A Linear Logic Model of State

Uday S. Reddy
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
Net: reddy@cs.uiuc.edu

Janurary 8, 1993

Abstract

We propose an abstract formal model of state manipulation in the framework of Girard's linear logic. Two issues motivate this work: how to describe the semantics of higher-order imperative programming languages and how to incorporate state manipulation in functional programming languages. The central idea is that a state is linear and "regenerative", where the latter is the property of a value that generates a new value upon each use. Based on this, we define a type constructor for states and a "modality" type constructor for regenerative values. Just as Girard's "of course" modality allows him to express static values and intuitionistic logic within the framework of linear logic, our regenerative modality allows us to express dynamic values and imperative programs within the same framework. We demonstrate the expressiveness of the model by showing that a higher-order Algol-like language can be embedded in it.

1 Introduction

Programming in the real world typically involves state-manipulation. All the major programming languages widely in use, except for "pure functional" languages like Haskell, support variables and assignments. A large number of programming applications ranging from window managers to process control systems involve the manipulation of state. Unfortunately, there are relatively few mathematical tools available to the programmer to reason about such programs and applications. The object of this paper to propose a model of state inspired by linear logic, which can serve as the foundation for building mathematical theories of state manipulation.

Linear logic has, from the beginning, evoked ideas of state. Girard [Gir87b] and Lafont [Laf88] give example applications for modelling state via linearity. Wadler has explicitly proposed linearity as a means for representing state in functional languages [Wad90, Wad91]. However, one finds that many aspects of state-manipulation are not modelled by these proposals. How does one model the identity of a history-sensitive object which persists over a period of time and supports operations for its use? How does one model sequencing? How does one model hierarchical abstractions where new objects are constructed from existing objects?

To build a model of state using linear logic ideas, we found it necessary to make two extensions to linear logic. One is a binary connective called "before" which was already proposed by Girard, and studied extensively in Retoré's dissertation [Ret93b]. The second is a "regenerative" storage operator (or modality) which stands in the same relation to "before" as the "of course" operator does to tensor. By extending intuitionistic linear logic with these two constructions, we obtain a logical system dubbed the "linear logic model of state".

While our own intuitions are derived from this logical system, for pedagogical clarity, we present its semantics first. In Section 2, we define the two constructions in the context of coherent spaces. Section 3 is a semantical study of the Kleisli category of the regenerative storage operator. We show that the maps of this category correspond to identifiable classes of functions. Next, we show that these constructions model state-manipulation by interpreting a fragment of Algol using them (Section 4). Finally, we present the logical system itself in Section 5.

2 Two Constructions in Coherent Spaces

For modelling state and sequential computations, we need two constructions in addition to those of linear logic. The first is a binary connective called "before" (written "$\triangleright$") which denotes sequential composition of components. The second is a "regenerative" storage operator (written "$\dagger$") which allows us to build sequentially reusable storage objects. In this section, we use the framework of Girard's coherent spaces [GLT89, Gir87a] to give concrete semantic intuitions about these constructions.

We assume a general familiarity with coherent spaces. See [GLT89, Chaps. 8 and 12] for basic notions and [Gir87a, Sec. 3] for coherent semantics of linear logic. (We refer to the "classical" features of this semantics for pedagogical reasons even though they are not essential for the model of state presented here. A reader is comfortable with intuitionistic linear logic may consult Appendix A for a quick review of the classical features.) We also mention various categorical facts in the passing. See [Mac71, Laf88, See89, Jac92] for definitions. First, recall the basic definition of coherent spaces.

2.1 Definition A *coherent space* is a pair $A = (|A|, \frown\!\!\!\!\smile_A)$ where

- $|A|$ is a set, whose members are called *tokens*, and
- $\frown\!\!\!\!\smile_A$ is a binary reflexive relation on $|A|$, called *coherence*.

