filled in y are persistent from a. •

If a cell is persistent (respectively, persistently enabled) then so is any cell preceding it, and so is its image by a distinguished morphism in A. Every persistent cell is persistently enabled. Note that, since every cell has a finite proof and abstraction decreases proof height, only a finite number of abstractions may be performed below a cell $pc \in \mathrm{E}(a)$, so that there must exist an $a' \in A_a$ such that $\delta_{a,a'}(pc)$ is persistently enabled. Moreover, it follows from proposition 5.19 that distinguished morphisms in A compose on persistently enabled cells. Our use of the term "persistently enabled" is justified by virtue of the following result.

Proposition 5.23 *For any pc persistently enabled from a, if $y \vdash_a pc$ then $\delta_{a,a'}(y) \vdash_{a'} \delta_{a,a'}(pc)$, for each $a' \in A_a$.*

Proof: Follows from definition of morphisms and persistence of events in y. ■

Proposition 5.24 *For any pc persistent from a, $\Psi_a(pc) = \{(\delta_{a,a'}(pc), u) \in a' \mid a' \in A_a\}$ is a directed set of events.*

Proof: For any two events (p_1c, u_1) and (p_2c, u_2) in $\Psi_a(pc)$ there exist $a_1, a_2 \in A_a$ such that $(p_ic, u_i) \in a_i$ and $p_ic = \delta_{a,a_i}(pc)$ for $i = 1,2$. By directedness of A, there is an $a_0 \in A_{a_1} \cap A_{a_2}$. Since distinguished morphisms compose on persistent cells, $\delta_{a,a_0}(pc) = \delta_{a_i,a_0}(p_ic)$ for $i = 1,2$. Hence, $(\delta_{a,a_0}(pc), u) \in a_0$ is an upper bound of (p_1c, u_1) and (p_2c, u_2) in $\Psi_a(pc)$. ■

As a consequence, whenever pc is persistent from a we may identify an event $\psi_a(pc) = \vee^i \Psi_a(pc)$. It is from these events that we construct a limit for A.

Proposition 5.25 *If $A \subseteq \mathcal{D}(M - M')$ is a directed set of algorithms then (using the above notation),*

$$\vee^i A = \{\psi_a(pc) \mid a \in A \ \& \ pc \text{ is persistent from } a\}$$

is a least upper bound for A in $\mathcal{D}(M - M')$.

Proof: $\vee^i A$ is certainly a set of events of $M - M'$. We show that it is a state, and a least upper bound for A as follows.

For each $a \in A$ define $\phi_a : \mathrm{E}(a) \to \mathrm{E}(\vee^i A)$ by $\phi_a(pc) = \sqcap_{a' \in A_a} \delta_{a,a'}(pc)$[7]. By proposition 5.21, for any pc persistent from a, $\psi_a(pc)$ has the form $(\phi_a(pc), u)$ for some u. We show

(1) For any pc persistently enabled from a, if $y \vdash_a pc$ then $\psi_a(y) \vdash_{\vee^i A} \phi_a(pc)$.

[7]We should really put $\phi_a(pc) = (\sqcap P)c$, where $P = \{p_1 \mid a' \in A_a \ \& \ p_1c = \delta_{a,a'}(pc)\}$. The abuse of notation is convenient.

(2) For any p_1c and p_2c persistent from a_1 and a_2, respectively, if $\phi_{a_1}(p_1c) = \phi_{a_2}(p_2c)$ then there exists $a' \in A_{a_1} \cap A_{a_2}$ such that $\delta_{a_1,a'}(p_1c) = \delta_{a_2,a'}(p_2c)$, and $\psi_{a_1}(p_1c) = \psi_{a_2}(p_2c)$.

For (1) we give details for the case when pc is has an output enabling. The reasoning for a query enabling is similar. We make essential use of proposition 5.23.

If $\{(p_jc_j, \textbf{output } v_j)\}_{j=1}^{l} \vdash_a pc$ then $\{(\delta_{a,a'}(p_jc_j), \textbf{output } v_j)\}_{j=1}^{l} \vdash_{a'} \delta_{a,a'}(pc)$ for each $a' \in A$. Hence $\delta_{a,a'}(pc) \in (\sqcup_{j=1}^{l} \delta_{a,a'}(p_jc_j))/_{\approx}$. Since $\sqcap$ is union and morphisms decrease classes, $\sqcap_{a' \in A_a} \delta_{a,a'}(pc) \in (\sqcap_{a' \in A_a} \sqcup_{j=1}^{l} \delta_{a,a'}(p_jc_j))/_{\approx}$. Therefore

$$\sqcap_{a' \in A_a} \sqcup_{j=1}^{l} \delta_{a,a'}(p_jc_j) \subseteq \sqcup_{j=1}^{l} \sqcap_{a' \in A_a} \delta_{a,a'}(p_jc_j).$$

The converse inclusion can be shown using directedness of A and the finiteness of the enabling. It follows that $\{(\phi_a(p_jc_j), \textbf{output } v_j)\}_{j=1}^{l} \vdash_{\vee^i A} \phi_a(pc)$, as required for (1).

For (2), suppose $\phi_{a_1}(p_1c) = \phi_{a_2}(p_2c)$. There must exist $a_1' \in A_{a_1}$ and $a_2' \in A_{a_2}$ such that $\delta_{a_1,a_1'}(p_1c) \Uparrow \delta_{a_2,a_2'}(p_2c)$. By directedness of A, there exists $a' \in A_{a_1'} \cap A_{a_2'}$. Let $p_i'c = \delta_{a_i,a'}(pc)$ for $i = 1, 2$; clearly $p_1' \Uparrow p_2'$. By the tree lemma, $p_1'c \ll_{a'}^{*} p_2'c$ or $p_2'c \ll_{a'}^{*} p_1'c$, so that, by (1), $\phi_{a'}(p_1'c) = \phi_{a'}(p_2'c)$ implies $p_1'c = p_2'c$.

