Concrete data structures and functional parallel programming

Gaétan Hains\textsuperscript{a,*}, Frédéric Loulergue\textsuperscript{a}, John Mullins\textsuperscript{b}

\textsuperscript{a}LIFO, Université d'Orléans, BP 6759, 45067 Orléans Cedex 2, France
\textsuperscript{b}DGEGI, Ecole Polytechnique, C.P. 6079, Succ. Centre-Ville, Montréal (Québec), Canada H3C 3A7

Received August 1997; revised November 1999
Communicated by P.-L. Curien

Abstract

A framework is presented for designing parallel programming languages whose semantics is functional and where communications are explicit. To this end, Brookes and Geva’s generalized concrete data structures are specialized with a notion of explicit data layout to yield a CCC of distributed structures called arrays. Arrays’ symmetric replicated structures, suggested by the data-parallel SPMD paradigm, are found to be incompatible with sum types. We then outline a functional language with explicitly distributed (monomorphic) concrete types, including higher-order, sum and recursive ones. In this language, programs can be as large as the network and can observe communication events in other programs. Such flexibility is missing from current data-parallel languages and amounts to a fusion with their so-called annotations, directives or meta-languages. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Functional programming; Deterministic parallel programming; Semantic models; Concrete data structures

1. Explicit communications and functional programming

Faced with the mismatch between parallel programming languages and the requirements of their users, researchers are advocating resource-aware programming tools [13] and programs with measurable utilization of network capacity [22, 25], where control parallelism can be mixed with data parallelism. In this article, we propose semantic models for languages whose programs are explicitly parallel and whose semantics is functional. Such languages address the above-stated requirements by (1) expressing data placement, and hence communications explicitly,
(2) allowing higher-order functions to take placement strategies and generic computations as arguments,
(3) allowing higher-order functions to monitor communications within other functions, and yet avoid the complexity of concurrent programming.

Our starting point is the observation that our goals can be reached if the programming language expresses “physical” processes as in Krishnan’s distributed CCS [19] and exhibits a property found in Berry and Curien’s CDS language [3]: the possibility for a program to use another program’s meaning as data because functional arguments can be evaluated. Indeed network utilization, degree of parallelism and processor allocation are all visible parameters of a program, once the process decomposition of its meaning is explicit. If, moreover, programs can read each other’s meaning, then dynamic coordination of resources becomes possible and the language is in that sense “resource-aware”. In this paper we show how this is possible by merging Brookes and Geva’s generalization [7] of concrete data structures [18] with a notion of explicit processes.

To each concrete data structure we associate a replicated version called an **array structure** whose events occur in a finite space of process locations. The array structure’s domain of configurations provides a richer structure than the maps from processes to values used for data-parallel functional programming [10, 20] under the name of **data fields**. Array structures are a natural generalization of the single-program multiple data or SPMD realization of data-parallel programming [5] whereby a single program is replicated on every processor. In the array model the program is itself a distributed object rather than the replication of a “scalar”. Yet its **type**, a concrete data structure, is the replication of a scalar type on every processor. The category of array structures defines a **single-type multiple-data** or STMD paradigm that is, to our knowledge, novel.

By construction, the array structure of a stable concrete data structure is not stable so a Cartesian closed category (CCC) on their domains cannot be built from sequential algorithms [12]. We obtain instead a Cartesian closed category of generalized concrete data structures with explicitly distributed continuous functions.

We then consider the possible implementation of Skillicorn’s categorical data types by user-defined higher-order functions on array structures. In [26] he has shown how to specify a wide variety of parallel computations using recursively-defined categorical data types. However, recursive definitions of array structures often use sum structures, and we observe that Berry and Curien’s sum construction for concrete structures is not applicable to array structures. Moreover no array structure can generate the separated sum of two array domains.

To avoid the sum-types problem we are pushed to relax the array construction and consider non-homogeneous generalized concrete data structures, i.e. gcds’s that are simply distributed, not replicated. We therefore consider the category of distributed concrete structures and continuous functions as denotational semantics for a language à la Berry–Curien. An operational semantics is defined for it and full abstraction is proved. The possibility of mixing data parallelism with control parallelism in this language was illustrated in [21]. Examples demonstrate this paradigm’s expressive power. The
(unsolved) problem of a universal syntax for states is raised and finally, our approach is compared with two other paradigms of functional parallel paradigm: algorithmic skeletons and first-class schedules.

The rest of the paper is intended to be self-contained and is structured as follows. Section 2 recalls all the necessary definitions and properties from the work of Berry, Curien, Brookes and Geva, and fills a few missing details. Section 3 presents the category of array structures, its parallel algorithms and the problem of sum types. Section 4 defines the language schema CDS*, its denotational and operational semantics, proves their equivalence and then applies it to the definition of parallel computations. Small program examples (Appendix A) illustrate the useful properties of CDS* with respect to communication management in a distributed memory system. Our approach is then compared with other paradigms for high-level parallel programming.

2. Concrete- and generalized concrete data structures

This section reviews the necessary notions and results about concrete data structures and the cpo of generalized concrete data structures. The reader is assumed to be familiar with domain theory and Cartesian closed categories [1, 14]. Missing proofs can be found in [7, 8, 12].

2.1. Concrete data structures

Event structures [27] are concrete models for concurrent computations. They represent dependences and conflicts between computational events. Coherent sets of events constitute states and po-sets of states of event structures are concrete representations for certain types of domains. Concrete data structures are event structures whose events are built from a static part, called a cell, and a dynamic part called a value. The explicit mapping of cells and therefore of events (as opposed to whole states) to processors realizes a notion of explicit processes in this very general and flexible framework. This is our motivation for using concrete data structures.

2.1.1. Definitions

Definition 1. A concrete data structure or cds is a quadruple \((C, V, E, \vdash)\) containing:
- a denumerable set \(C\) of cells,
- a denumerable set \(V\) of values,
- a set \(E \subseteq C \times V\) of allowed events such that for every \(c \in C\) there exists a \(v \in V\) such that \((c, v) \in E\) and
- an enabling relation \(\vdash\) between finite sets of events and cells.

In the remainder, \(M, M', N \ldots\) will denote cds’s. Events \((c, v)\) and enableings \(\{(c_1, v_1), \ldots, (c_k, v_k)\} \vdash c\) will often be written \(cv\) and \(c_1v_1, \ldots, c_kv_k \vdash c\) respectively. A cell \(c\) is said to be initial (written \(\vdash c\)) if it is enabled by the empty set. Cell \(c\) is said to be filled in a set of events \(y\) if \(\exists v. cv \in y\). The set of cells filled in \(y\) is written \(F(y)\). If \(y \vdash c\) then \(y\) is said to be an enabling of \(c\) which is then considered enabled by every
superset \( y' \) of \( y \). This fact is written \( y \vdash y', c. \) \( E(y) \) is the set of cells enabled by \( y \). The set of cells accessible from \( y \) is \( A(y) = E(y) - F(y) \).

**Definition 2.** A state of \( M \) is a set of events \( x \subseteq E \) that is
- **functional:** if \( cv_1, cv_2 \in x \) then \( v_1 = v_2 \), and
- **safe:** if \( c \in F(x) \) then \( \exists y \subseteq x. y \vdash c \).

We write \( \mathcal{D}_M \), the set of states of \( M \), ordered by set inclusion, and call it the domain generated by \( M \). Such po-sets are called *concrete domains*.

We say that \( y \) covers \( x \) (written \( x \prec y \)) if \( x \subseteq y \) and if for every \( z, x \subseteq z \subseteq y \) implies \( z = y \), where \( x, y, z \) are states of a given cds. We write \( x \prec c y \) when \( x \prec y, c \in A(x) \) and \( c \in F(y) \).

For two cells \( c, d \) of a given cds \( M \), define \( c \ll d \) if and only if some enabling of \( d \) contains an event \( cv \). \( M \) is said to be well founded if \( \ll \) is well-founded. In the remainder we only consider well-founded cds’s. A cds is said to be *stable* if for every state \( x \) and cell \( c \in E(x) \), when \( X \vdash c \) and \( X' \vdash c \) with \( X, X' \subseteq x \), then \( X = X' \). A stable and well-founded cds is called *deterministic* (dcds).

**Example 3.** The cds \( \text{Bool} = (\{B\}, \{V, F\}, \{BV,BF\}, \{\vdash B\}) \) and \( \text{Nat} = (\{N\}, \{Nn \mid n \in \mathbb{N}\}, \{\vdash N\}) \) generate the flat domains of booleans and naturals, respectively:

\[
\begin{array}{cccc}
BF & BV & N0 & N1 & Nn \ldots \\
\downarrow & \downarrow & \uparrow & \uparrow & \uparrow \\
\emptyset & & & & \\
\end{array}
\]

The cds \( \text{Vnat} = (\mathbb{N}, \{*\}, \{n * \mid n \in \mathbb{N}\}, \{\vdash 0 \} \cup \{n \vdash n + 1 \}) \) generates the domain of lazy naturals:

\[
\begin{array}{c}
\{n * \mid n \in \mathbb{N}\} \\
\vdots \\
\{0*, 1*\} \\
\uparrow \\
\{0*\} \\
\uparrow \\
\emptyset
\end{array}
\]

**Proposition 4** (Kahn–Plotkin). Concrete domains are Scott domains, i.e. algebraic complete partial orders whose compacts are denumerable. The least element is the empty set of events and the l.u.b. of a directed set of states is their union as sets of events. The compacts are the finite states.

Concrete structures are thus appropriate types for first-order functional programs. From the point of view of denotational semantics, a cds is identified with its domain of states. Let us now describe some categories whose objects are concrete domains.
2.1.2. Product and sequential exponential

We first present the category of cds's built by Berry and Curien.

Definition 5. Let us write \( c.i \) the cell \( c \) labeled by integer \( i \). The product of two cds's \( M_1 \) and \( M_2 \) is a cds \( M_1 \times M_2 \) defined by

- \( C_{M_1 \times M_2} = \{ c.i \mid c \in C_{M_1}, \ i = 1, 2 \} \),
- \( V_{M_1 \times M_2} = V_{M_1} \cup V_{M_2} \),
- \( E_{M_1 \times M_2} = \{ c.i \ v \mid c.v \in E_{M_1}, \ i = 1, 2 \} \),
- for \( i = 1, 2 \), \( c_1.v_1, \ldots, c_k.v_k \vdash_{M_1 \times M_2} c.i \) iff \( c_1.v_1, \ldots, c_k.v_k \vdash_{M_i} c \).

It is easily verified that \( M_1 \times M_2 \) is a cds whose domain is order-isomorphic to the product of \( \mathcal{D}(M_1) \) and \( \mathcal{D}(M_2) \) ordered component-wise: the states of \( M_1 \times M_2 \) are the superpositions of pairs of states from the \( M_i \) [12].