We write $\alpha \frown\!\!\!\!\smile_A \alpha'$ as $\alpha \frown\!\!\!\!\smile \alpha'$ [mod A] for convenience (or just $\alpha \frown\!\!\!\!\smile \alpha'$, depending on context). Define notations:

$$\begin{array}{lrcl}
\text{(strict coherence)} & \alpha \frown \alpha' \ [\text{mod } A] & \iff & \alpha \frown\!\!\!\!\smile \alpha' \ [\text{mod } A] \wedge \alpha \neq \alpha' \\
\text{(strict incoherence)} & \alpha \smile \alpha' \ [\text{mod } A] & \iff & \neg\, \alpha \frown\!\!\!\!\smile \alpha' \ [\text{mod } A] \\
\text{(incoherence)} & \alpha \asymp \alpha' \ [\text{mod } A] & \iff & \alpha \smile \alpha' \ [\text{mod } A] \vee \alpha = \alpha' \\
 & & \iff & \neg\, \alpha \frown \alpha' \ [\text{mod } A]
\end{array}$$

A *linear map* $F : A \multimap B$ is a subset of $|A| \times |B|$ such that, for all $(\alpha, \beta), (\alpha', \beta') \in F$,

$$\alpha \frown\!\!\!\!\smile \alpha' \Rightarrow \beta \frown\!\!\!\!\smile \beta' \ \wedge \ \alpha \frown \alpha' \Rightarrow \beta \frown \beta'$$

Coherent spaces and linear maps form a category **COHL**.

An *element* (or *point*) of A is a coherent set in A, *i.e.*, a subset $a \subseteq |A|$ such that, for all $\alpha, \alpha' \in a$, $\alpha \supset\!\!\!\subset \alpha'$ [mod A]. The elements of A form an atomic Scott domain under inclusion order.[1] We denote this domain by $\mathcal{D}(A)$.[2] A liner map $F : A \multimap B$ determines a function $f : \mathcal{D}(A) \to \mathcal{D}(B)$ given by $f(a) = \{\, \beta : \exists \alpha \in a.\, (\alpha, \beta) \in F \,\}$. The function f is *stable*, *i.e.*, preserves meets of pairs of compatible elements:

$$x \uparrow y \Longrightarrow f(x \sqcap y) = f(x) \sqcap f(y) \tag{1}$$

and *linear*, *i.e.*, preserves the lubs of bounded sets:

$$S \uparrow \Longrightarrow f(\bigsqcup S) = \bigsqcup f(S) \tag{2}$$

Henceforth, we will call linear, stable functions simply "linear functions". Note that a linear function is continuous and strict. Conversely, given a linear function $f : \mathcal{D}(A) \to \mathcal{D}(B)$, we can recover F (called the *trace* of f) by

$$F = \{\, (\alpha, \beta) : \alpha \in |A|,\ \beta \in f(\{\alpha\}) \,\}.$$

It can be shown that the category **COHL** is equivalent to the category of atomic Scott domains and linear functions.

We call a linear map a *linear injection* if its underlying relation $f \subseteq |A| \times |B|$ is an injection. Linear injections are monomorphisms in **COHL** (among others).

Before

2.2 Recall the connectives $\otimes$ and $⅋$:

$$\begin{array}{l} |A \otimes B| = |A| \times |B| \\ \qquad (\alpha, \beta) \supset\!\!\!\subset (\alpha', \beta') \text{ [mod } A \otimes B] \iff \alpha \supset\!\!\!\subset \alpha' \text{ [mod } A] \wedge \beta \supset\!\!\!\subset \beta' \text{ [mod } B] \\ |A ⅋ B| = |A| \times |B| \\ \qquad (\alpha, \beta) \frown (\alpha', \beta') \text{ [mod } A ⅋ B] \iff \alpha \frown \alpha' \text{ [mod } A] \vee \beta \frown \beta' \text{ [mod } B] \end{array}$$

Notice that

$$\begin{array}{lcl} (\alpha, \beta) \supset\!\!\!\subset (\alpha', \beta') \text{ [mod } A \otimes B] & \Longrightarrow & (\alpha, \beta) \supset\!\!\!\subset (\alpha', \beta') \text{ [mod } A ⅋ B] \\ (\alpha, \beta) \frown (\alpha', \beta') \text{ [mod } A \otimes B] & \Longrightarrow & (\alpha, \beta) \frown (\alpha', \beta') \text{ [mod } A ⅋ B] \end{array}$$

This implies a linear injection $\mathbf{mix} : A \otimes B \multimap A ⅋ B$.