Note that, by directedness, if pc is persistent from a then for any $a' \in A_a$, $\psi_a(pc) = \psi_{a'}(\delta_{a,a'}(pc))$ (and $\phi_a(pc) = \phi_{a'}(\delta_{a,a'}(pc))$). Therefore we have

$$\psi_{a_1}(p_1c) = \psi_{a'}(\delta_{a_1,a'}(p_1c)) = \psi_{a'}(\delta_{a_2,a'}(p_2c)) = \psi_{a_1}(p_2c),$$

as required for (2).

Safety and functionality of $\vee^i A$ are corollaries of (1) and (2), respectively, so that $\vee^i A$ is indeed a state.

To show that $\vee^i A$ is an upper bound of A, observe that ϕ_a is a morphism from a to $\vee^i A$, for each $a \in A$: it preserves all output events and output enablings, it weakens persistent queries, and it abstracts all other queries. The range of ϕ_a is indeed $\mathrm{E}(\vee^i A)$, as a corollary of (1).

Finally, to show that $\vee^i A$ is a least upper bound of A, let b be an upper bound of A. Define $\phi : \mathrm{E}(\vee^i A) - \mathrm{E}(b)$ by

$$\phi(p_0c) = \sqcap\{\delta_{a,b}(pc) \mid a \in A \;\&\; pc \in \mathrm{E}(a) \;\&\; p_0c = \phi_a(pc)\}.$$

It is easy to show that ϕ is a morphism, and that $\vee^i A \leq^i b$ by ϕ. ■

Example 5.26 Consider the sequence of algorithms $\texttt{id}_{\texttt{Nat}}{}^m \in \mathcal{D}(\textbf{Nat} - \textbf{Nat})$ for $m \geq 0$,

$$\texttt{id}_{\texttt{Nat}}{}^{m} = \begin{array}{l} \left\{ \boxed{[\emptyset]}\,\texttt{n}{=}\textbf{query} \cup_{k \leq m} \boxed{[\{\ \texttt{n}{=}k\ \}]} \right\} \\ \cup \ \left(\cup_{k \leq m} \left\{ \boxed{[\{\ \texttt{n}{=}k\ \}]}\,\texttt{n}{=}\textbf{output } k \right\} \right) \end{array}$$

This is an increasing sequence, and its lub is $\texttt{id}_{\texttt{Nat}}$, the identity algorithm on **Nat**. In this case, all filled cells are persistent and the distinguished morphisms never abstract. •

Example 5.27 Consider the sequence of algorithms $\mathtt{min}^m : \mathcal{D}(\mathbf{LNat} \times \mathbf{LNat} - \mathbf{LNat})$ for $m \geq 0$, defined by:

$$\mathtt{min}^m = \cup_{i=0}^{m} \mathtt{min}_i,$$

using the notation of Figure 12. This again is an increasing sequence, and its lub is $\mathtt{min}$. Again all filled cells persist and the distinguished morphisms do not abstract. •

Example 5.28 Recall the algorithms $\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3$ of examples 5.12 and 5.18. Since $\mathtt{a}_1 \leq^i \mathtt{a}_2 \leq^i \mathtt{a}_3$, they form a chain. All filled cells of $\mathtt{a}_2$ and $\mathtt{a}_3$ are persistent, but only the output cell of $\mathtt{a}_1$ is persistent. The lub of this chain, as expected, is $\mathtt{a}_3$. •

5.6 Countable DCDSs and Algebraicity

Following Berry and Curien, we now restrict attention to DCDSs having a countable set of cells and values. We show that if M and M' are countable then so is $M - M'$. Since all of our atomic DCDSs were countable, the countability restriction does not affect any of the results or definitions given so far. From here on we will work exclusively with first order countable DCDSs.

An element of a pre-order is *isolated* iff whenever it is below a least upper bound of a directed set it must be below some element of that set. Recall that a query is uniquely determined by its minimal elements; we refer to these as the query's *branches*. We say that an observable algorithm is *finite and finitely branching* (or *ffb*) iff it has a finite number of events, and each of its queries has a finite number of branches. We will show that the isolated algorithms are precisely the ffb algorithms, that there are countably many isolated algorithms in any countable DCDS, and that every algorithm is a lub of its isolated approximations, thus establishing that algorithms ordered by intensional strictness form an ω-algebraic pre-order.

Example 5.29 The identity algorithm on **Nat** is not ffb, since it has infinitely many output events and its query has infinitely many (mutually inconsistent) branches. The $\mathtt{min}$ algorithm is not ffb, because it has infinitely many events. The following algorithm of $\mathbf{Nat} \times \mathbf{Nat} - \mathbf{Bool}$ is not ffb, since it is finite but its query has infinitely many (equivalent) branches:

$$\left\{ \begin{array}{l} \boxed{[\langle \emptyset, \emptyset \rangle]}\, \mathtt{b}{=}\mathtt{query}\ \cup_{k \in \mathbb{N}} \boxed{\begin{array}{l} [\langle \{\ \mathtt{n}{=}k\ \},\ \ \emptyset \rangle] \\ \quad [\langle \emptyset\ \ ,\{\ \mathtt{n}{=}k\ \} \rangle] \end{array}} \\ \cup_{k \in \mathbb{N}} \boxed{\begin{array}{l} [\langle \{\ \mathtt{n}{=}k\ \},\ \ \emptyset \rangle] \\ \quad [\langle \emptyset\ \ ,\{\ \mathtt{n}{=}k\ \} \rangle] \end{array}}\, \mathtt{b}{=}\mathbf{output}\ \mathtt{tt} \end{array} \right\}$$

In examples 5.26 and 5.27 each of the $\mathtt{id_{Nat}}^m$ and $\mathtt{min}^m$ algorithms $(m \geq 0)$ is ffb. •

Proposition 5.30 *A first order countable DCDS has countably many ffb algorithms.*

Proof: There are countably many events in an atomic DCDS, hence countably many finite sets of events in the representation of a first order DCDS. It follows that there are countably many finitely branching queries and countably many finite classes, and hence that the ffb algorithms are countable. ▪