Definition 6. Let \( \mathcal{D}_{\text{fin}}(M) \) be the set of finite states of \( M \). The Berry–Curien exponential of two cds's \( M \) and \( M' \) is a cds \( M \Rightarrow M' \) defined by

- \( C_{M \Rightarrow M'} = \mathcal{D}_{\text{fin}}(M) \times C_{M'} \), and we will write \( xc' \) for the couple \( (x, c') \).
- \( V_{M \Rightarrow M'} = \{ \text{output } v' \mid v' \in V_{M'} \} \cup \{ \text{valof } c \mid c \in C_M \} \),
- \( \{ xc'(\text{output } v') \mid x \in \mathcal{D}_{\text{fin}}(M), \ c'^{v'} \in E_{M'} \} \)
- \( \cup \{ xc'(\text{valof } c) \mid x \in \mathcal{D}_{\text{fin}}(M), \ c' \in C_{M'}, \ c \in A(x) \} \),
- \( x_1c_1'(\text{output } v_1'), \ldots, x_kc_k'(\text{output } v_k') \vdash_{M \Rightarrow M'} xc' \) if and only if \( c_1'^{v_1'}, \ldots, c_k'^{v_k'} \vdash_{M'} c' \) and \( \bigcup_{i=1}^k x_i = x \).
- \( xc'(\text{valof } c) \vdash_{M \Rightarrow M'} yc' \) if and only if \( x \prec_{c} y \).

The exponential is verified to be a cds and its states are called sequential algorithms. Given one of them \( a \in \mathcal{D}(M \Rightarrow M') \) and \( x \in \mathcal{D}(M) \), the application\(^1\) \( a.x \) of \( a \) to \( x \) is defined by

\[ a.x = \{ c'.v' \mid \exists y. \ y \subseteq x \text{ and } yc'(\text{output } v') \in a \}. \]

The valof label is used to introduce queries or “intermediate” values. The output label identifies proper output values.

If \( M' \) is stable then \( a.x \) is a state of \( M' \) and \( x \mapsto a.x \) is a continuous function from \( \mathcal{D}(M) \) to \( \mathcal{D}(M') \), called the I/O function or extensional content of algorithm \( a \). If \( M' \) is a cds, then \( M \Rightarrow M' \) is also deterministic (stable and well founded). The output events describe the I/O function of \( a \) and the valof events determine its intensional or control content. By design, sequential algorithms may not specify non-sequential functions like the parallel OR. They describe the evaluation order or arguments, which is more than the meaning of functions.

The following example highlights the stronger expressiveness of algorithms over functions by showing two distinct algorithms for the same strict AND function. The

\(^1\)Application is written \( a.x \) and this notation should not be confused with the labeling of cells with integers \( c.i \) used to construct product structures. Both notations are taken from Berry and Curien’s work as described in [12].
strict left \textsc{AND} algorithm $\text{AND}_{sl} : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}$ evaluates its left argument first then its second argument. It is defined as follows:

$$\text{AND}_{sl} = \{ \begin{cases} \{ \} B(\text{valof} B.1), & \{B.1 F\}B(\text{valof} B.2), \\ \{B.1 V\}B(\text{valof} B.2), & \{B.1 V, B.2 F\}B(\text{output} F), \\ \{B.1 F, B.2 V\}B(\text{output} F), & \{B.1 F, B.2 F\}B(\text{output} F), \\ \{B.1 V, B.2 V\}B(\text{output} V) \end{cases} \}.$$ 

The strict right \textsc{AND} algorithm $\text{AND}_{sr} : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}$ is symmetrically defined:

$$\text{AND}_{sr} = \{ \begin{cases} \{ \} B(\text{valof} B.2), & \{B.2 F\}B(\text{valof} B.1), \\ \{B.2 V\}B(\text{valof} B.1), & \{B.2 V, B.1 F\}B(\text{output} F), \\ \{B.2 F, B.1 V\}B(\text{output} F), & \{B.2 F, B.1 F\}B(\text{output} F), \\ \{B.2 V, B.1 V\}B(\text{output} V) \end{cases} \}.$$ 

### 2.1.3. Domain equations, sum of cds’s

Typed functional languages use recursive type definitions whose solutions are guaranteed by the cpo structure of the category of types and continuity of sums and products. When types are abstract domains, the least fixed point solutions of such equations are unique up to isomorphism. But in the case of concrete structures, Berry and Curien have defined an approximation between cds’s which provides exact solutions. To arrive at this a cpo of cds’s is needed whose order, structure inclusion, is stronger than the inclusion of concrete domains.

**Definition 7.** Let $M = (C_M, V_M, E_M, \vdash_M)$ and $M' = (C_{M'}, V_{M'}, E_{M'}, \vdash_{M'})$ be two cds’s. We say that $M$ is included in $M'$, and write $M \subseteq M'$ if

$$C_M \subseteq C_{M'}, \quad V_M \subseteq V_{M'}, \quad E_M \subseteq E_{M'}, \quad \vdash_M \subseteq \vdash_{M'}.$$ 

Let $X$ be a set. We write $\text{CDS}(X)$ (resp. $\text{DCDS}(X)$) the set of all cds’s (resp. dcds’s) $M = (C, V, E, \vdash)$ such that $C, V \subseteq X$.

**Proposition 8** (Berry–Curien). $(\text{CDS}(X), \subseteq)$ (resp. $(\text{DCDS}(X), \subseteq)$) is a cpo whose least element is $\text{Null} = (\emptyset, \emptyset, \emptyset, \emptyset)$ and where the l.u.b. of a directed set of cds’s is obtained by taking the union of the different components (unions of cells, etc.). The compacts of this cpo are the cds’s whose sets of cells and values are finite.

The least fixed point cds of an equation using continuous operations on a $\text{CDS}(X)$ is a set union. The most common operations used in this way are the product and the sum. We have already seen the former, let us now recall Berry and Curien’s definition of sum.

**Definition 9.** Given $M = (C_M, V_M, E_M, \vdash_M)$ and $M' = (C_{M'}, V_{M'}, E_{M'}, \vdash_{M'})$ the sum cds $M + M'$ is defined by

- $C_{M + M'} = \{S\} \cup \{c.L \mid c \in C_M\} \cup \{c'.R \mid c' \in C_{M'}\}$,
- $V_{M + M'} = \{L, R\} \cup V_M \cup V_{M'}$, 

\[ E_{M+M'} = \{SL, SR\} \cup \{c.Lv \mid cv \in E_M\} \cup \{c'.Rv' \mid c'v' \in E_{M'}\}, \]

\[ \vdash_{M+M'} S, \]

\[ SL, c_1, Lv_1, \ldots, c_k, Lv_k \vdash_{M+M'} c.L \text{ iff } c_1v_1, \ldots, c_kv_k \vdash_M c, \]

\[ SR, c'_1, Rv'_1, \ldots, c'_k, Rv'_k \vdash_{M+M'} c'.R \text{ iff } c'_1v'_1, \ldots, c'_kv'_k \vdash_{M'} c'. \]

It is straightforward to verify that \( M + M' \) is a cds. In it, cell \( S \) (definition 9) acts as a selector: once a state contains \( SL \) (resp. \( SR \)), it can only increase by including (labeled) states of \( M \) (resp. \( M' \)). It follows that \( \mathcal{D}(M + M') \) is isomorphic to the separated sum of \( \mathcal{D}(M) \) and \( \mathcal{D}(M') \).

For example, we obtain \( \text{Bool} + \text{Nat} \) from the cds’s \( \text{Bool} \) and \( \text{Nat} \):

\[ C_{\text{Bool}+\text{Nat}} = \{S, B.L, N.R\}, \quad V_{\text{Bool}+\text{Nat}} = \{L, R, V, F\} \cup \mathbb{N}, \]

\[ E_{\text{Bool}+\text{Nat}} = \{SL, SR, B.LV, B.LF\} \cup \{N.Rn \mid n \in \mathbb{N}\}, \]

where \( S \) is initial, \( SL \vdash B.L, \ SR \vdash N.R \). Domain \( \mathcal{D}(\text{Bool} + \text{Nat}) \) is

\[
\begin{array}{ccc}
{SL, B.LV} & {SL, B.LF} & \cdots \\
\downarrow & \downarrow & \\
{SL} & {SR, N.R0} & \emptyset \\
\downarrow & \downarrow & \\
{SR} & {SR, N.R1} & \\
\end{array}
\]

**Proposition 10** (Berry–Curien). The constructors \( \times, + \) and \( \Rightarrow \) are continuous functions \( \text{CDS}(X)^2 \rightarrow \text{CDS}(X) \) (resp. \( \text{DCDS}(X)^2 \rightarrow \text{DCDS}(X) \)).

This property validates the recursive definition of cds’s. For example, the set of lists on base type \( M_0 \) is the least fixed point of \( M = \text{Null} + (M_0 \times M) \), or concretely: the union of its approximations.

2.1.4. A Cartesian closed category and a sequential language

Berry and Curien have shown that neither continuous functions, stable functions nor so-called sequential functions preserve dcds’s (or cds’s). However, as stated above, if \( M \) and \( M' \) are dcds’s then so are the space of sequential algorithms \( M \Rightarrow M' \) and the product \( M \times M' \). It would appear that we have a Cartesian closed category of dcds’s and sequential algorithms. This is indeed the case but the existence of a composition is not trivially true. It was proved in [12] by using so-called abstract algorithms whose construction amounts to a separation of the control-strategy from the I/O part of algorithms.
From this category, Berry and Curien have built a sequential functional language called CDS [3] and proved a full abstraction result for it. Its types are dcds’s and programs denote sequential algorithms, i.e. states of exponential dcds’s. The operations of CDS are those of “ordinary” typed functional languages except that functions are enumerated as sets of events.

In CDS, base-type data and algorithms are all states. Algorithms can therefore evaluate other algorithms and read their meaning. For example, a program whose type is of the form $\text{Bool} \times \text{Bool} \Rightarrow \text{Bool}$ can distinguish between various algorithms for the AND function: left-strict, right-strict etc. This property of dcds models is potentially useful for parallel programming where evaluation strategies play a critical role.

The auxiliary question of a source language more convenient than CDS has been avoided in [3]. But more recent work on sequentiality [9] has expanded and generalized that framework to a full abstraction result for SPCF, a sequential extension of PCF with errors. SPCF defines a more realistic language having theoretical properties equivalent to those of CDS.

Unfortunately, our requirements for asynchronous parallel languages force us to avoid sequential algorithms and dcds’s. It is therefore impossible to adapt SPCF to the approach described here.

### 2.2. Generalized dcds’s

Brookes and Geva [7,8] have generalized the notion of dcds by adding an order on cells. This construction ensures Cartesian closure for continuous functions and removes the need to use deterministic (hence sequential) dcds’s.

#### 2.2.1. Definitions

**Definition 11.** A generalized dcds (or gcds) is a tuple $M = ((C, \leq), V, E, \vdash)$ where

- $(C, \leq)$ is a denumerable po-set of cells,
- $V$ is a denumerable set of values,
- $E \subseteq C \times V$ is a set of allowed events, closed upwards for cells: if $cv \in E$ and $c \leq c'$ then $c'v \in E$,
- enabling $\vdash$ is a relation between finite sets of events and cells, and must be closed upwards for cells: if $y \vdash c$ and $c \leq c'$ then $y \vdash c'$.

A state of gcds $M$ is a set $x \subseteq E$ that is functional, safe and closed upwards for cells: if $cv \in x$ and $c \leq c'$ then $c'v \in x$. For $x$ a set of events, define

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

The order on cell plays no particular role for a base-type gcds which is usually given the discrete ordering on cells (and is then equivalent to a standard cds). The case of higher-order gcds’s is different: upwards closure of states for $\leq$ allows the application of an exponential state regardless of the output gcds. Unlike sequential algorithms on cds’s, there is no stability (dcds) requirement and as a result exponential gcds’s encode continuous functions.
The set of states of a gcds is written $\mathcal{D}(M)$ and ordered by set inclusion as before. Po-sets generated thus are called generalized concrete domains.