2.3 We can now postulate another connective $\triangleright$ ("before") with linear injections:

$$A \otimes B \multimap A \triangleright B \multimap A ⅋ B.$$

[1] An *atom* is an element x such that $x' \sqsubseteq x$ implies $x' = \bot \vee x' = x$. A domain is *atomic* if every element is the lub of the set of atoms below it.

[2] Girard [GLT89] calls $\mathcal{D}(A)$ a coherent space and calls A the *web* of the coherent space.

There is only possible definition of this new connective:

$$
\begin{aligned}
&|A \rhd B| = |A| \times |B| \\
&\quad (\alpha, \beta) \frown (\alpha', \beta') \; [\text{mod } A \rhd B] \iff \\
&\qquad \alpha \frown \alpha' \; [\text{mod } A] \vee \alpha = \alpha' \wedge \beta \frown \beta' \; [\text{mod } \beta]
\end{aligned}
$$

Contrast the coherence tables for the three connectives:

$A \otimes B$ ([mod A] rows, [mod B] columns)

	$\frown$	$=$	$\smile$
$\frown$	$\frown$	$\frown$	$\smile$
$=$	$\frown$	$=$	$\smile$
$\smile$	$\smile$	$\smile$	$\smile$

$A \rhd B$ ([mod A] rows, [mod B] columns)

	$\frown$	$=$	$\smile$
$\frown$	$\frown$	$\frown$	$\frown$
$=$	$\frown$	$=$	$\smile$
$\smile$	$\smile$	$\smile$	$\smile$

$A \parr B$ ([mod A] rows, [mod B] columns)

	$\frown$	$=$	$\smile$
$\frown$	$\frown$	$\frown$	$\frown$
$=$	$\frown$	$=$	$\smile$
$\smile$	$\frown$	$\smile$	$\smile$

We denote the two linear injections by **ser** : $A \otimes B \multimap A \rhd B$ and **par** : $A \rhd B \multimap A \parr B$.

2.4 What does this new connective mean? First, recall the standard linear logic intuitions. $A \otimes B$ denotes independent composition and $A \parr B$ denotes composition with (possible) communication.[3] $A \rhd B$ is "in between" the two. It denotes communication with possible *one way* communication: from the A component to the B component. The injection $A \otimes B \multimap A \rhd B$ should be understood as saying, if we have a pair of computations with no communication between the components, we have a pair with possible one-way communication. Similarly, $A \rhd B \multimap A \parr B$ means: if we have a pair of computations with one-way communication then we have a pair with some communication.

2.5 The following properties immediately follow from the definition. "Before" is associative and has **1** as its unit:

$$
\begin{aligned}
\textbf{sassl} : A \rhd (B \rhd C) \quad &\cong \quad (A \rhd B) \rhd C : \textbf{sassr} \\
\textbf{sdell} : \mathbf{1} \rhd A \quad &\cong \quad A : \textbf{sinsl} \\
\textbf{sdelr} : A \rhd \mathbf{1} \quad &\cong \quad A : \textbf{sinsr}
\end{aligned}
$$

It can be verified that the category **COHL** obtains a monoidal structure from $\rhd$ and **1**. However, notice that it is not symmetric: $A \rhd B \ncong B \rhd A$.

2.6 "Before" is its own dual:

$$(A \rhd B)^{\perp} \quad = \quad A^{\perp} \rhd B^{\perp}$$

That means, the production as well as the consumption of before-pairs is carried out in the same fashion: with possible communication from the left to right.