Proposition 5.31 *The ffb approximations to an algorithm form a directed set.*

Proof: Let $\hat{a}$ be an algorithm, and let a_1 and a_2 be two ffb approximations to $\hat{a}$. By proposition 5.5, $a' = \delta_{a_1,\hat{a}}(a_1) \cup \delta_{a_2,\hat{a}}(a_2)$ is an algorithm; it is an approximation to $\hat{a}$ by inclusion, and it has a finite number of events. a' is not necessarily finitely branching, but we may derive from it an ffb algorithm a such that $a_1, a_2 \leq^i a \leq^i a' \leq^i \hat{a}$. The key idea is to perform the following operation (inductively): if q' is a query of a' that weakens the queries q_1 of a_1 and q_2 of a_2, then replace q' in a by a query q that contains only those branches of q' that are below branches of q_1 or q_2. q will necessarily be finitely branching, since branches are themselves finite sets of events. Similarly if q' weakens a query from either a_1 or a_2. Note that replacement of queries of a' by finitely branching subsets may lead to splitting of equivalence classes, which needs to be handled by the construction of a (or, alternatively, some extra elements of q' may be retained so as to prevent splitting). It is straightforward to show that an algorithm a so obtained satisfies the required properties. ■

Proposition 5.32 *Every algorithm is the lub of its ffb approximations.*

Proof: First, we fix, for every query q, an enumeration of its branches, and we define a sequence of finite queries $\{q_n\}_{n\geq 1}$, such that q_n is the upwards closure of the first n branches of q. Thus the sequence is decreasing with respect to $\sqsubseteq$, and we have $q = \sqcap_{n\geq 0} q_n$.

For any algorithm a, given an enumeration of queries as above, we define a sequence of finitely branching approximations to a.

$$\text{fb}_n(a) = \{(p'c, \text{fb}_n(u)) \in \text{E}_{M-M'} \mid (pc, u) \in a \;\&\; p'c \in \text{fb}_{a,n}(pc)\}$$

where, for $n \geq 1$, the functions $\text{fb}_n : \text{V}_{M-M'} \rightarrow \text{V}_{M-M'}$ are defined by

$$\text{fb}_n(\textbf{query } q) = \textbf{query } q_n \qquad \text{fb}_n(\textbf{output } v) = \textbf{output } v$$

and the functions $\text{fb}_{a,n} : \text{E}(a) \rightarrow \mathcal{P}(\text{C}_{M-M'})$ are inductively defined by

$$\text{fb}_{a,n}(pc) = \{p'c \mid y \vdash_a pc \;\&\; \text{fb}_{a,n}(y) \vdash_{M-M'} p'c \;\&\; p' \sqsubseteq p\}$$

(where $\text{fb}_{a,n}(y) = \{(p'_j c_j, \text{fb}_n(u_j)) \mid \exists (p_j c_j, u_j) \in y \;.\; p'_j c_j \in \text{fb}_{a,n}(p_j c_j)\}$). Note that $\text{fb}_{a,n}(pc)$ may be empty.

For any algorithm a, we define a sequence of finite depth approximations to a: $(a)_0 = \emptyset$ and for each $n \geq 0$, $(a)_{n+1} = \{(pc, u) \in a \mid pc \in \text{E}((a)_n)\}$.

Now we combine these two ideas: for each n, $(\text{fb}_n(a))_n$ is finite and finitely branching. It is straightforward to show that the sequence $\{(\text{fb}_n(a))_n\}_{n\geq 0}$ is an increasing chain of ffb approximations to a whose lub is a. ■

Proposition 5.33 *The isolated elements of $(\mathcal{D}(M - M'), \leq^i)$ are the ffb algorithms.*

Proof:

- We show that every ffb algorithm is isolated. Let $\hat{a}$ be the lub of a set A of algorithms, and let a be an ffb algorithm such that $a \leq^i \hat{a}$.

  For each $pc \in \text{E}(a)$, $\delta_{a,\hat{a}}(pc) \in \text{E}(\hat{a})$, so that there exist $a' \in A$ and $p'c \in \text{E}(a')$ such that $\phi_{a'}(p'c) = \delta_{a,\hat{a}}(pc)$. But $\delta_{a,\hat{a}}(pc) \sqsubseteq p$, and p is finitely branching; hence, by directedness, we can choose a' and $p'c$ such that additionally $p' \sqsubseteq p$.

Now, if $a' \in A$ has a suitable cell $p/c \in \mathrm{E}(a')$ that satisfies the above, then so does every $a'' \in A_{a'}$; therefore, since a has only finitely many events, and A is directed, there exists an $a'' \in A$ that satisfies the above requirements for all $pc \in \mathrm{E}(a)$ simultaneously. But now it is easy to show that $a \leq^i a''$, and therefore a is isolated.

- We show that every isolated algorithm is ffb. Let $\hat{a}$ be an isolated algorithm. Since $\hat{a}$ is the lub of the directed set of its ffb approximations, there must exist some $a_0 \leq^i \hat{a}$, an ffb approximation to $\hat{a}$, such that $\hat{a} \leq^i a_0$. Without loss of generality assume that both a_0 and $\hat{a}$ are observable, and it follows by anti-symmetry that $\hat{a} = a_0$ is ffb. ■

Corollary 5.34 *$(\mathcal{D}(M \to M'), \leq^i)$ is a directed-complete and ω-algebraic pre-order, and its isolated elements are the ffb algorithms.*

The fact that the intensional strictness ordering enjoys these order-theoretic properties enables us to adapt the usual semantic account of recursively defined objects to the algorithmic setting. It is well known that every continuous function on a directed-complete partial order has a (unique) least fixed point, which can be constructed explicitly as the limit of a chain of iterates. A similar result holds for a directed-complete pre-order, except that the least fixed point is only unique up to equivalence. While we do not intend to explore recursion deeply in this paper, we give a simple example to show that parallel algorithms may be defined recursively.