**Proposition 12** (Brookes and Geva). Generalized concrete domains are consistently complete Scott domains where the least element is the empty set and the l.u.b. of a directed set of states is their union as sets of events. The compacts are the upwards $\leq$-closures of finite sets of events.

### 2.2.2. The category of gcds’s and continuous functions

The category $\text{gCDScont}$ is defined with gcds’s as objects and continuous functions between their domains as arrows. Composition is function composition and the identity arrows are the identity functions. The empty gcds $\text{Null}$ is a terminal object in this category.

**Definition 13.** The product of two gcds’s $M_1$ and $M_2$ is defined by

- $C_{M_1 \times M_2} = \{ c,i | c \in C_{M_i}, \ i = 1,2 \}$,
- $c,i \leq_{M_1 \times M_2} c',i'$ iff $c \leq_{M_i} c'$ and $i = i'$,
- $V_{M_1 \times M_2} = V_{M_1} \cup V_{M_2}$,
- $E_{M_1 \times M_2} = \{ c,i \ v | cv \in E_{M_i}, \ i = 1,2 \}$,
- for $i = 1,2$, $c_1,i \ v_1, \ldots, c_k,i \ v_k \vdash_{M_1 \times M_2} c,i$ iff $c_1 v_1, \ldots, c_k v_k \vdash_{M_i} c$.

When $M_1,M_2$ are gcds’s it is easily verified that $M_1 \times M_2$ is a gcds and that its domain $\mathcal{D}(M_1 \times M_2)$ is order-isomorphic to $\mathcal{D}(M_1) \times \mathcal{D}(M_2)$ ordered pointwise.

The obvious projections $\pi_i(x) = \{ cv | c,i \ v \in x \}$ are continuous, pairing $(f,g) = \lambda x \in \mathcal{D}(M). (f(x),g(x))$ is an operation from $[\mathcal{D}(M) \to \mathcal{D}(M_1)] \times [\mathcal{D}(M) \to \mathcal{D}(M_2)]$ into $[\mathcal{D}(M) \to \mathcal{D}(M_1 \times M_2)]$, and with this structure the product of gcds’s is a categorical product for $\text{gCDScont}$.

The (extensional) exponential of two gcds’s is essentially the extensional part (output values) of sequential algorithms on the corresponding cds.

**Definition 14.** For two gcds’s $M,M'$, the gcds $M \to M'$ is defined by

- $C_{M \to M'} = \mathcal{D}_{\text{finite}}(M) \times C_{M'}$, where $x'c' \leq_M x'y'd'$ iff $x' \subseteq y'$ and $c' \leq_{M'} d'$,
- $V_{M \to M'} = V_{M'}$,
- $E_{M \to M'} = \{ xc'v' \ | x \in C_{M \to M'}, c'v' \in E_{M'} \}$,
- $x_1 c'_1 v'_1, \ldots, x_k c'_k v'_k \vdash_{M \to M'} x'c' \text{ iff } c'_1 v'_1, \ldots, c'_k v'_k \vdash_{M'} c'$ and $x_i \subseteq x$ for every $i$.

Here $\mathcal{D}_{\text{finite}}(M)$ is the set of compacts of $\mathcal{D}(M)$. By Proposition 12 they are finite sets of $\leq$-incomparable events of $M$. An exponential event $xc'v'$ means that input states containing $x$ will generate output event $c'v'$.

**Proposition 15** (Brookes and Geva). If $M,M'$ are gcds’s then $M \to M'$ is also a gcds and $\mathcal{D}(M \to M')$ is order-isomorphic to the space of continuous functions $[\mathcal{D}(M) \to \mathcal{D}(M')]$, ordered pointwise.
The isomorphisms are, for \( a \in \mathcal{D}(M \to M') \) a state and \( f \) a continuous function
\[
a \mapsto \lambda z \in \mathcal{D}(M).\{c'v' \mid \exists x \subseteq z. xc'v' \in a\}
\]
\[
f \mapsto \{xc'v' \mid c'v' \in f(x)\}.
\]

**Proposition 16** (Brookes and Geva). \textit{gcds inclusion} is Cartesian closed.

The Cartesian-closed structure comes from functions \( \text{app}_{M,M'} \) and \( \text{curry}_C \):
\[
\text{app}_{M,M'} : (M \to M') \times M \to M',
\]
\[
\text{curry}_C : (C \times M) \to M \to (M \to M'),
\]
where \( \text{app}(a,x) = a.x, \text{curry}(a) = \{xc(x_Mc')v' \mid (\Psi(x_C,x_M))c'v' \in a\} \) and \( \Psi \) is the isomorphism between \( \mathcal{D}(C) \times \mathcal{D}(M) \) and \( \mathcal{D}(C \times M) \).

For two gcds's \( M, M' \) we define \( M + M' \) as in Definition 9 except that the order on cells is the disjoint union of \( \preceq_M, \preceq_{M'} \) and \( \{S \leq S\} \). The selector cell \( S \) is incomparable with other cells. It is straightforward to verify the following.

**Proposition 17.** When \( M, M' \) are gcds's, \( M + M' \) is also a gcds and \( \mathcal{D}(M + M') \) is isomorphic to the separated sum of \( \mathcal{D}(M) \) and \( \mathcal{D}(M') \).

Given a set \( X \), let \( \text{GCDS}(X) \) be the set of all gcds's \( M = ((C, \preceq), V,E,\vdash) \) such that \( C, V \subseteq X \) and define \( M \subseteq M' \) as for cds's, with the additional provision that \( \preceq_M \) be a sub-relation of \( \preceq_{M'} \).

**Proposition 18.** \( \text{GCDS}(X, \subseteq) \) is a cpo whose least element is Null and where the l.u.b. of a directed set of gcds's is obtained by taking the union of the different components (unions of cells, etc.). The compacts of this cpo are the gcd's M having finitely many cells and values.

**Proof.** Gcds inclusion is a partial order and Null is the least element for it because it is component-wise inclusion on 5-tuples of sets.

Let \( S \) be a directed set of gcds's. For every \( M_1, M_2 \in S \) there exists \( M \in S, M_1 \subseteq M, M_2 \subseteq M \). Define \( \widecheck{M} = (\widehat{C}, \leq, \widehat{V}, \widehat{E}, \widehat{\vdash}) \) where \( \widehat{C} = \bigcup_{M \in S} C_M, \leq = \bigcup_{M \in S} \preceq_M, \widehat{V} = \bigcup_{M \in S} V_M, \widehat{E} = \bigcup_{M \in S} E_M \) and \( \widehat{\vdash} = \bigcup_{M \in S} \vdash_M \). By Proposition 8, \( (\widehat{C}, \widehat{V}, \widehat{E}, \widehat{\vdash}) \) is a cds and to verify that \( \widehat{M} \) is a gcds it is sufficient to verify the required properties of \( \leq \). Reflexivity of \( \leq \) is obvious. Transitivity holds because if \( c_1 \leq c_2 \leq c_3 \) then there exists \( M_1, M_2 \in S \) such that \( c_1, c_2 \in M_1, c_2, c_3 \in M_2 \) and \( c_1 \preceq_M c_2 \preceq_M c_3 \). But since \( S \) is directed, there exists \( M_i \in S \) such that \( M_i \subseteq M_3 \) for \( i = 1, 2 \) and so \( \preceq_M \subseteq \preceq_{M_3} \) for \( i = 1, 2 \). Since \( \preceq_{M_i} \) is transitive by hypothesis, transitivity follows for \( \leq \). To prove anti-symmetry suppose \( c_1 \leq c_2 \leq c_1 \). Then there exists \( M_1, M_2 \in S \) such that \( c_1 \preceq_M c_2 \preceq_M c_1 \) and again because \( S \) is directed there exists \( M \in S \) such that \( c_1 \preceq_M c_2 \preceq_M c_1 \), and since \( \leq_M \) is anti-symmetric we have \( c_1 = c_2 \). Upwards \( \leq \)-closures of \( \widehat{E} \) and of \( \widehat{\vdash} \) are verified.
along the same lines. We have shown that \( \hat{M} \in GCDS(X) \) and by construction it must be the l.u.b. of \( S \).

If \( M \in GCDS(X) \) is such that \( C_M, V_M \) are finite, it is easily verified that \( M \) is a compact. Conversely, assume an infinite number of cells \( C_M = \{c_1, c_2, \ldots\} \) and let \( M_n \) be the restriction of \( M \) to \( C_n = \{c_1, \ldots, c_n\} \), i.e.

\[
M_n = (C_n, \preceq_M \cap C_n^2, V_M, E, \vdash_M \cap (E \times C_n)),
\]

where \( E = E_M \cap (C_n \times V_M) \). Then \( M_n \) is a gcds and \( S = \{M_n\}_{n \geq 0} \) is a chain (hence a directed subset) of \( GCDS(X) \). Clearly, \( \bigcup S = M \) but \( M \) is contained in none of the \( M_n \). So \( M \) is not a compact. Similarly, if \( V_M \) is infinite then \( M \) is not a compact. \( \square \)

There is a property of gcds’s analogous to Proposition 10. It validates the recursive definition of gcds’s.

**Proposition 19.** The constructors \( \times, + \) and \( \rightarrow \) are continuous functions of \( GCDS(X)^2 \) into \( GCDS(X) \).

**Proof.** Continuity of \( \times \) and \( + \) is straightforward to verify. Consider the exponential. Given \( E = M_1 \rightarrow M_2 \) and \( E' = M'_1 \rightarrow M'_2 \) where \( M_1 \subseteq M'_1 \) and \( M_2 \subseteq M'_2 \), the inclusion \( E \subseteq E' \) follows from the definition of exponential gcds’s. For example \( xc \preceq_M \rightarrow M_2 \) \( yd \) is equivalent to \( x \subseteq y, \ c \preceq_M y \) \( d \) which implies \( x \subseteq y, \ c \preceq_M y \) \( d \), i.e. \( xc \preceq_M \rightarrow M'_1 \) \( yd \). Preservation of limits follows from set-theoretical properties. \( \square \)

3. Array structures

The most common paradigm for data-parallel algorithms is that of Single Program Multiple Data (SPMD) programming, whereby a single program is replicated on every processor. Execution then proceeds by alternating between asynchronous local computations and global synchronization barriers. Communications and synchronizations are explicit so as to let the programmer in control of the equilibrium between computation, communication and synchronization. In the BSP [22] variant of this paradigm a portable cost-model is defined (for suitably restricted programs) in terms of communication traffic and frequency of barriers. This section presents a model for SPMD-like languages having compositional semantics, recursion and higher-order functions. It is proved that such languages may not use sum types.

A reasonable model of SPMD programs must be able to describe the asynchronous parts of computations. To build such a model, we cannot augment the category of gcds’s with explicit processes because its arrows, the sequential algorithms, require stable concrete data structures. On the other hand, the generalized cds’s of Brookes and Geva (Section 2.2) use continuous functions and thus remove the stability-sequentiality requirement.
3.1. Definitions

Given a gcds we construct an array gcds by replicating its cells over a finite set \( I \) of nodes or indices. This generalizes the usual notion of arrays: ours are arrays of events, not of values. Array indices represent addresses in a static processor network and every array structure refers to the same set of indices.