[3] The fact that communication is not mandatory signifies that coherent spaces interpret linear logic together with the MIX rule.[Gir87a, p. 99]

2.7 Define a construction $\multimap\!\!\triangleright$ by $A \multimap\!\!\triangleright B = A^\perp \triangleright B$. Its coherence relation may be stated as

$$\begin{array}{l}(\alpha, \beta) \frown (\alpha', \beta') \, [\mathrm{mod}\, A \multimap\!\!\triangleright B] \iff \\ \qquad \alpha \frown \alpha' \, [\mathrm{mod}\, A] \Rightarrow \alpha = \alpha' \wedge \beta \frown \beta' \, [\mathrm{mod}\, B]\end{array}$$

It is easily verified that there is a linear injection **par** : $(A \multimap\!\!\triangleright B) \multimap (A \multimap B)$. Consider the elements (coherent sets) of $A \multimap\!\!\triangleright B$. These may be seen as certain kinds of functions from elements of A to elements of B, called *sequential* functions.[4] A sequential function extracts a single token from its input to produce a token of output. However, it cannot wait for a "demand" on the output to extract the input token. For example, the identity function on *int* is a sequential function, but the identity function on $A \,\&\, B$ is not. $((0, \alpha), (0, \alpha))$ and $((1, \beta), (1, \beta))$ are not coherent in $A \,\&\, B \multimap\!\!\triangleright A \,\&\, B$. Two strictly coherent inputs such as $(0, \alpha)$ and $(1, \beta)$ cannot be handled in a sequential function because the choice between them can only be made based on an external demand for the output.

Whenever A is a *discrete* coherent space (that is, $\alpha \frown \alpha' \Rightarrow \alpha = \alpha'$), every linear function $A \multimap B$ is a sequential function. Note that discrete coherent spaces include primitive spaces like *int* and *bool* and are closed under $\otimes$ and $\oplus$. Discrete coherent spaces with sequential functions form a category.

If D is a discrete coherent space, for every $F : D \triangleright A \multimap B$, we have

$$\begin{array}{rcllcl}\mathbf{scurry}\, F & : & A \multimap (D \multimap B) & & = & \{ (\alpha, (\delta, \beta)) : ((\delta, \alpha), \beta) \in F \} \\ \mathbf{sapply} & : & D \triangleright (D \multimap B) \multimap B & & = & \{ ((\delta, (\delta, \beta)), \beta) : \delta \in |D|, \, \beta \in |B| \}\end{array}$$

such that $\mathbf{sapply} \circ (\mathrm{id} \triangleright (\mathbf{scurry}\, F)) = F$ and $\mathbf{scurry}\, F$ is unique.

2.8 The above features of $\triangleright$ show that it has a curious hybrid behavior: it behaves a little like $\otimes$ as well as $⅋$. The fact that $A^\perp \triangleright B$ is a function space is of much significance in our modelling of state. A computation of type $A^\perp \triangleright B$ first computes the $A^\perp$ component—extracting in the process some information about an A-typed value—and then potentially uses this information to produce the B component. This captures some of the behavior of storage cells. For example, the sequential function $id : int^\perp \triangleright int$ models two steps of a "buffer" that stores an integer using the $int^\perp$ component and retrieves it using the *int* component. The recursive type:

$$C = \mathbf{1} \,\&\, (int^\perp \triangleright int \triangleright C)$$

models buffers that repeat this process indefinitely.

2.9 The hybrid behavior of $\triangleright$ gives rise to a number of curious properties. It distributes over both $\oplus$ and $\&$ in the first argument position:

$$\begin{array}{rcl}(A \oplus B) \triangleright C & \cong & (A \triangleright C) \oplus (B \triangleright C) \\ (A \,\&\, B) \triangleright C & \cong & (A \triangleright C) \,\&\, (B \triangleright C)\end{array}$$

[4] These are not related to the notion of sequential functions in the sense of Kahn and Plotkin. We use the term "sequential" to mean that subcomputations are sequenced rather than to mean the absence of parallelism.