Example 5.35 Let $inc : C_{\mathbf{LNat}} \to C_{\mathbf{LNat}}$ be the function which adds 1 to each cell index; this extends to the queries and classes involved in the construction of $\mathbf{LNat} \times \mathbf{LNat} \to \mathbf{LNat}$ in the obvious way, so that for example

$$inc(\langle S^n(\bot), S^n(0)\rangle) \cup \langle S(\bot), S(\bot)\rangle = \langle S^{n+1}(\bot), S^{n+1}(0)\rangle .$$

Let $\Phi : \mathcal{D}(\mathbf{LNat} \times \mathbf{LNat} \to \mathbf{LNat}) \to \mathcal{D}(\mathbf{LNat} \times \mathbf{LNat} \to \mathbf{LNat})$ be the function defined by

$$\Phi(a) = \mathtt{min}_0 \cup inc(a),$$

referring to Figure 12 for the definition of $\mathtt{min}_0$. Clearly, Φ is continuous and has a least fixed point $\bigvee^i_{n \geq 0} \Phi^n(\emptyset) = \mathtt{min}$. This example formalizes the intuition that $\mathtt{min}$ is obtained by "iterating" a parallel-or like kernel. ●

5.7 Monotonicity and Continuity

Proposition 3.17 states that currying and uncurrying are isomorphisms with respect to the set inclusion ordering. We now show further that they are order-isomorphisms with respect to the intensional strictness order.

Proposition 5.36 *Currying and uncurrying are monotone and continuous with respect to the intensional strictness order.*

Proof: Observe that, for all $a, a' \in \mathcal{D}(M_1 \times M_2 \to M')$, if $a \leq^i a'$ by f then $\mathrm{curry}(a) \leq^i \mathrm{curry}(a')$ by the morphism $\mathrm{curry} \circ f \circ \mathrm{uncurry}$. ■

We next show that application is monotone with respect to $\leq^i$. Let $a, a' \in \mathcal{D}(M \to M')$ with $a \leq^i a'$ by $f : \mathrm{E}(a) \to \mathrm{E}(a')$, and $x, x' \in \mathcal{D}(M)$ with $x \leq^i x'$. We must find a morphism $h : \mathrm{E}(a \cdot x) \to \mathrm{E}(a' \cdot x')$. To construct such a morphism, we need to focus on the events of $a \cdot x$ whose source events in a correspond (under f) to events in a' which project by x'; each such event of $a \cdot x$ will thus determine an event of $a' \cdot x'$. We call these the *f-preserved* events of $a \cdot x$.

Definition 5.37 All output events of $a \cdot x$ are f-preserved. A query event $(pc, \mathbf{query}\ q) \in a \cdot x$ is f-preserved if f weakens its source event $(\mathrm{source}_{a,x}(pc), \mathbf{query}\ \hat{q}) \in a$, with $q = \pi_x(\hat{q})$, into $(f(\mathrm{source}_{a,x}(pc)), \mathbf{query}\ \hat{q}') \in a'$, with $\hat{q}' \sqsubseteq \hat{q}$, and $\emptyset \notin \pi_{x'}(\hat{q}')$. Cells filled in f-preserved events of $a \cdot x$ are also said to be f-preserved.

Given $pc \in \mathrm{E}(a \cdot x)$, define $P_1(pc)$ to be the set of $\ll^+_{a \cdot x}$-maximal f-preserved cells below pc,

$$P_1(pc) = \{rd \mid \begin{array}{l} rd \ll^+_{a\cdot x} pc \ \&\ rd \text{ is } f\text{-preserved } \& \\ \forall r_1 d \ .\ rd \ll^+_{a\cdot x} r_1 d \ll^+_{a\cdot x} pc \Rightarrow r_1 d \text{ is not } f\text{-preserved} \end{array} \}.$$

Define $h : \mathrm{E}(a \cdot x) \to \mathrm{E}(a' \cdot x')$ by

$$h(pc) = \begin{cases} p'c & \text{if } \{(p_j c_j, \mathbf{output}\ v_j)\}_{j=1}^{l} \vdash_{a\cdot x} pc \\ & \quad \&\ \{(h(p_j c_j), \mathbf{output}\ v_j)\}_{j=1}^{l} \vdash_{a'\cdot x'} p'c \ \&\ p' \sqsubseteq p \\ & \\ p'c & \text{if } (p_1 c, \mathbf{query}\ q) \vdash_{a\cdot x} pc \ \&\ p_1 c \text{ is } f\text{-preserved} \\ & \quad \&\ (h(p_1 c), \mathbf{query}\ q') \vdash_{a'\cdot x'} p'c \ \&\ p' \sqsubseteq p \\ & \\ h(p_1 c) & \text{if } (p_1 c, \mathbf{query}\ q) \vdash_{a\cdot x} pc \ \&\ p_1 c \text{ is not } f\text{-preserved} \end{cases}$$

For $pc \in \mathrm{E}(a \cdot x)$ let $P_2(pc) = \{r'd \mid r'd \ll_{a'\cdot x'} h(pc)\}$ be the set of cells in $a' \cdot x'$ that enables $h(pc)$. ●

Next we show some properties of h, which establish that h is a morphism.

Proposition 5.38 *For $pc \in \mathrm{E}(a \cdot x)$,*

(1) The function h is well defined on pc.

(2) h maps the maximal f-preserved cells below pc onto the enabling of $h(pc)$ in $a' \cdot x'$:

$$\{h(rd) \mid rd \in P_1(pc)\} = P_2(pc).$$

(3) If pc is f-preserved then $h(pc) \in \mathrm{F}(a' \cdot x')$ and $h(pc) \in \pi_x(f(\mathrm{source}_{a,x}(pc)))/_{\approx}$.

(4) If $(pc, \mathbf{output}\ v) \in a \cdot x$ then $(h(pc), \mathbf{output}\ v) \in a' \cdot x'$, and if $(pc, \mathbf{query}\ q) \in a \cdot x$ is f-preserved then $(h(pc), \mathbf{query}\ q') \in a' \cdot x'$ for some $q' \sqsubseteq q$.

Proof: By induction on pc.

(1),(2) Consider the unique enabling of pc in $a \cdot x$.