**Definition 20.** Let \( M = (C_0, \leq_0, V_0, E_0, \vdash_0) \) be a gcds. The array data structure, \( \text{ads} \) or **array structure** over \( M \), \( M^\square = (C, \leq, V, E, \vdash) \) is defined as follows.

- \( C = I \times C_0 \). A cell \((i, c)\) will be written \( ic \) in the array structure and said to be located at (location) \( i \). Cells are ordered locally: \( ic \leq jc' \) iff \( i = j \) and \( c \leq_0 c' \).
- \( V = V_0 \).
- \( E = I \times E_0 \) and event \((i, cv)\) will be written \( icv \).
- The enabling relation \( \vdash \) is between \( \mathcal{P}_{\text{fin}}(I \times E_0) \) and \( I \times C_0 \). For any two locations \( i, j \):

\[
\{jc_1v_1, \ldots, jc_kv_k\} \vdash ic \iff \{c_1v_1, \ldots, c_kv_k\} \vdash_0 c.
\] (1)

Notice that events on the left-hand side of (1) must be co-located. This is motivated as follows. Given the symmetry of array structures, the only other possible choices would be to allow any subset of \( I \) or all of \( I \) (and then a fixed arity for \( \vdash_0 \)) for the locations of enabling events. The latter is too restrictive since it amounts to a global synchronization for every enabling action. The former is indeed possible but unnatural with \( M^\square \): an arbitrary set of locations for the enablings of a symmetric structure. It is then more natural to have complete flexibility with general gcds’s: arbitrary but explicitly specified locations for every component. We will consider this alternative later.

Enablings for which \( j = i \) allow \( M^\square \) to recover a copy of the enabling relation of \( M \) at any location \( i \): events located at a common site enable cells at the same site according to \( \vdash_0 \). Non-local \((j \neq i)\) enablings determine the expansion of states to new locations; a cell located at \( i \) is enabled by a set of events located at a neighbouring index of \( j \). As a result, \( \vdash_0 \)-initial cells remain initial at every location.

To verify that \( M^\square \) is a gcds, we observe that \( C \) is denumerable because \( I \) and \( C_0 \) are both denumerable. By hypothesis \( V \) is also countable. The type of \( E \) is correct since \( E_0 \subseteq C_0 \times V_0 \) implies \( E \subseteq I \times C_0 \times V_0 = C \times V \). Because \( M \) is a gcds, it follows that \( E \) and \( \vdash \) are closed upwards with respect to \( \leq \). The following property of the enabling relation must also be satisfied to make \( M^\square \) a gcds.

**Lemma 21.** \( \ll \) is well-founded.

**Proof.** By considering enablings in \( \vdash \) we find that, \(jc' \ll ic \) only if \( c' \ll_0 c \). Therefore a descending chain \( i_1c_1 \gg i_2c_2 \gg \cdots \) corresponds to a descending chain in \( M \): \( c_1 \gg_0 c_2 \gg_0 \cdots \). But since \( M \) is a gcds, its \( \ll_0 \) relation is well-founded and such chains must be finite. \( \square \)
We have verified that

**Proposition 22.** $M^{\square}$ is a gcds.

Proposition 12 is therefore applicable.

**Corollary 23.** The set $\mathcal{D}(M^{\square})$ of states of an array structure is a consistently complete Scott domain where $\text{sup} = \cup$ and $\bot = \emptyset$. We will call it an array domain and its states arrays over $M$. Its compacts are the upwards $\leq$-closure of finite arrays.

Two more remarks about $M^{\square}$.

1. In general, the projection $t(i) = t \cap (\{i\} \times C_0 \times V_0)$ is not a state of $\mathcal{D}(M)$, because cells of $t \in \mathcal{D}(M^{\square})$ may be enabled remotely. For example, $\{i0*, i1*, j1*, j2*\} \in \mathcal{D}($Vnat$^{\square})$ but (assuming $i \neq j$) its projection on location $j$ is not a state of Vnat: event $j1*$’s only enabling is remote. This is precisely how array structures generalize data fields [10, 20], by removing the obligation for local parts of dynamic structures to be observable scalar structures.

2. Because a cell can be enabled either locally or remotely, enabling is not unique even when they were so in $M$. In the above example, $i1* \vdash j2*$ and $j1* \vdash j2*$ within the same state. It follows from this second remark that the array construction does not preserve stability if applied to stable cds’s. As a result there is no hope of inventing a distributed version of $\Rightarrow^2$ and this is why we use a sub-category of gCDScont.

3.2. A category of array domains

3.2.1. Composition and terminal object

Let $M_1, M_2, M_3$ be gcds’s. The identity transformations on $\mathcal{D}(M^{\square})$ are continuous and function composition preserves continuity. Define category ADScont as a sub-category of gCDScont with ads as objects and continuous functions between their domains as arrows. By definition $\text{Null}^{\square} = \text{Null}$ and so $\text{Null}^{\square}$ is a terminal object of ADScont.

3.2.2. Product: pairs of arrays

The product of array structures is a special case of the product of gcds’s. In $M_1^{\square} \times M_2^{\square}$, as with every two gcds’s, the two enabling relations are superimposed without interaction. A pair of arrays therefore corresponds to an array of pairs.

**Lemma 24.** Given gcds’s $M_1, M_2$, $\mathcal{D}((M_1 \times M_2)^{\square})$ and $\mathcal{D}(M_1^{\square} \times M_2^{\square})$ are isomorphic.

**Proof.** Consider a state $x$ of $(M_1 \times M_2)^{\square}$ and define

$$\text{split}_{M_1, M_2} x = (x_1, x_2) \quad \text{where} \quad x_i = \{cv \mid c.i \cdot v \in x\}.$$

---

$^2$Hypothetically: using vector events or similar constructions.
Clearly \(x_1\) and \(x_2\) are sets of events in \(M_1^\square\) and \(M_2^\square\) respectively. It is straightforward to verify that both are functional (because \(x\) is), that all their events have enablings (by definition of the product enabling) and that they are closed upwards for the cell order (component-wise in \(M_1^\square \times M_2^\square\)). As a result \(x_i\) is a state of \(M_i^\square\). The inverse of \(\text{split}_{M_1,M_2}\) is \(\text{merge}_{M_1,M_2}\), it reconstructs \(x\): \(\text{merge}(x_1,x_2) = \{c.i.v \mid cv \in x_i, \ i = 1,2\}\). The correspondence is bijective and preserves unions. \(\square\)

The product is a categorical product in \(\text{gCDScont}\) and preserves array domains. It is therefore a categorical product in \(\text{ADScont}\).

### 3.2.3. Exponential domains: array transformations

**Proposition 25.** For \(M,N\) gcds’s, \(\mathcal{D}((M^\square \to N^\square))\) and \(\mathcal{D}((M^\square \to N^\square))\) are isomorphic.

**Proof.** The two gcds’s are identical up to a permutation of components. Consider \((M^\square \to N^\square) = (C, \leq, V, E, \vdash)\), the cells

\[
C = \mathcal{D}_{\text{fin}}(M^\square) \times C_N^\square = \mathcal{D}_{\text{fin}}(M^\square) \times I \times C_N
\]

\[
\equiv I \times \mathcal{D}_{\text{fin}}(M^\square) \times C_N = I \times C_{M^\square \to N^\square} = C_{(M^\square \to N^\square)^\square},
\]

the cells orderings

\[
\leq = \subseteq \times \leq_N^\square = \subseteq \times \text{id}_I \times \leq_N
\]

\[
\equiv \text{id}_I \times \subseteq \times \leq_N = \text{id}_I \times \leq_M^\square \to N^\square = \leq_{(M^\square \to N^\square)^\square}
\]

and the values \(V = V_N^\square = V_N = V_{M^\square \to N^\square} = V_{(M^\square \to N)^\square}\). The events are \(E_{M^\square \to N^\square} = \mathcal{D}_{\text{fin}}(M \times E_N\) and \(E_{M^\square \to N^\square} = I \times E_M\) and to verify their equivalence:

\[
E_{M^\square \to N^\square} = \mathcal{D}_{\text{fin}}(M^\square) \times E_{N^\square} = \mathcal{D}_{\text{fin}}(M^\square) \times I \times E_N
\]

\[
\equiv I \times \mathcal{D}_{\text{fin}}(M^\square) \times E_N = I \times E_{M^\square \to N^\square} = E_{(M^\square \to N^\square)^\square}.
\]

Recalling the definitions of enabling for the exponential and array structures:

\[
x_1e_1,\ldots,x_ke_k \vdash_{M^\square \to N^\square} x \text{ c iff } e_1,\ldots,e_k \vdash_N^\square \text{ c and } \forall l.x_l \subseteq x.
\]

\[
je_1,\ldots,je_k \vdash_{M^\square \to N^\square} ic \text{ iff } e_1,\ldots,e_k \vdash_{M^\square \to N^\square} ic
\]

we verify that the enablings are the same up to our permutation:

\[
\vdash_{M^\square \to N^\square} = \{((\{x_1je_1,\ldots,x_je_k\}xic) \mid je_1,\ldots,je_k \vdash_{M^\square \to N^\square} \text{ c and } \forall l.x_l \subseteq x\}
\]

\[
= \{((\{jx_1e_1,\ldots,jxe_k\}xic) \mid e_1,\ldots,e_k \vdash_{M^\square \to N^\square} \text{ c and } \forall l.x_l \subseteq x\}
\]

\[
= \{((\{jx_1e_1,\ldots,jxe_k\}xic) \mid x_1e_1,\ldots,x_ke_k \vdash_{M^\square \to N^\square} \}
\]

The isomorphisms interchange the position of indices with input states in functional events \((ixe \leftrightarrow xie)\) and both are continuous. \(\square\)

The above proof does not use the fact that the input domain is an array domain. Therefore

**Corollary 26.** For \(M, N\) gcds’s, \(\mathcal{D}(M \rightarrow N) \cong \mathcal{D}((M \rightarrow N))^\square\).

The proposition implies Cartesian closure for the category of arrays.

**Corollary 27.** \(\text{ADS}_\text{cont}\) is a CCC.

**Proof.** \(\text{ADS}_\text{cont}\) is closed for (the terminal object and) the product and exponentiation of its enclosing category \(\text{gCDS}_\text{cont}\). Application and currying are taken from \(\text{gCDS}_\text{cont}\). \(\square\)

To illustrate the structure of \(\text{ADS}_\text{cont}\), consider the following example. Read arrays \(t \in \mathcal{D}(\text{Vnat}^\square)\) as maps from \(I\) to unary integers with the provision that for example \(\{0^*, 1^*, 2^*, 7^*\}\) is interpreted as 7 (remember that in general \(t(i)\) is not a state of \(\text{Vnat}\)). According to this representation, a union of sets of events corresponds to their pointwise maximum. Consider the maximum function:

\[
\max^\square : \text{Vnat}^\square \rightarrow \text{Vnat}^\square : \forall i. (\max^\square t)i = \bigcup \{ti | i \in I\}.
\]

It computes the overall maximum of integers in the array and distributes the result everywhere as shown in Fig. 1. \(\max^\square\) is a state of \((\text{Vnat}^\square \rightarrow \text{Vnat})^\square \cong (\text{Vnat}^\square \rightarrow \text{Vnat}^\square)\) and its events have the form \(xin^* \cong ixn^*\) where \(x \in \mathcal{D}_\text{fin}(\text{Vnat}^\square)\).