Contrast this with the fact that $\otimes$ only distributes over $\oplus$ and $⅋$ only over $\&$:

$$\begin{array}{rcl} (A \oplus B) \otimes C & \cong & (A \otimes C) \oplus (B \otimes C) \\ & \cong & (C \otimes A) \oplus (C \otimes B) \quad \cong \quad C \otimes (A \oplus B) \\ (A \& B) ⅋ C & \cong & (A ⅋ C) \& (B ⅋ C) \\ & \cong & (C ⅋ A) \& (C ⅋ B) \quad \cong \quad C ⅋ (A \& B) \end{array}$$

We also have various weak distributivity properties:

$$\begin{array}{rcl} A \otimes (B \rhd C) & \multimap & (A \otimes B) \rhd C \\ (A \rhd B) \otimes C & \multimap & A \rhd (B \otimes C) \\ (A ⅋ B) \rhd C & \multimap & A ⅋ (B \rhd C) \\ A \rhd (B ⅋ C) & \multimap & (A \rhd B) ⅋ C \end{array}$$

These properties are best understood in terms of allowed communication paths:

$\otimes$	no communication
$\rhd$	communication left to right
$⅋$	communication in both directions

2.10 Let $F(A_1, \ldots, A_n)$ be a construction involving $\otimes$ and $\rhd$. We can define a partial order $\leq_F$ on the set $\{1, \ldots, n\}$ such that $i \leq j$ iff F allows communication from A_i to A_j (*i.e.*, A_i and A_j occur to the left and right, respectively, of a $\rhd$ connective). Then, we have a linear injection $F(A_1, \ldots, A_n) \multimap G(A_1, \ldots, A_n)$ whenever $\leq_F$ is included in $\leq_G$. This allows one to formulate a sequent calculus with sequents treated as partial orders of proposition occurrences [Ret93b, Ret93a].

2.11 We briefly mention the categorical properties at play here, which seem necessary for building a model of state. We have a symmetric monoidal closed category $(\mathbf{C}, \otimes, \mathbf{1}, \multimap)$ with an additional monoidal structure $(\rhd, \mathbf{1})$ such that the primary monoidal structure $(\otimes, \mathbf{1})$ is a "submonoidal structure" of $(\rhd, \mathbf{1})$, *i.e.*, there is a natural monic $\mathbf{ser} : _ \otimes _ \rightarrow _ \rhd _$ that preserves the associated structure (**assl**, **dell** and **delr**).

Call an object D *discrete* if

$$\mathrm{Hom}(D \rhd A, B) \quad \cong \quad \mathrm{Hom}(A, D \multimap B)$$

The corresponding combinators are denoted **scurry** : $\mathrm{Hom}(D \rhd A, B) \rightarrow \mathrm{Hom}(A, D \multimap B)$ and **sapply** : $D \rhd (D \multimap B) \rightarrow B$. It follows that the tensor unit $\mathbf{1}$ is discrete.

2.12 Acknowledgements The idea of a "before" connective is said to be originally due to Girard. C. Retoré has done an extensive study of linear logic extended with "before" [Ret93b, Ret93a]. I am indebted to S. Abramsky for pointing me to the earlier work.

Regenerative storage

2.13 Next, we define a *regenerative* storage operator "†" in the same spirit of Girard's "of course" operator:

$$
\begin{aligned}
|\dagger A| &= |A|^* \qquad \text{(finite sequences over } |A|) \\
s \frown t \ [\text{mod} \dagger A] \iff\ & s \text{ prefix } t \lor \\
& t \text{ prefix } s \lor \\
& s = s_0 \cdot \langle\alpha\rangle \cdot s' \land t = s_0 \cdot \langle\beta\rangle \cdot t' \land \alpha \frown \beta \\
& \quad \text{for some } s_0, s', t' \in |A|^* \text{ and } \alpha, \beta \in |A|
\end{aligned}
$$

A somewhat "cleaner" statement of the coherence condition is:

$$
\begin{aligned}
&\langle \alpha_1, \ldots, \alpha_n \rangle \frown \langle \alpha'_1, \ldots, \alpha'_m \rangle \ [\text{mod} \dagger A] \iff \\
&\quad \forall k \leq \min(n, m),\ (\forall i < k,\ \alpha_i = \alpha'_i) \Longrightarrow \alpha_k \frown \alpha'_k \ [\text{mod } A]
\end{aligned}
$$

Following Hoare [Hoa85], we call the tokens of $\dagger A$ " traces". A *trace* denotes the information extracted from a storage object in one particular execution of a program. The object itself is then modelled as the set of traces it supports. The coherence condition states that two traces are coherent if, at the first point of divergence between them, if any, they differ coherently.