If $\{(p_j c_j, \mathbf{output}\ v_j)\}_{j=1}^{l} \vdash_{a\cdot x} pc$, then, by induction hypothesis (1) and (4), for any $1 \leq j \leq l$, h is defined on $p_j c_j$ and $(h(p_j c_j), \mathbf{output}\ v_j) \in a' \cdot x'$. Therefore $h(pc)$ is the unique $p'c$ such that $\{(h(p_j c_j), \mathbf{output}\ v_j)\}_{j=1}^{l} \vdash_{a'\cdot x'} p'c$ and $p' \sqsubseteq p$. Moreover, $P_1(pc) = \{p_j c_j\}_{j=1}^{l}$, so (2) follows.

If $(p_1 c, \mathbf{query}\ q) \vdash_{a\cdot x} pc$ then, by induction hypothesis (1), $h(p_1 c)$ is well defined. If $(p_1 c, \mathbf{query}\ q)$ is not f-preserved, then $h(pc)$ is taken to be $h(p_1 c)$; thus $P_1(pc) = P_1(p_1 c)$ and $P_2(pc) = P_2(p_1 c)$. Property (2) for pc follows by induction hypothesis (2) for $p_1 c$.

If, on the other hand, $(p_1 c, \mathbf{query}\ q)$ is f-preserved, then, by induction hypothesis (4), $(h(p_1 c), \mathbf{query}\ q') \in a' \cdot x'$ for some $q' \sqsubseteq q$. Then $h(pc)$ is defined to be the (uniquely determined) $p'c$ such that $(h(p_1 c), \mathbf{query}\ \pi_{x'}(\hat{q}')) \vdash_{a'\cdot x'} p'c$ and $p' \sqsubseteq p$. Moreover, $P_1(pc) = \{p_1 c\}$ and $P_2(pc) = \{h(p_1 c)\}$, so (2) holds.

(3) Assume that pc is filled in $a \cdot x$. There exists $p_0c \ll^*_{a\cdot x} pc$ such that $P_1(pc) \vdash_{a\cdot x} p_0c$. By 4.8 and 4.10 we have

$$\text{source}_{a,x}(P_1(pc)) \vdash_a \text{root}_{a,x}(p_0c) \ll^*_a \text{source}_{a,x}(pc).$$

Since the cells in $P_1(pc)$ are f-preserved, by applying f we get:

$$f(\text{source}_{a,x}(P_1(pc))) \vdash_{a'} f(\text{root}_{a,x}(p_0c)) \ll^*_{a'} f(\text{source}_{a,x}(pc)).$$

For $\bar{x} \in \pi_x(\hat{r})$, we write $\pi_{x,\bar{x}}(\hat{r}d)$ for the cell rd such that $r \in \pi_x(\hat{r})/_\approx$ and $\bar{x} \in r$; r is uniquely determined. We also use the obvious extension to a set of cells or events.

Choose any $\bar{x} \in p$. Clearly, $x \in \pi_{x'}(f(\text{source}_{a,x}(pc)))$. Now, since the cells in $P_1(pc)$ are f-preserved, we have.

$$\pi_{x',\bar{x}}(f(\text{source}_{a,x}(P_1(pc)))) \vdash_{a'\cdot x'} \pi_{x',\bar{x}}(f(\text{root}_{a,x}(p_0c))) \ll^*_{a'\cdot x'} \pi_{x',\bar{x}}(f(\text{source}_{a,x}(pc))).$$

But if the query chain for $\pi_{x',\bar{x}}(f(\text{root}_{a,x}(p_0c))) \ll^*_{a'\cdot x'} \pi_{x',\bar{x}}(f(\text{source}_{a,x}(pc)))$ is of non-zero length, then some cell p_1c such that $p_0c \ll^*_{a\cdot x} p_1c \ll^+_{a\cdot x} pc$ is f-preserved, contradicting the definition of $P_1(pc)$. Therefore $\pi_{x',\bar{x}}(f(\text{root}_{a,x}(p_0c))) = \pi_{x',\bar{x}}(f(\text{source}_{a,x}(pc)))$, and

$$\pi_{x',\bar{x}}(f(\text{source}_{a,x}(P_1(pc)))) \vdash_{a'\cdot x'} \pi_{x',\bar{x}}(f(\text{source}_{a,x}(pc))).$$

By induction hypothesis (3) and (2), $P_2(pc) \vdash_{a'\cdot x'} \pi_{x',\bar{x}}(f(\text{source}_{a,x}(pc)))$, while, by definition of $P_2(pc)$, $P_2(pc) \vdash_{a'\cdot x'} h(pc)$. But $h(pc)$ and $\pi_{x',\bar{x}}(f(\text{source}_{a,x}(pc)))$ are upper bounded by p, so that, by proposition 3.11, they must be equal.

(4) Follows immediately from (3). Note that if $x \subseteq x'$ and $q' \sqsubseteq q$ then $\pi_{x'}(q') \sqsubseteq \pi_x(q)$. ■

Corollary 5.39 *Application is monotone in both arguments: if $a \leq^i a'$ by f and $x \leq^i x'$ then $a \cdot x \leq^i a' \cdot x'$ by h, as defined above.*

Definition 5.40 The *input-output approximation* order $\leq^e_M$ on $\mathcal{D}(M)$ is defined by induction on M as follows.

For an atomic DCDS M let $\leq^e_M$ be set inclusion.
For a product $M_1 \times M_2$ let $\leq^e_{M_1 \times M_2}$ be defined componentwise.
For an arrow type $M - M'$ let $a \leq^e a'$ iff $\forall x \in \mathcal{D}(M)\ .\ a \cdot x \leq^e a' \cdot x$. •

Input-output approximation orders algorithms by the pointwise order on their input-output functions. It is a pre-order, and two algorithms are input-output equivalent whenever they compute the same function. For instance, the or algorithms in figure 15 fall into four equivalence classes, corresponding to the functions *sor*, *lor*, *ror* and *por*, and the diagram collapses to the pointwise ordering on these functions.