Let \(I = \{a, b, c\}\) and write array \(t\) as \([x | y | z]\) where \(x = t(a),\ y = t(b),\ z = t(c)\).

### 3.3. Parallel algorithms and sum types

Brookes and Geva have constructed the co-Kleisli CCC of gcds’s and parallel\(^3\) algorithms as framework for an intensional semantics where algorithms, are related to each other and to continuous functions. It encodes “timed” states as continuous functions \(\text{Vnat} \rightarrow M\). The timed states are inputs to algorithms from \(M\) to \(N\): continuous functions \((\text{Vnat} \rightarrow M) \rightarrow N\).

---

\(^3\) Not sequential in the sense of Berry–Curien.
We have defined [15, 16] an array version of this construction where $\mathbb{Vnat}$ is replaced by the product of $|I|$ copies of $\mathbb{Vnat}$. The result is a category $\mathbb{ADSalg}$ of array structures and algorithms.

Its algorithms are more expressive than the continuous functions of $\mathbb{ADScont}$. However their distributed implementation is troublesome in the following sense. Array algorithms violate local causality: a clock tick at location $i$ can cause events at location $j$ for $i \neq j$. For this reason, we do not consider array algorithms to be applicable to explicitly parallel languages.

Moreover, the array construction itself suffers from a basic flaw: it does not allow for a definition of sum types. We now prove this new result.

Suppose, by contradiction, the existence of a sum $M_1 + M_2$ for $\mathcal{D}(M_1)$ and $\mathcal{D}(M_2)$. The question is to find a gcds $M_+$ with this property. An obvious candidate is $M_1 + M_2$ but:

**Lemma 28.** The separated sum of $\mathcal{D}(M_1)$ and $\mathcal{D}(M_2)$ is not isomorphic to $\mathcal{D}((M_1 + M_2))$.

**Proof.** Let $M_1 = M_2 = \text{Null}$. Then $M_1 = M_2 = \text{Null} = \text{Null}$ as we have seen before. We have $\mathcal{D}((\text{Null} + \text{Null})) = \mathcal{D}((\text{Null} + \text{Null}))$ a 3-element flat domain isomorphic to $\mathcal{D}(\text{Bool})$. On the other hand, $\text{Null} + \text{Null}$ is $(\{S\}, =, \{L, R\}, \{SL, SR\}, \vdash)$ which is equivalent to gcds $\text{Bool}$. Therefore $\mathcal{D}((\text{Null} + \text{Null}))$ is isomorphic to $\mathcal{D}(\text{Bool})$, the Cartesian product of $I$ copies of $\mathcal{D}(\text{Bool})$. Whenever $I$ contains at least two elements, the two domains are different. $\square$

In fact, the array construction is incompatible with sum types.

**Proposition 29.** There exists no gcds $M_+$ for which $\mathcal{D}(M_+)$ is isomorphic to the separated sum of $\mathcal{D}(M_1)$ and $\mathcal{D}(M_2)$.

**Proof.** Let again $M_1 = M_2 = \text{Null}$, $M_1 = M_2 = \text{Null} = \text{Null}$ and $\mathcal{D}(M_1) + \mathcal{D}(M_2) \cong \mathcal{D}(\text{Bool})$. Assume a 3-element index set $I = 1, 2, 3$, or larger. If $M_+ \cong \text{Null}$ then the property fails. So $M_+$ must have at least one initial cell $c_0$ and some allowed event $c_0v$. As a result $\mathcal{D}(M_+)$ contains the flat 4-element po-set $\{0 \leq \{ic_0v\} \mid i = 1, 2, 3\}$ and so cannot be isomorphic to $\mathcal{D}(\text{Bool})$. $\square$
We conclude that the symmetric data-parallel aspect of array structures is compatible with higher-order functions but not with sum structures. For this reason we find it necessary to move out of the SPMD framework by removing the symmetry requirement and using general gcds’s with explicit process locations.

4. gCDScont as model for parallel programs

In the previous section, we observed that array structures are incompatible with sum types. But without sum, it is impossible to build more complex structures such as trees. This forces us to consider non-homogeneous gcds’s which are distributed and not replicated. We assume again a finite set \( I \) of indices or addresses in a static multiprocessor network. A distributed concrete structure is a gcds with a total function \( A \) from the set of cells to the set of indices, called the location function. To be exact, the category of distributed concrete structures is not that of gcds’s, but locations only affect the intended implementation (yet in a crucial way: dependences between requests/values generate communications or not depending on locations of their associated cells) and can be overlooked at the categorical level. The distributed gcds constitute a CCC that is closed for sums and equivalent to gCDScont: a normal gcds corresponds to a distributed one with constant location function (e.g. \( A(c) = 1, \forall c \)) and a distributed gcds corresponds to a normal one where location values are part of the cell names (as in \( c \mapsto (c, A(c)) \)). In other words, location is part of the cell name and no implicit link is made between cells having the same de-located name. This is the assumption made in [21], with the understanding that practical programming languages based on the same model would also provide location-abstraction mechanisms. From now on, “gcds” and “gCDScont” will refer to distributed structures and their category.

For the purpose of the language schema given below, it is sufficient to know that every cell is statically allocated to a processor. The combination of this information with the operational semantics determines the parallel execution of a given program.

4.1. A language schema: CDS*

We are now able to construct a simple programming language called CDS*, that meets our initial goals (Section 1) for applying concrete types to parallel programming:

1. Data placement is explicit in the syntax, and so are the functional dependences which generate communications.

2. A higher-order function may take as argument a sequential placement function computing the processor index of a given sequential task. The main function can then generate a parallel function based on this placement.

\footnote{From this point of view data-parallelism should remain an algorithm-design paradigm and not be used as a principle of language design. This is indeed how the term was introduced by D. Hillis and G. Steele in their article Data parallel algorithms, Communications of the ACM, December 1986.}
As illustrated by the TASTER and $H$ (monitor) programs below, higher-order functions are able to monitor functional dependences within other functions. This capability can be applied to load-balancing or communication minimization.

$CDS^*$ is a functional language schema similar to a simply typed lambda-calculus whose types are gcds’s. We refer to $CDS^*$ as a language schema because the question of a practical syntax for its states (in particular, functional ones) is left open. This choice is analogous to the algorithmic skeletons paradigm where the host language is disconnected from those functions whose implementation is parallel. The $CDS^*$ paradigm is more general because the language itself gives essential information about those parallel implementations: data distribution and data dependences in functions. We will return to the skeletons paradigm in Section 4.4.

For the sake of clarity, we will omit types from the presentation of $CDS^*$’s operational semantics. On the other hand, the programming examples will be explicitly (and simply) typed: implicit typing and polymorphism for concrete types are open problems.

Another practical problem concerning $CDS^*$ is that of memory management for its implementation. Concrete states are histories of unlimited size and they can lead to space-inefficient programs. However memory management is a general weakness of declarative languages and there is no indication that a $CDS^*$-based language could not apply well-understood techniques such as garbage collection, static analysis or linear types. For this reason we consider this question as orthogonal to the present study.

$CDS^*$ functions are not written as $\lambda$-abstraction but as gcds states. $CDS^*$ is a general model for predefined parallel functions (algorithmic skeletons in the sense of [11]) with an explicit and abstract treatment of events. Inter-event dependences that generate communications are explicit in the language.

$CDS^*$ as defined here is a functional harness that makes explicit the events of the parallel functions it composes. Apart from state, terms use the following operators: application $T.U$, composition $T \circ U$, fixed-point $\text{fix}(T)$, $\text{curry}$, $\text{uncurry}$, coupling $(T,U)$ and pairing $(T,U)$. Their grammar is

$$T ::= x \mid T.T \mid T \circ T \mid \text{fix}(T) \mid \text{curry}(T) \mid \text{uncurry}(T) \mid (T,T) \mid \langle T,U \rangle$$

where $T$ is the non-terminal for terms and the syntax of state constants $x$ is left unspecified. Appendix A illustrates one possible syntax for state constants.

The denotation of terms is given by the following:

**Definition 30.** The denotational semantics $\langle T \rangle$ of a term is given by

- $\langle x \rangle = x$,
- $\langle T.U \rangle = \langle T \rangle . \langle U \rangle$,
- $\langle T_1 \circ T_2 \rangle = \langle T_1 \rangle \circ \langle T_2 \rangle$,
- $\langle \text{fix}(T) \rangle = \bigcup_n \langle T \rangle^n \cdot \emptyset$,
- $\langle \text{curry}(T) \rangle = \text{curry} \langle T \rangle$,
- $\langle \text{uncurry}(T) \rangle = \text{curry}^{-1} \langle T \rangle$,
- $\langle (U_1, U_2) \rangle = \{ c.i.v \mid cv \in \langle U_i \rangle \}$,
- $\langle \langle T_1, T_2 \rangle \rangle = \{ xc.i.v \mid xcv \in \langle T_i \rangle \}$,
where the operations on the right-hand side of equations are those of the category \( \text{gCDScont} \).

4.2. Operational semantics for CDS\(^*\)

We now present an operational semantics for CDS\(^*\) and show its equivalence with the denotational semantics (as stated without proof in \[21\]). The first definition is that of memo terms, that memorize parts of their computation.

**Definition 31.** States are memo terms. Couples, pairs, currying, uncurrying of memo terms are memo terms. If \( T \) and \( U \) are two memo terms and \( x \) is a state of the same type as \( U \), then \([T.U,x]\) is a memo term, where \( x \) is called a table and represents the current approximation of \( U \): a subset of its meaning. If \( T \) is a memo term and \( x \) is a state of the same type as the input to \( T \), then \([\text{fix}(T),x]\) is a memo term. If \( T \) and \( T' \) are memo terms and \( F \) is a set of pairs of states \((x,x')\) of the same types as the input to \( T \) and \( T' \), respectively, then \([T' \circ T,F]\) is a memo term. In this context, \( x \) is an approximation of the input of \( T \) and \( x' \) is an approximation of \( T' \)'s meaning applied to \( x \). CDS\(^*\) expressions are memo terms from which tables \( x \) and composition tables \( F \) have been removed.

We then define an inference system on a set of requests \( T?c \) and answers \( T!v \) where \( T \) is a memo term, \( c \) a cell of the same type and \( v \) a value. The operational semantics is distributed in the sense that every request/answer is associated with a cell, as thus with a process location. The parallel interpretation of a rule

\[
\frac{T''?c' \triangleright \cdots}{T?c \triangleright \cdots}
\]

is, upwards: request emanating from the process location of \( c \), downwards: reply from the process location of \( c' \). In the case of rule \([C3]\) below, requests are sent to many cells and the reply is collective.

In the rules \([AP1]\), \([AP2]\), table \( x \) represents the current approximation of the input \( U \). The union \( \sqcup \) is defined over memo terms whose pure-term parts are equal. Its result \( \sqcup_c U_c \) has the same pure-term part as its arguments \( U_c \) and as approximation-part, the union of approximation parts of \( U_c \)'s (and recursively so).