2.14 The elements of $\dagger A$ are coherent sets of traces. It is useful to focus attention on the prefix-closed elements of $\dagger A$, called *trace sets.* A trace set S can be represented as a (potentially infinite) tree T, with arcs labelled by tokens of A, such that

$$s \text{ is a path of } T \iff s \in S$$

The nodes of the tree may be interpreted as the "states" of a storage object and the arcs as possible "state transitions". The coherence condition states that, in every state, the set of available transitions must be coherent.

2.15 Examples

(i) Consider an "integer stepper" object that successively steps through the integers. Its behavior is captured by the set of traces:

$$\langle\rangle,\ \langle 0\rangle,\ \langle 0, 1\rangle,\ \ldots$$

This set is an element of $\dagger\mathit{int}$. Its tree representation is shown in Fig. 1(a).

(ii) Consider a "counter" object that stores an integer value and supports operations "fetch" and "increment". It is modelled by an element of the coherent space $\dagger(\mathit{int} \,\&\, \mathbf{1})$ where the two components *int* and $\mathbf{1}$ model the two operations. Note that the choice between the two operations is *external.* The tokens of $\mathit{int} \,\&\, \mathbf{1}$ are, as usual, $(0, n)$ for each integer n and $(1, *)$. For readability, we write these tokens as *fetch*.n and *inc*.$*$. The traces of the counter object include

$$
\begin{aligned}
&\langle\rangle \\
&\langle \mathit{fetch}.0\rangle,\ \langle \mathit{inc}.*\rangle \\
&\langle \mathit{fetch}.0,\ \mathit{inc}.*\rangle,\ \langle \mathit{inc}.*,\ \mathit{fetch}.1\rangle,\ \langle \mathit{fetch}.0,\ \mathit{fetch}.0\rangle,\ \langle \mathit{inc}.*,\ \mathit{inc}.*\rangle \\
&\langle \mathit{fetch}.0,\ \mathit{inc}.*,\ \mathit{fetch}.1\rangle,\ \ldots
\end{aligned}
$$