Proposition 5.41 *For a first order DCDS M, $\leq^i_M$ is contained in $\leq^e_M$.*

Proof: An easy corollary of monotonicity of application with respect to $\leq^i$. ■

Thus, whenever $a \leq^i a'$ we also have $a \leq^e a'$. The converse fails, because the input-output approximation order is not properly sensitive to computation strategy. For instance, **lsor** $\leq^e$ **rsor** but these two algorithms have incompatible computation strategies and are incomparable under the intensional order. Putting this result together with the earlier remark that intensional strictness properly includes set inclusion (proposition 5.6), we may summarize by saying that the intensional order is strictly coarser than set inclusion and strictly finer than input-output approximation.

Next we prove that with the intensional ordering application is indeed continuous.

Proposition 5.42 *For any $x \in \mathcal{D}(M)$ and non-empty $Q \subseteq \mathcal{P}_s(\mathcal{F}_{\text{fin}}(\text{rep}(M - M')))$,*

$$\pi_x(\sqcap Q) = \sqcap\{\pi_x(q) \neq \emptyset \mid q \in Q\}.$$

where the right hand side is to be taken as the empty set in case the glb is undefined, i.e., $\pi_x(q) = \emptyset$ for every $q \in Q$.

Proof: Immediate: recall that the glb is just set union. ∎

Proposition 5.43 *Application is continuous in both arguments: if A is a directed set of algorithms of $M - M'$ and X is a directed set of states of M, then $(\bigvee^i A)\cdot(\bigvee^i X)$ and $\bigvee^i\{a \cdot x \mid a \in A \;\&\; x \in X\}$ are equivalent.*

Proof: Let $Z = \{a \cdot x \mid a \in A \;\&\; x \in X\}$. This is easily seen to be a directed set of states of M', by monotonicity of application. Let $\hat{a}$, $\hat{x}$ and $\hat{z}$ be the lubs of A, X and Z, respectively. By monotonicity of application, $\hat{z} \leq^i \hat{a} \cdot \hat{x}$. We show that $\hat{a} \cdot \hat{x} \subseteq \hat{z}$. We use notations and definitions as in the proof of directed-completeness (proposition 5.25), and indicate A, X or Z to select the appropriate context. We also use the notation $\pi_{x,\bar{x}}(pc)$ as in proposition 5.38 for the cell $p'c$ such that $p' \in \pi_x(p)/_{\approx}$ and $\bar{x} \in p'$ (provided $\bar{x} \in \pi_x(p)$).

We prove by induction on $\hat{p}'c \in \mathrm{F}(\hat{a} \cdot \hat{x})$ that:

If $(\hat{p}'c, \hat{u}') \in \hat{a} \cdot \hat{x}$ and $x \in \hat{p}'$ then there exist $a \in A$, pc persistent from a in A, and $x \in X$, such that

(1) $\phi_a^A(pc) = \text{source}_{\hat{a},\hat{x}}(\hat{p}'c)$.

(2) $\pi_{x,x}(pc)$ is persistently enabled from $a \cdot x$ in Z, and $\hat{p}'c = \phi_{a\cdot x}^Z(\pi_{x,\bar{x}}(pc))$.

(3) $\pi_{x,\bar{x}}(pc)$ is persistent from $a \cdot x$ in Z, and $(\hat{p}'c, \hat{u}') = \psi_{a\cdot x}^Z(\pi_{x,\bar{x}}(pc))$.

(4) $(\hat{p}'c, \hat{u}') \in \hat{z}$.

Note that if a, pc and x satisfy the above, then so do any $a' \in A_a$, $\delta_{a,a'}(pc)$ and $x' \in X_x$; we rely on this to make successive assumptions about a and x that can be met by increasing a and x without invalidating any of the preceding conclusions.

Let $(\hat{p}'c, \hat{u}') \in \hat{a} \cdot \hat{x}$, with $x \in \hat{p}'$. Then $(\text{source}_{\hat{a}\cdot\hat{x}}(\hat{p}'c), \hat{u}) \in \hat{a}$ with $\hat{u}' = \pi_{\hat{x}}(\hat{u})$. By definition of $\hat{a}$ (proposition 5.25), there exist $a \in A$ and pc persistent from a in A such that $\psi_a^A(pc) = (\text{source}_{\hat{a},\hat{x}}(\hat{p}'c), \hat{u})$, and (1) holds.

Since $\bar{x} \in \hat{p}'$, there must exist some finite $x_0 \subsetneq \hat{x}$ such that $\langle x_0, \bar{x}\rangle \in \phi_a^A(pc)$. By algebraicity, there exists $x \in X$ such that $x_0 \subseteq x$. Without loss of generality, we can choose a so that $\langle x_0, \bar{x}\rangle \in p$, and $\bar{x} \in \pi_x(p)$.

Let $\hat{y}' \vdash_{\hat{a}\cdot\hat{x}} \hat{p}'c$. Therefore

- $\text{source}_{\hat{a},\hat{x}}(\hat{y}') \vdash_{\hat{a}} \text{root}_{\hat{a},\hat{x}}(\hat{p}'c) \ll_{\hat{a}}^{*} \text{source}_{\hat{a},\hat{x}}(\hat{p}'c)$.

Now, by the induction hypothesis, for any $(\hat{p}'_j c_j, \hat{u}'_j) \in \hat{y}'$ there exist appropriate $a_j \in A$, $p_j c_j$ persistent from a_j in A, and $x_j \in X$ that verify the induction hypothesis for $\hat{p}'_j c_j$. Since $\hat{y}'$ is finite, we can choose a and x larger than each a_j and x_j, respectively, so that a and x verify the induction hypothesis for each $\hat{p}'_j c_j$. Therefore, there exists a set $y \subseteq a$ of persistent events with $\psi^A_a(y) = \text{source}_{\hat{a},\hat{x}}(\hat{y}')$, such that $y \vdash_a p_0 c \ll^*_a pc$, where $\phi^A_a(p_0 c) = \text{root}_{\hat{a},\hat{x}}(\hat{p}'c)$. The enabling $\psi^A_a(y) \vdash_{\hat{a}} \text{root}_{\hat{a},\hat{x}}(\hat{p}'c)$ is not fully satisfied by $\hat{x}$, while each of the enablings in the (finite) query chain $\text{root}_{\hat{a},\hat{x}}(\hat{p}'c) \ll^*_{\hat{a}} \text{source}_{\hat{a},\hat{x}}(\hat{p}'c)$ is fully satisfied by $\hat{x}$. It is therefore possible to choose x sufficiently large so that it projects the enabling $y \vdash_a p_0 c$ and fully satisfies all the enablings in the chain $p_0 c \ll^*_a pc$; note that $y \vdash_a pc$ may not be fully satisfied by $x \subseteq \hat{x}$. We thus obtain $\pi_{x,\bar{x}}(y) \vdash_{a \cdot x} \pi_{x,\bar{x}}(pc)$, and by induction hypothesis (3), $\pi_{x,\bar{x}}(pc)$ is persistently enabled from $a \cdot x$ in Z. Moreover, from (3) we have