In the rules for composition of \( T' \circ T \), we have an input state \( x \) and a table \( F \) of pairs \((z,z')\) where \( z \) is a set of input events and \( z' \) a set of output events for \( T;z \) (i.e. input events for \( T' \)). From \( x \) and \( F \), two states \( y, y' \) are defined as follows. The first one, \( y \) is the set of input events found in \( F \) and relevant w.r.t \( x \) (i.e. element of \( x \)). The second one, \( y' \) is the set of output events found in \( F \) and relevant w.r.t. \( x \). More precisely, \( y = \bigcup\{z \mid \exists (z,z') \in F. z \subseteq x\} \) and \( y' = \bigcup\{z' \mid \exists (z,z') \in F. z \subseteq x\} \). By construction, for each pair \((z,z')\) of \( F \), \( z' \) represents an approximation of the input to \( T' \).

Rules \([AP2]\), \([F2]\), \([C2]\) and \([C3]\) are the only ones introducing sub-requests to be executed in parallel. These sub-requests are selected from the set \( A(x) \) of cells that are
accessible from $x$ (in order to follow access conditions).

$$cv \in x \quad x?c \triangleright x!v \quad [E] \quad \frac{T?c \triangleright T''?c}{T?c \triangleright T''!v} \quad [\text{TRN}]$$

$$\frac{T!v \triangleright (T_{1}, U)!!v \cdot (T_{1}, U)!!v}{(T, U)!v \triangleright (T_{1}, U)!v} \quad [\text{CL1}] \quad \frac{U!v \triangleright U_{1}!v}{(T, U)!v \triangleright (T, U_{1})!v} \quad [\text{CL2}]$$

$$\frac{T?!xc \triangleright T_{1}!v}{(T, U)!!x(c'{.}1) \triangleright (T_{1}, U)!v'} \quad [\text{P1}] \quad \frac{U?!xc \triangleright U_{1}!v'}{(T, U)!!x(c'{.}2) \triangleright (T, U_{1})!v'} \quad [\text{P2}]$$

$$T?(x, x')c'' \triangleright T_{1}!v'' \quad [\text{CUR}]$$

$$\frac{T?x(x'x'c'') \triangleright T_{1}!v''}{\text{uncurry}(T)!!x(x', x')c'' \triangleright \text{uncurry}(T_{1})!v''} \quad [\text{UNC}]$$

$$\frac{T?!xc \triangleright T_{1}!v'}{[T.U, x]c \triangleright [T_{1}.U, x]!v'} \quad [\text{AP1}]$$

$$\frac{U?!c_{1} \triangleright U_{1}!v_{1} \ldots U?!c_{n} \triangleright U_{n}!v_{n}}{[T.U, x]c \triangleright [T(U', x \cup y)]c} \quad [\text{AP2}]$$

where $\forall i = 1, \ldots, n$, $c_{i} \in A(x)$, $U' = \bigsqcup_{i} U_{i}$ and $y = \bigsqcup_{i} \{c_{i}v_{i}\}$.

$$\frac{T?!xc \triangleright T_{1}!v}{[\text{fix}(T), x]c \triangleright [\text{fix}(T_{1}), x]!v} \quad [\text{F1}]$$

$$\frac{T?!yc \triangleright T_{1}!v}{[T' \circ T, F]!!yc \triangleright [T' \circ T, F]!!v} \quad [\text{C1}]$$

$$\frac{T'?yc \triangleright T_{1}'!v \ldots T?!yc \triangleright T_{n}'!v}{[T' \circ T, F]!!yc \triangleright [T' \circ T, F \cup \{(y, y' \cup y')\}]!!yc} \quad [\text{C2}]$$

where $T_{1} = \bigsqcup_{i} T_{i}$, $y' = \bigsqcup_{i} \{c_{i}'v_{i}'\}$ and $\forall i = 1, \ldots, n$, $c_{i}' \in A(y')$.

$$\frac{x?c_{1} \triangleright x!v_{1} \ldots x?c_{n} \triangleright x!v_{n}}{[T' \circ T, F]!!yc \triangleright [T' \circ T, F \cup \{(y, y' \cup y')\}]!!yc} \quad [\text{C3}]$$

where $y_{1} = \bigsqcup_{i} \{c_{i}v_{i}\}$ and $\forall i = 1, \ldots, n$, $c_{i} \in A(y)$.

A full abstraction result holds for the language CDS∗ and its operational semantics: for every term $T$, $cv \in \llbracket T \rrbracket$ iff $\exists T' \hat{T}?c \triangleright T''!v$ where $\llbracket T \rrbracket$ is the denotational semantics of term $T$ and $\hat{T}$ is the memo term obtained from $T$ by setting all tables to the empty set. Its proof uses the following lemma:
**Lemma 32.** If $T ?c \triangleright T'!v$ or $T ?c \triangleright T''?c$ then $[T] = [T']$.

We prove adequacy of the denotational semantics with respect to the operational semantics:

**Proposition 33.** For every memo term $T$, the existence of a $T'$ such that $T ?c \triangleright T'!v$ implies $cv \in [T]$.

**Proof.** We prove this result by induction on the number $n$ of applications of transitivity rule [TRN] used in the derivation. Case $n = 0$.

By induction on the derivation.

- If the rule applied at root is [E], the derivation is $x?c \triangleright x!v$ which can happen only if $cv \in [x]$.

- Proofs of rules [CL1], [CL2], [P1], [P2], [CUR], [UNC], [AP1], [F1] and [C1] are similar (derivations with [AP2], [F2], [C2] or [C3] must use [TRN]). Let us consider the case [AP1]. We have

$$
\frac{T?x!v' \quad \triangleright \quad T_1!v'}{[T.U,x] ?c' \triangleright [T_1.U,x]!v'}.
$$

By induction hypothesis $xc'v' \in [T]$. Moreover $x \subseteq [U]$, so by definition of application, $c'v' \in [T].[U]$

Case $n > 0$. Before the first use of the rule [TRN], the derivation can only be composed of rule which have one and only one branch. So $T ?c \triangleright U'?!v'$ is derived from $U ?c' \triangleright U''!v'$, this derivation is a tree without branch and $T'?c' \triangleright T'!v'$ is such that

$$
\frac{U?c' \triangleright U'?c' \quad U'?c' \triangleright U''!v'}{U?c' \triangleright U''!v'}.
$$

By induction hypothesis on the number of [TRN], we obtain $c'v' \in [U']$ and Lemma 32 implies $[U] = [U']$. The case where no rule [TRN] is applied yields $cv \in [T]$. □

The opposite also holds.

**Proposition 34.** For every memo term $T, cv \in [T]$ implies the existence of a memo term $T'$ such that $T ?c \triangleright T'!v$. 
Proof. We prove this result by induction on term $T$.

- Let $cv \in x$. Then rule [E], gives

$$x?cv \triangleright x!v.$$  

- All rules which do not use tables can be treated in a similar way. For example, for the term \texttt{curry}(T) we have

$$x(x'c'')v'' \in [\text{curry}(T)] \text{ iff } (\Psi(x,x'))c''v'' \in [T]$$

(recall that $\Psi$ is the isomorphism between the domain of the product gcds’s and the product of the gcds domains). By induction hypothesis, we have

$$T?(x,x')c'' \triangleright T'!v''.$$  

We apply rule [CUR] and obtain

$$\text{curry}(T)?x(x'c'') \triangleright \text{curry}(T')!v''.$$  

- In the case of the application, a term of shape $T:U$, we have

$$c'v' \in [T,U] \text{ iff } \exists x \subseteq [U]. xc'v' \in [T].$$

Let us consider state $x$. By definition of exponential structure, it is a finite state. We define

$$x_0 = \emptyset, \quad x_{n+1} = \{cv \in x \mid \exists y:y \subseteq x_n \text{ and } y \in c\},$$

so that there exists a $N$ such that $x = x_N$, since $x$ is finite. By induction hypothesis we have for all $cv \in x_i$:

$$U?cv \triangleright U!v$$

and furthermore $F(x_{i+1}\setminus x_i) = A(x_i)$, so rule [AP2] can be applied to obtain for $i = 1,N$,

$$[T,U,x_i]?c' \triangleright [T,U_1,x_{i+1}]?c'. \quad (2)$$

Rule [TRN] gives

$$[T,U,\emptyset]?c' \triangleright [T,U_1,x]?c'$$

and we assumed above that $xc'v' \in [T]$. By induction hypothesis, $T?xc' \triangleright T_1!v'$.

Rule [AP1] gives

$$[T,U_1,x]?c' \triangleright [T_1,U_1,x]!v'.' \quad (3)$$
From derivations (2), (3) and rule [TRN], we conclude that

\[ [T,U,\emptyset]c' \triangleright T'!v'. \]

- If the term is of shape \( \text{fix}(T) \) or \( T' \circ T \), the proof is similar with differences in the construction of the table. \( \square \)

4.3. Programming with CDS\(^*\)

As was mentioned earlier, every cell in a gcds is intended to be located on a single processor throughout the program’s evaluation. This notion is made precise by the following definition.

**Definition 35.** Consider two distributed concrete structures

\[ M = (\langle C_M, \leq_M \rangle, V_M, E_M, \vdash_M, A_M) \quad N = (\langle C_N, \leq_N \rangle, V_N, E_N, \vdash_N, A_N). \]

The product \( M \times N \) is defined as the product of the two gcds’s, where the location function is given by

\[ A_{M \times N}(c) = \begin{cases} A_M(c) & \text{if } c \in C_M, \\ A_N(c) & \text{if } c \in C_N. \end{cases} \]

The exponential \( M \rightarrow N \) is defined as the exponential of the two gcds’s, with events located at their output:

\[ A_{M \rightarrow N}(xc) = A_N(c). \]

An array structure corresponds trivially to a distributed gcds. By this correspondence, \( \text{ADScont} \) is a subcategory of \( \text{gCDScont} \). Moreover, the locations given to the cells of product and exponential of array structures correspond to the locations given to the cells of product and exponential of distributed concrete structures. A key difference between the two categories is that the separated sum exists in \( \text{gCDScont} \). The separated sum of two distributed concrete structures \( M \) and \( N \) will be defined as the separated sum of the two underlying gcds’s. The location function of \( M + N \) is obvious for the cells of \( M \) and \( N \), but a choice has to be made for the selector. A separated sum is therefore parametrized by the location of the selector.

**Definition 36.** Consider the above two distributed concrete structures \( M \) and \( N \), and \( i \in I \) an index. The distributed concrete structure \( M +_i N \) is defined as the gcs \( M + N \) where the location function is given by

\[
\begin{align*}
A_{M +_i N}(c, L) &= A_M(c), \\
A_{M +_i N}(c, R) &= A_N(c), \\
A_{M +_i N}(Sel) &= i.
\end{align*}
\]
As a result, gCDS\text{cont} recovers all the array structures and solves the problem of sum types.

Appendix A defines the concrete syntax of CDS\textsuperscript{*} and gives examples of:

- Parallel AND functions.
- A “taster” function that analyzes their strictness.
- A “monitor” function that analyzes inter-processor dependences in other functions.