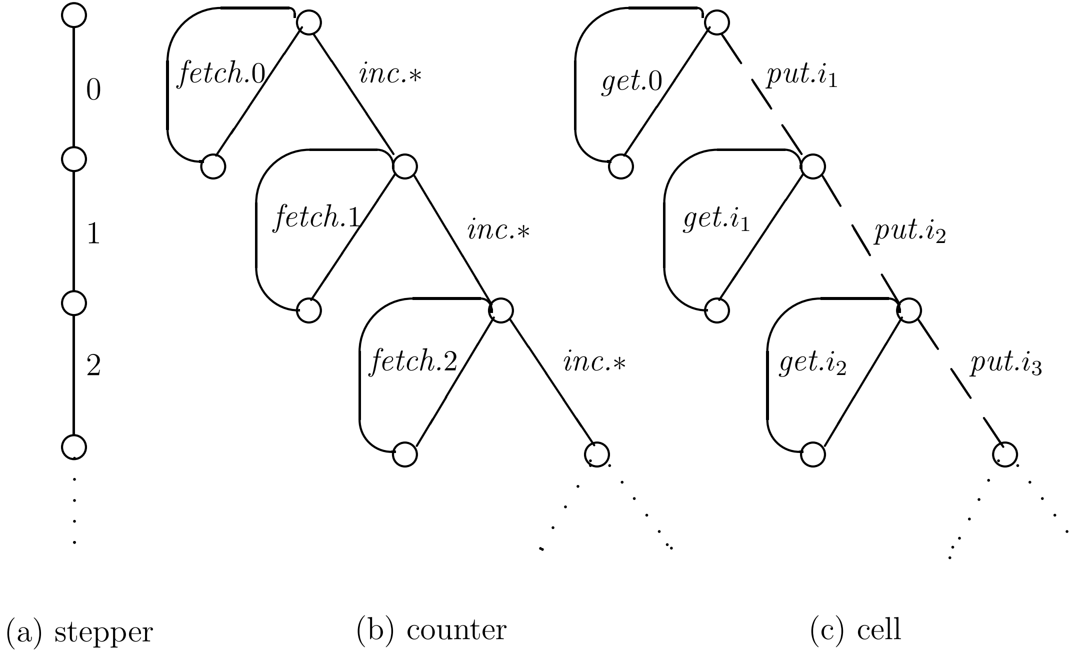

Figure 1: Trace sets as trees

The behavior tree corresponding to this trace set is shown in Fig. 1(b). (The back arc is a meta-level notation to say that its source node has the same set of transitions as its target node.)

(iii) A storage cell for integers can be modelled as an element of $\dagger(int \,\&\, int^{\perp})$. Write the tokens of $int \,\&\, int^{\perp}$ as $get.i$ and $put.j$ respectively. Then, an integer cell has the behavior shown in Fig. 1(c). (The arcs with labels $put.i_k$ represent an infinite number of outgoing arcs, one for each value of i_k. The arcs labelled $get.i_k$, however, represent single arcs.) We denote the trace set of a cell with initial value i by $cell_i$ and use $cell$ for $\bigcup_{i \in \omega} cell_i$.

It is natural to ask if we can have storage cells of any type A. Unfortunately, such cells give rise to difficulties of the kind mentioned in 2.7. A cell that holds values of type $int \,\&\, int$ has, among its traces,

$$\langle put.(0, i), get.(0, i)\rangle \quad \text{and} \quad \langle put.(1, j), get.(1, j)\rangle$$

The two put tokens are incoherent: $(0, i) \frown (1, j)$ [mod $(int \,\&\, int)$] and, so, $(0, i) \smile (1, j)$ [mod $(int \,\&\, int)^{\perp}$]. Thus, our model allows only storage cells that hold values of discrete types. We interpret this negative result as follows: Since a storage cell is limited to a sequential behavior, storing a value in it involves providing the entire information about the value "in one shot". Only discrete-typed values can be provided in this fashion.

It is possible that this is merely a limitation of the coherent space model which may be circumvented in a more sophisticated model. But, more likely, storing non-discrete-typed values in cells is an inherently complex operation and our understanding of such behavior is still very limited. Note also that, one could simulate the effect of storing complex objects by storing "names" or "references" to such objects.

2.16 It may be verified that there are linear maps:

$$\begin{array}{rcll} \textbf{done} & : & \dagger A \multimap \mathbf{1} & = \{(\langle\rangle, *)\} \\ \textbf{seq} & : & \dagger A \multimap \dagger A \rhd \dagger A & = \{ (s \cdot t, (s,t)) : s, t \in |A|^* \} \end{array}$$

and they make $\dagger A$ a comonoid in the monoidal structure $\langle \mathbf{COHL}, \rhd, \mathbf{1} \rangle$.

There are also linear maps:

$$\begin{array}{rcll} \textbf{sread} & : & \dagger A \multimap A & = \{ (\langle \alpha \rangle, \alpha) : \alpha \in |A| \} \\ \textbf{Seq} & : & \dagger A \multimap \dagger\dagger A & = \{ (s_1 \cdot \ldots \cdot s_n, \langle s_1, \ldots, s_n \rangle) : s_1, \ldots, s_n \in |A|^* \} \end{array}$$

such that $\langle \dagger, \mathbf{sread}, \mathbf{Seq} \rangle$ is a comonad.