$$\hat{y}' = \psi^Z_{a \cdot x}(\pi_{x,\bar{x}}(y)) \vdash_{\hat{z}} \phi^Z_{a \cdot x}(\pi_{x,\bar{x}}(pc)),$$

and since $\hat{p}'c$ and $\phi^Z_{a \cdot x}(\pi_{x,\bar{x}}(pc))$ are consistent (both contain $\bar{x}$), then, by proposition 3.11, they must be equal, and we have established (2).

If $\hat{u}' =$ **output** v choose a so that $(pc, \textbf{output } v) \in a$. Then $(\pi_{x,\bar{x}}(pc), \textbf{output } v) \in a \cdot x$, $\pi_{x,\bar{x}}(pc)$ is clearly persistent from $a \cdot x$ in Z, and $(\hat{p}'c, \textbf{output } v) = \psi^Z_{a \cdot x}(\pi_{x,\bar{x}}(pc))$, establishing (3) for the output case.

If $\hat{u}' =$ **query** $\hat{q}'$ things are somewhat more complicated. First, note that $\hat{z}$ is also the lub of $Z' = \{a' \cdot x' \mid a' \in A_a \;\&\; x' \in X_x\}$, so that, without loss of generality, we may assume that if $a \cdot x \leq^i a' \cdot x'$ in Z then $a \leq^i a'$ and $x \subseteq x'$.

We choose x so that $\pi_x(q) \neq \emptyset$, where q is the query that fills pc in a. Since pc is persistent from a in A, for every $a' \in A_a$ and $a'' \in A_{a'}$, $\delta_{a',a''}$ does not abstract at $\delta_{a,a'}(pc)$. But since projection by $\hat{x}$ does not fully satisfy at $\phi^A_a(pc)$, then for every $a' \in A_a$ and $x' \in X_x$, $\delta_{a,a'}(pc)$ is projected by x', but is not fully satisfied, so that $\delta_{a' \cdot x', a'' \cdot x''}$ does not abstract at $\delta_{a \cdot x, a' \cdot x'}(\pi_{x,\bar{x}}(pc))$ for $a \leq^i a' \leq^i a''$ and $x \subseteq x' \subseteq x''$, and $\pi_{x,\bar{x}}(pc)$ is persistent from $a \cdot x$.

It remains to show that $\hat{p}'c$ is filled with the same queries in both $\hat{a} \cdot \hat{x}$ and $\hat{z}$, *i.e.*, that

$$\pi_{\hat{x}}(\sqcap Q^A_a(pc)) = \sqcap\{\pi_{x'}(q) \mid q \in Q^A_a(pc) \;\&\; x' \in X_x\},$$

where $Q^A_a(pc) = \{q' \mid a' \in A_a \;\&\; (\delta_{a,a'}(pc), \textbf{query } q') \in a'\}$. But by proposition 5.42,

$$\pi_{\hat{x}}(\sqcap Q^A_a(pc)) = \sqcap\{\pi_{\hat{x}}(q) \mid q \in Q^A_a(pc)\}$$

and the rest follows from the directedness of X and the finiteness of query elements.

We have established (3), and (4) is an immediate consequence, thereby completing the proof by induction. Finally, from (4) we conclude that $\hat{a} \cdot \hat{x} \subseteq \hat{z}$. ∎

Corollary 5.44 *The input-output function of every algorithm in $\mathcal{D}(M \to M')$ is a continuous function from $(\mathcal{D}(M), \leq^i_M)$ to $(\mathcal{D}(M'), \leq^i_{M'})$.*

Example 5.45 The input-output function of the algorithm $\texttt{min}$ is *min*. For each $n \geq 0$ we have

$$\texttt{min} \cdot \langle S^n(\bot), S^n(\bot)\rangle = \texttt{min}^n \cdot \langle S^\omega(\bot), S^\omega(\bot)\rangle = S^n(\bot).$$

Hence,

$$\texttt{min} \cdot \langle S^\omega(\bot), S^\omega(\bot)\rangle = \bigvee^i_{n \geq 0} \texttt{min} \cdot \langle S^n(\bot), S^n(\bot)\rangle = \bigvee^i_{n \geq 0} \texttt{min}^n \cdot \langle S^\omega(\bot), S^\omega(\bot)\rangle = S^\omega(\bot).$$

●

6 Future Research Directions

We regard this paper as a first step towards a general theory of determinate parallelism. We have developed intuitively appealing notions of parallel algorithms, the input-output function of an algorithm, application and currying of algorithms. We have introduced an intensional strictness ordering on first order algorithms that appears to be a natural generalization of the usual extensional order on continuous functions, in the sense that whenever $a \leq^i a'$ the input-output function of a approximates the input-output function of a' extensionally. The class of first order parallel algorithms is closed under currying and uncurrying, and contains many interesting algorithms for non-sequential functions; it is already significantly different from the class of first order sequential algorithms.