### 4.4. CDS\textsuperscript{*}, algorithmic skeletons and first-class schedules

The examples given in Appendix A demonstrate that the expressive power of CDS\textsuperscript{*} is sufficient to meet the objectives stated in Section 1. But given that those objectives are shared by many parallel language designers, one may ask: \textit{why define yet another semantic model for parallel languages?} The answer is that CDS\textsuperscript{*} is more general and flexible than existing paradigms for declarative parallel programming. In support of this view, we will now compare CDS\textsuperscript{*} with two other paradigms: algorithmic skeletons and explicit schedules.

Skeleton-based programming \cite{11} is theoretically equivalent to the use of \textit{externally-defined} parallel functions in a sequential language (and could be formalized by $\delta$-rules in the $\lambda$-calculus). Despite its lack of universality, this paradigm offers practical advantages and is the object of much current research \cite{24}. Skillicorn’s study of skeleton definitions with \textit{categorical data types} (CDTs) supports the view that they are a reasonable vehicle for high-level parallel programming. Yet he concludes \cite{26} that:

More work needs to be done on integrating different data type constructions into a single framework […] The other missing element is a framework for exploring implementations with the same level of formality as we use for exploring homomorphisms. Such frameworks exist only within limited environments, for example systolic arrays implemented on synchronous mesh architectures, but they need to be developed for richer computation styles.

Our study has shown that CDS\textsuperscript{*} supports a flexible style of deterministic parallel programming (broader than SPMD), higher-order functions, recursive types, sum types and a weak form of reflexivity. It is therefore a suitable implementation framework for CDTs and despite the unsolved problem of state syntax, defines the most general form of skeleton-programming to date.

As a final observation in favour of the CDS\textsuperscript{*} framework, we observe that its motivations are the same as those given by Mirani and Hudak for their \textit{explicit schedules} \cite{23}, namely declarative parallel programming with explicit elements of the evaluation strategy. The explicit schedules are (dynamic) program annotations describing the evaluation strategy for a functional program. A schedule is a partial order on events of two kinds: the demand ($d\;e$) for expression $e$’s evaluation and the waiting ($r\;e$) for the return of $e$’s value. Complex schedules are built from such events by independent parallel composition and fully-synchronous sequential composition. Moreover, the parallel functional language is given external system calls to monitor values like processor load: “resource-aware” programming. It should be clear from the above presentation
that CDS* generalizes explicit schedules w.r.t monitoring, while being more specific with respect to its events:

1. Values like processor load and communication volumes can be read from CDS* functions (i.e., functional states) before they are applied, removing the need for external system calls. In the words of Berry [2]: one may evaluate CDS* expressions of any type, including functional ones.

2. A CDS* event corresponds to a pair: demand and wait for an expression’s evaluation.

In fact, it appears that the technique of explicit schedules could be used to define complete and/or efficient subsets of the CDS* operational semantics. It is in that sense complementary to the use of concrete data structures.

5. Conclusion

Let us summarize the state of our knowledge about cds’s and parallel programming. Stable cds’s (or their sequential algorithms) are not appropriate for distributed implementations because replication of cds’s across explicit processes breaks stability. However, Brookes and Geva’s gcds’s are (trivially) compatible with distribution in the category of continuous functions and we have given them a distributed operational semantics à la CDS. It was also found that, to allow for sum types, the gcds’s should not be replicated in data-parallel fashion but given unconstrained event locations. When we attempt to distribute the more expressive parallel algorithms of Brookes and Geva (with a notion of local time, i.e., replicated clock structures) we face the following unsolved problem. Their algorithms are defined by an abstract co-monad construction with the goal of providing a theory of deterministic concurrent programs. But the concrete examples they give use a centralized notion of time and when we distribute both the generalized cds’s and the parallel algorithms, causality is lost. It remains to see whether notions of distributed algorithms can be designed with realistic implementations, i.e., without implicit remote causality.

From a programming point of view, our work is an attempt to isolate a theory of functional parallel programming, distinct from both concurrent programming (by its deterministic semantics) and from imperative parallel programming (by the inclusion of higher-order functions). Current state of the art in this area is that higher-order functions must be severely restricted [4] or that the size of data structures must be static [17] for efficient compilation. The results presented here should allow more flexible languages to attain similar and predictable performance. Problems remaining to be solved in this direction are the design of a polymorphic/implicit system of distributed concrete types, memory management and the definition of a practical syntax for state definitions.

To conclude let us note that from a historical perspective, it is natural that concrete data structures should be applied to the problem of functional parallel programming.
Given that declarative semantics is what makes functional languages attractive, but that parallelism is an operational property, the problem of full abstraction creates an apparent contradiction: how can a functional program prescribe its parallel evaluation if parallelism is a property of specific algorithms and not one of the function it denotes? The appropriate tools to analyze this question are those used by Berry, Curien, Brookes and Geva to study full abstraction, namely concrete structures and exponentials.

Acknowledgements

G. Hains and J. Mullins wish to thank Fujitsu Labs (Hains), ENS-Lyon (Hains and Mullins) and KTH Teleinformatik (Mullins) for visiting positions during different phases of this work. J. Mullins is supported by NSERC research grant no. 138321-99 from the Canadian Government and F. Loulergue has been supported by a doctoral scholarship from the French MENRT. Olivier Charloton has contributed to Section 3.3. We thank the anonymous referees and P.-L. Curien for their suggestions on improving the paper’s structure and presentation.

Appendix A. Example CDS* programs

We now demonstrate CDS* programming with small example programs executed by a sequential simulator (written in Caml) for the operational semantics. The syntax for states is much too simple for practical use and yet sufficient to demonstrate the expressive power of CDS* programs.

A.1. Syntax for types

A type is a set of cells associated with possible values. Cells are labelled with network addresses called indices ranging over a finite interval 1..N (N = 4 in the examples below). Those addresses correspond to the values given by function $\Lambda$ in a distributed geds. A type declaration is a list of cell declarations. The name of the cell and its network address is given after the keyword cell. The enumeration of the values the cell can take is given after the keyword values. A cell can be either activated by the empty set of events – in this case the keyword initial is used – or by one or more non-empty set of events. An event is written $C@I=V$ where $C$ is a name, $I$ a network address and $V$ a value. The sets of events are given by enumeration, every event is separated form the next one by a comma.

Two examples of type declarations follow. The first is an array of one boolean per index. The second is an array of cells that can be filled by only one value NON STRICT. This type is used by the TASTER program (see below).

```plaintext
type BOOL_ARRAY =
     begin
```
cell B@1 values T,F initial
cell B@2 values T,F initial
cell B@3 values T,F initial
cell B@4 values T,F initial
end;

type T_TASTER =
begin
cell TASTE@1 values NON_STRICT initial
cell TASTE@2 values NON_STRICT initial
cell TASTE@3 values NON_STRICT initial
cell TASTE@4 values NON_STRICT initial
end;

A.2. Syntax for states

In the basic syntax, states (functional or not) are given by enumeration of their events. The declaration of a state begins with the keyword let followed by the name of the state. After a colon the type of the state is given. A type expression is either a type name, the product of two type expressions (written *) or the exponential of two type expression (written ->). The events of the state are enumerated after an equal sign. Events are separated by commas and the enumeration of events are surrounded by brackets. This syntax is also used for cells of functional states. For a first-order function, an event has the form \{events enumeration\}C@I=V.

In the following examples we use an extended syntax : indices and values can be replaced respectively by index variables and value variables whose identifier begins with a &. The range of these variables are given at the end of the state declaration after the keyword where. A range has the form \&v in [enumeration of values].

A.3. Parallel AND and function TASTER

Our first example function is a parallel AND that performs the conjunction of the booleans of the array and returns the result on each index:

let AND : BOOL_ARRAY->BOOL_ARRAY =
{ {B@&l1=F}B@&l2=F;
{B@1=T,B@2=T,B@3=T,B@4=T}B@&l2=T }
where \&l1 in [1,2,3,4] \&l2 in [1,2,3,4];

The following function also performs a conjunction but is strict on the first index:

let AND_S1 : BOOL_ARRAY->BOOL_ARRAY =
{ {B@1=F}B@&l1=F;
{B@1=T,B@&l2=F}B@&l1=F;
{B@1=T,B@2=T,B@3=T,B@4=T}B@&l1=T }
where \&l1 in [1,2,3,4] \&l2 in [2,3,4];
The taster function takes as argument a function on arrays of booleans and returns an array of cells indicating for each index whether the argument function is non-strict on this index. This illustrates the language’s reflexivity.

```plaintext
let TASTER: (BOOL_ARRAY->BOOL_ARRAY)->T_TASTER =
  {{ {B@01=&v1}B@&l1=&v2 } TASTE@&l1=NON_STRICT;
    { {B@02=&v1}B@&l1=&v2 } TASTE@&l2=NON_STRICT;
    { {B@03=&v1}B@&l1=&v2 } TASTE@&l3=NON_STRICT;
    { {B@04=&v1}B@&l1=&v2 } TASTE@&l4=NON_STRICT;
  { {B@01=&v1,B@02=&v2}B@&l1=&v3 }TASTE@&l12=NON_STRICT;
  { {B@01=&v1,B@03=&v2}B@&l1=&v3 }TASTE@&l13=NON_STRICT;
  { {B@01=&v1,B@04=&v2}B@&l1=&v3 }TASTE@&l14=NON_STRICT;
  { {B@02=&v1,B@03=&v2}B@&l1=&v3 }TASTE@&l123=NON_STRICT;
  { {B@02=&v1,B@04=&v2}B@&l1=&v3 }TASTE@&l124=NON_STRICT;
  { {B@03=&v1,B@04=&v2}B@&l1=&v3 }TASTE@&l134=NON_STRICT;
  { {B@01=&v1,B@02=&v2,B@03=&v3}B@&l1=&v4 }TASTE@4=NON_STRICT;
  { {B@01=&v1,B@02=&v2,B@04=&v3}B@&l1=&v4 }TASTE@3=NON_STRICT;
  { {B@02=&v1,B@03=&v2,B@04=&v3}B@&l1=&v4 }TASTE@1=NON_STRICT;
  { {B@01=&v1,B@02=&v3,B@03=&v4}B@&l1=&v4 }TASTE@2=NON_STRICT}
```

where

\&v1 in \{T,F\} \&v2 in \{T,F\} \&v3 in \{T,F\} \&v4 in \{T,F\}
\&l in \{1,2,3,4\} \&l1 in \{2,3,4\} \&l2 in \{1,3,4\}
\&l3 in \{1,2,4\} \&l4 in \{1,2,3\} \&l12 in \{3,4\}
\&l13 in \{2,4\} \&l14 in \{2,3\} \&l123 in \{1,4\}
\&l124 in \{1,3\} \&l134 in \{1,2\};

Keyword \texttt{eval} is used to request the evaluation of the state expression given. Application is written “.”.

```plaintext
eval TASTER.AND;
eval TASTER.AND_S1;
```

The simulator returns the following results:

Evaluation
----------

\texttt{(TASTER.AND)=\{TASTE@4=NON\_STRICT, TASTE@3=NON\_STRICT,}
\texttt{TASTE@2=NON\_STRICT, TASTE@1=NON\_STRICT\}}
\texttt{(TASTER.AND\_S1)=\{TASTE@4=NON\_STRICT, TASTE@3=NON\_STRICT,}
\texttt{TASTE@2=NON\_STRICT\}}

### A.4. A longer example: the communication monitor

We will design here a CDS* program reading functional dependences in a given family of functions, and producing thus an upper bound on the number of communications their evaluation would require. This property of the language is one of its main advantages for parallel programming.
We will use the following notations: \( \#(x) \) is the cardinality of set \( x \) and \( <_M \) is the Brookes-Geva order on \( M \)'s cells, where \( M \) is a distributed gcds. As before \( A_M(c) \) is the process index of cell \( c \) in \( M \). Consider a state \( a \) of type \( M \rightarrow N \) and a state \( x \) of type \( M \). The problem reduces to the computation of an upper bound on the communications necessary for evaluating the application \( a.x \) for any \( x \).