We have tried to stay close in spirit to the foundational work of Berry and Curien, and have to a large extent emulated their development: beginning with algorithms, defining application, then constructing input-output functions. As we have pointed out, there is a simple embedding of their (first order) sequential algorithms into our parallel algorithms that preserves the function computed by an algorithm. Sequential algorithms correspond to parallel algorithms with trivial parallelism: each query involves a single cell. However, the generalization to the concurrent setting has forced us to depart from set inclusion as the underlying order and to adopt a new order with respect to which application is well behaved. It is interesting to look back and determine to what extent the phenomena of abstraction and weakening, upon which our ordering is based, occur in the Berry-Curien model. Weakening in the sequential setting is reduced to set inclusion, but abstraction is not. Our intensional strictness pre-order induces a pre-order on the Berry-Curien model, still (strictly) coarser than set inclusion and (strictly) finer than input-output approximation. All of this is not surprising: a conjecture we would like to substantiate is that the relationship between the set inclusion and intensional strictness orderings on algorithms is analogous to the relationship between the stable and the pointwise orderings on functions.

One of the key features in our model is the use of queries instead of valof commands. We regard queries as generalized sequentiality indices, perhaps better called *computation indices*, since they are applicable to the parallel setting. We can characterize the class of parallel algorithms which have a stable input-output function, in Berry's sense, in terms of their computation indices: an algorithm computes a stable function iff the branches of each of its observable queries are mutually inconsistent, or, equivalently, iff each of its observable classes has a least element. We intend to develop these ideas and to investigate the new notions of stability and sequentiality obtained by employing intensional strictness as the underlying order on states. We conjecture that (in line with remarks made earlier) the curried parallel-or *cpor* will turn out to be sequential in this new sense, since its input type has a single cell, while the uncurried *por* remains parallel (as it should). This example also suggests that we should regard as "fully" sequential only those algorithms which remain sequential under currying and uncurrying.

The intensional strictness order seems to be a natural outcome of our definition of application, which in turn seems quite intuitive. This new ordering, however, only makes application well behaved for first order DCDSs. Our proofs of monotonicity and continuity for application do not extend to the higher order case, where intensional strictness on the representation departs from set inclusion. A reason for the failure at higher order types is that addition of non-observable query events to an algorithm no longer constitutes an increase in the information content of the algorithm (as shown in example 5.8) ; therefore, a higher order algorithm is not able to build incrementally an internal representation of an argument which itself is an algorithm simply by issuing queries about the query structure of that argument. A modification is needed to the way in which the internal representation is built; one possibility is to change the *values* of $M \to M'$ to be *trees* whose internal nodes correspond to queries, and whose leaves correspond to output events.

In addition to our present limitation to first order types, we do not have yet a satisfactory notion of algorithm composition. This has not prevented us from defining application and input-output functions, but of course without composition we cannot use our algorithms to define a category. Perhaps it is worth remarking that Berry and Curien [BC82, Cur86] present application and input-output functions before constructing a suitable composition for sequential algorithms, and even in the sequential case the definition of composition is given indirectly, by means of "abstract algorithms". It may not then be surprising that we have found it difficult to find a suitable parallel generalization.

We have used representation and base DCDSs in our formulation of parallel algorithms so as to be able to express curried algorithms. While this rather complicates the internal structure of algorithms, it does facilitate the definition of currying and uncurrying as operations on algorithms. Nevertheless, the use of rep and base seems to be at least partially responsible for our difficulty in formulating a notion of composition for algorithms, and we would like to explore alternative ways to define algorithms. For instance, we might try to define $M \to M'$ using events of form (pc, u) with p a class over M, c a cell of M', and u either an output over M' or a query over M, but requiring that consistent inputs lead to *consistent* output commands, instead of the current requirement that consistent inputs lead to the *same* output command. In order to allow this we would need to endow CDSs with an order structure so that we can define what it means for inputs or outputs to be consistent. In a related paper [BG] we explore properties of a generalized form of CDS in which cells and values are equipped with partial orders, with appropriate modifications to the notion of state.

Much more remains to be done. Ultimately we would like to construct a model of parallel algorithms that makes sense at all types and yields a cartesian closed category, so as to provide an intensional semantics for the λ-calculus. In such a semantics the denotation of a term would reflect accurately the *efficiency* with which it computes its results, or other intensional aspects. This should allow us to formalize the sense in which (for example) our `min` algorithm computes the *min* function in complexity $\mathtt{O}(min(m, n))$.

We can also formulate an intuitively natural ordering that reflects the *degree of parallelism* (or *eagerness*) exhibited by an algorithm, so that, for instance, `psor` is indeed the most parallel of the algorithms for *sor*, while the two sequential algorithms `lsor` and `rsor` are local minima for this ordering. There appears to be a natural hierarchy among parallel algorithms, based on our notion of degree of parallelism. We plan to investigate this parallelism order and the structure of this hierarchy, in the hope that our ideas may help in assessing the relative expressive power of various parallel primitives.

7 Acknowledgements

We thank the anonymous referees, who made helpful suggestions that led to improvements in the presentation of the paper.

References

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

[BC85] G. Berry and P.-L. Curien. Theory and practice of sequential algorithms: the kernel of the applicative language CDS0. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, chapter 2, pages 35–87. Cambridge University Press, 1985.

[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 λ-calculi. In *Proc. 5th Coll. on Automata, Languages and Programming*, number 62 in Lecture Notes in Computer Science, pages 72–89, Berlin, New-York, July 1978. Springer-Verlag.

[BG] S. Brookes and S. Geva. Continuous functions and parallel algorithms on concrete data structures. In *Mathematical Foundations of Programming Semantics, 7th International Conference, Carnegie Mellon University, Pittsburgh, March 1991*, Lecture Notes in Computer Science. Springer-Verlag.

[BG90] S. Brookes and S. Geva. Towards a theory of parallel algorithms on concrete data structures. In *Semantics for Concurrency, Leicester 1990*, pages 116–136. Springer-Verlag, 1990.

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

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

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

[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.

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

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

[Saz75] V. Yu. Sazonov. Sequentially and parallelly computable functionals. In *Proc. Symp. on Lambda-Calculus and Computer Science Theory*, number 37 in Lecture Notes in Computer Science. Springer-Verlag, 1975.

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

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