### A.4.1. Hypotheses on the language's implementation

We will assume that evaluation of cell \( c' \) in \( a.x \) starts from \( A(c') \) and, knowing events \( ye'v' \) of \( a \) (which are by definition co-located) attempts to establish \( y \subseteq x \). We write \( \text{comm}(yc') \) for our estimation (upper bound) of the number of communications generated in this manner while evaluating cell \( c' \) of \( a.x \). It is necessary to specify \( y \) in \( \text{comm}(yc') \) because state \( a \) may contain a cell \( zc'v' \) for which \( z \notin y \) and \( y \notin z \) and whose evaluation may produce fewer or more communications than \( yc' \). There are two cases for defining the value of \( \text{comm}(yc') \), depending on whether \( c' \) is initial in \( N \).

1. **comm(yc') for \( c' \) an initial cell of \( N \)**

   Consider \( yc'v' \in a \) such that
   
   (a) \( y \subseteq x \)
   (b) \( \forall y_1c_1v_1 \in a, \ yc' \leq_M N y_1c_1 \)

   Hypothesis (b) means that cell \( yc' \) is filled in the set of events generating state \( a \). It thus indicates a minimum input to fill cell \( c' \) in \( a.x \). We then define
   
   \[
   \text{comm}(yc') = \#(\{cv \in y | A_M(c) \neq A_N(c')\})
   \]

   because the events \( cv \) in the above set are those whose location \( A_N(c') \) must be queried before deciding that \( y \subseteq x \) and concluding \( c'v' \in a.x \). This measure does not account for communications leading to events in the enabling of \( yc' \). It should therefore be applied by induction over the enabling structure.

2. **comm(yc') for \( c' \) an non-initial cell of \( N \)**

   Let \( Y'' \) be the set of enablings of \( yc' \) in \( x \) and \( \{y_1c'_1v'_1, \ldots, y_kc'_k\} \subseteq Y'' \). The input events already known at \( A(c') \) are the events in those sets \( y_i \) for which \( A(c_i) = A(c') \). Let us write \( y_{dc} \) for that set of events.

   \[
   y_{dc} = \bigcup_{i \in \{1, \ldots, k\} | A(c_i) = A(c')} y_i
   \]

   We then define
   
   \[
   \text{comm}(yc') = \#(\{cv \in y | A_M(c) \neq A_N(c')\}) - \min_{Y''} #(y_{dc})
   \quad \text{(A.1)}
   \]

   because certain cells whose presence in \( x \) is sought are already known by process \( A(c) \). Here again, the measure does not account for communications leading to events in the enabling of \( yc' \). It is an estimation of the communications needed to evaluate \( c' \) in \( a.x \) once it has been activated.

As a consequence of definition (A.1), \( \text{comm}(yc') \) is an approximation. It is impossible to know which enablings of \( Y'' \) have been used to activate \( yc' \), because our gcds
are not deterministic. Some communications may therefore refer to redundant enablings and lead \( \text{comm}(yc') \) to overestimate the number of communications generated by the implementation.

**A.4.2. Functions to monitor**

Let us now define the family of functions whose communications are to be monitored. We will write \( \langle e_0, \ldots, e_{p-1} \rangle \) for the parallel state containing state \( e_0 \) at location 0, ..., state \( e_{p-1} \) at location \( p - 1 \). A list containing states \( e_0, \ldots, e_k \) at a given location will be written \([e_0, \ldots, e_k]\). The functions to monitor have the following properties:

- They take as input \( \langle l_0, \ldots, l_{p-1} \rangle \): a list at each location.
- They produce as output at each location a representation of the queue of incoming messages there: an array of length \( p - 1 \) containing lists received from the other processes:

\[
\langle [l_{0,1}, \ldots, l_{0,p-1}], [l_{1,0}, l_{1,2}, \ldots, l_{1,p-1}], \ldots, [l_{p-1,0}, \ldots, l_{p-1,p-2}] \rangle.
\]

The output arrays are of length \( p - 1 \) because array at location \( i \) does not contain an element at index \( i \). We are only interested in dependences between processors and intra-dependences are useless.

- The concatenation of all output lists with a given second index (origin location) is a sub-list of the input list at that (origin) location:

\[
\forall i, \bigcirc_{j \neq i} l_{j,i} \subseteq l_i
\]

where \( \bigcirc \) denotes concatenation.

In other words, the functions to monitor move a (possibly empty) subset \( l_{j,i} \), of every input list \( l_i \) to another location \( j \).

To keep the example tractable in our enumerative syntax we will use:

- lists of length at most 3 built from only one value \( A \). Their gcds is:

```plaintext
type LIST =
begin
  cell L0 values A,NIL initial
  cell L1 values A,NIL access L0=A
  cell L2 values A,NIL access L1=A
  cell L3 values NIL access L2=A
end
```

The input type \( \text{LIST3} \) is made of a copy of \( \text{LIST} \) on every processor, with local enablings only. Define \( k + 1 \) to be the height of cell \( L_k \).

- two processors only: output arrays of lists are then of length 1 and their type is therefore isomorphic to the input type.

```plaintext
type DATAIN = LIST3;
type DATAOUT = LIST3;
```

We will also assume that

- the functions output correct lists, i.e. with an end-of-list marker \( \text{NIL} \).
• if a function contains event \{..., cv ...,\}c′v′ then it must also contain an event \{..., cv₁ ...,\}c′v′₁ for every possible value v₁ of cell c.
• if the value of an output cell c' depends on an input cell of height k then the function may only be strict on cells of height at most k (w.r.t. the evaluation of c').

The above constraints limit the combinatorial explosion of the monitor.

A.4.3. The monitor

The monitor takes as input one of the functions a specified above and returns a lower bound on the estimation comm: the maximal number of communications generated by an application a:x for every output location. It uses the following type for its output:

```plaintext
type NAT3 =
begin
  cell N0 values 0,1,2,3 initial
  cell N1 values 0,1,2,3 initial
end;
```

The monitor itself is written by case enumeration and in two halves, one about messages reaching processor 0 and the other about messages reaching processor 1:

```plaintext
let H : (DATAIN->DATAOUT) -> NAT3 = {

{ {}L0@1=NIL }N@01=0,
  { {L0@0=NIL}L0@1=NIL,
    {L0@0=A}L0@1=A,
    {L0@0=A}L1@1=NIL }N@01=1,

  { {L0@0=NIL}L0@1=NIL,
    {L0@0=A}L0@1=A,
    {L0@0=A,L1@0=NIL}L1@1=NIL,
    {L0@0=A,L1@0=A}L1@1=A,
    {L0@0=A,L1@0=A}L2@1=NIL }N@01=2,

  { {L0@0=A,L1@0=NIL}L0@1=NIL,
    {L0@0=A,L1@0=A}L0@1=A,
    {L0@0=A,L1@0=A}L1@1=NIL }N@01=2,

  { {L0@0=A,L1@0=A,L2@0=NIL}L2@1=NIL,
    {L0@0=A,L1@0=A,L2@0=A}L2@1=A,
    {L0@0=A,L1@0=A,L2@0=A}L2@1=A,
    {L0@0=A,L1@0=A,L2@0=A}L2@1=A,
```
{L0@0=A,L1@0=NIL,L2@0=NIL,L3@0=NIL }N0=0,
{L0@0=A,L1@0=A,L2@0=NIL,L3@0=NIL }N0=1,
{L0@0=A,L1@0=A,L2@0=NIL,L3@0=A }N0=2,
{L0@0=A,L1@0=A,L2@0=A,L3@0=A }N0=3,
Here are three examples of functions to monitor. Without loss of generality, they only communicate from location 1 to location 0. Their output list at location 1 is always empty.

The first one, HASH1, copies any list of height at most 2 and therefore depends on at most two cells: L0@1 and L1@1. Its communication volume towards location 0 will be at most 2.

let HASH1:DATAIN->DATAOUT={
{L0@1=NIL,
 {L0@1=&v}L0@0=&v,
 {L0@1=A,L1@1=&v}L1@0=&v,
 {L0@1=A,L1@1=A}L2@0=NIL }
} where &v in [A,NIL];

The second function communicates a list of height $k$ (at most 3) with its top elements truncated. It therefore depends on at most 3 cells.

let HASH2:DATAIN->DATAOUT={
{L0@1=NIL,
 {L0@1=A,L1@1=&v}L0@0=&v,
 {L0@1=A,L1@1=A,L2@1=&v}L1@0=&v,
 {L0@1=A,L1@1=A,L2@1=A}L2@0=NIL }
} where &v in [A,NIL];

The third function copies the whole list from location 1 to location 0. It therefore depends on 3 cells.

let HASH3:DATAIN->DATAOUT={
{L0@&l=NIL,
 {L0@1=&v}L0@0=&v,
 {L0@1=A,L1@1=&v}L1@0=&v,
 {L0@1=A,L1@1=A,L2@1=&v}L2@0=&v,
 {L0@1=A,L1@1=A,L2@1=&v}L3@0=NIL }
} where &v in [A,NIL];

The evaluation requests

```
 eval H.HASH1; eval H.HASH2; eval H.HASH3;
```

produce the expected results (through the CDS* simulator):

```
type LIST3 defined.
type DATAIN defined.
type DATAOUT defined.
type NAT3 defined.
value H : ( (DATAIN -> DATAOUT) -> NAT3) defined.
value HASH1 : (DATAIN -> DATAOUT) defined.
```
value HASH2 : (DATAIN -> DATAOUT) defined.
value HASH3 : (DATAIN -> DATAOUT) defined.

Evaluation
~~~~~~~~~~
(H.HASH1)=\{N@0=2,N@1=0\}
(H.HASH2)=\{N@0=3,N@1=0\}
(H.HASH3)=\{N@0=3,N@1=0\}

and this completes the example.

References

[20] B. Lisper, Data parallelism and functional programming, in: École de Printemps sur le Data-Parallélisme,
Lengauer, M. Griebl, S. Gorlatch (Eds.), Euro-Par’97 Parallel Processing, Lecture Notes in Computer
Euro-Par’96 Parallel Processing, Lecture Notes in Computer Science, vol. 1123–1124, Lyon, LIP-ENSL,
Programming Languages and Computer Architecture, FPCA’95, La Jolla, CA, ACM Press, 1995.
Y. Robert (Eds.), Euro-Par’96 Parallel Processing, Lecture Notes in Computer Science, vol. 1123–1124,
Advances in Petri nets; Proceedings of an Advanced Course, Bad Honnef, 8–19 September 1986,