Intuitively, the map **done** closes the trace of an object. The map **seq** obtains two traces from a given trace: one that can be used immediately and another that can be used in future (after the first trace is closed). So, **seq** allows a storage object to be sequentially reused. The map **sread** extracts one component of a trace and closes it, while the map **Seq** splits a trace into a trace of traces whose components can again be used only sequentially.

2.17 There is a monomorphism

$$\mathbf{Ser} : !A \multimap \dagger A \quad = \quad \{ (\mathit{comp}(s), s) : s \in |A|^*, \ \mathit{comp}(s) \in |!A| \}$$

where $!A$ is as defined in [GLT89, Gir87a] and $\mathit{comp}(\langle \alpha_1, \ldots, \alpha_n \rangle) = \{\alpha_i : 1 \leq i \leq n\}$ is the set of "components" of a trace. This may seem surprising at first because for each coherent set $a \in |!A|$, there are many enumerations of its members forming tokens of $|\dagger A|$. However, notice that all such enumerations are coherent in $\dagger A$. In particular, for any down-closed family X of coherent sets,

$$\mathit{Ser}_A(X) \quad = \quad \{ s : \mathit{comp}(s) \in X \} \quad = \quad \bigcup_{a \in X} a^*$$

is a trace set. Such trace sets have the property that whenever $s \in S$, every trace t with the same set of components as s is also in S. Viewed in terms of trees, they are seen to have the same set of transitions at every node. For example, the counter and cell trace sets shown in Fig. 1 have this property when restricted to *fetch* and *get* transitions respectively.

Normally, such trace sets denotes sequences of "passive observations" of a storage object which do not modify the internal state of the object. Since the state is not modified, the sequence in which the observations are made is insignificant. For this reason, we call them *passive trace sets*.

2.18 Definition In general, given a category with the structure mentioned in 2.11, we require two monoidal comonads ! and † with the former being a sub-comonad of the latter via a comonad monomorphism $\mathbf{Ser} : ! \rightarrow \dagger$. Further, $!A$ must be a comonoid with respect to $(\otimes, \mathbf{1})$ and $\dagger A$ a comonoid with respect to $(\rhd, \mathbf{1})$. Since $(\otimes, \mathbf{1})$ is a submonoidal structure of $(\rhd, \mathbf{1})$, $!A$ also becomes a comonoid with respect to $(\rhd, \mathbf{1})$, via the monic **ser**. This comonoid must be a "subcomonoid" of $\dagger A$ via the natural monic **Ser** which must now be a morphism of comonoids as well.

It is easy to verify that the above definitions satisfy these requirements. The only detail not covered previously is the fact that † is a "monoidal" comonad. This means, first, that † is a monoidal functor, with maps:

$$\begin{array}{rcl}
\mathbf{striv} & : & \mathbf{1} \multimap \dagger\mathbf{1} \\
 & = & \{\, (*, \langle * \rangle^n) : n \in \omega \,\} \\
\mathbf{sprom} & : & \dagger A \otimes \dagger B \multimap \dagger(A \otimes B) \\
 & = & \{\, ((\langle \alpha_1, \ldots, \alpha_n \rangle, \langle \beta_1, \ldots, \beta_n \rangle),\ \langle (\alpha_1, \beta_1), \ldots, (\alpha_n, \beta_n) \rangle) \\
 & & \quad : n \in \omega,\ \forall i < n,\ \alpha_i \in |A|, \beta_i \in |B| \,\}
\end{array}$$

satisfying certain coherence conditions [Koc72, Jac92]. Further, the natural transformations **sread** and **Seq** are monoidal transformations.

To this, we add the usual requirements [See89] that the category have finite products and coproducts, and that ! maps the the product monoidal structure $(\&, \top)$ to $(\otimes, \mathbf{1})$. (This implicitly makes ! a monoidal comonad.) The resulting structure is called an *LLMS-category* (for Linear Logic Model of State).

3 Representation results

An important property of Girard's "of course" storage operator is that its co-Kleisli category gives stable functions, *i.e.*, there is an order isomorphism

$$\mathcal{D}(!A \multimap B) \quad \cong \quad [\mathcal{D}(A) \to_s \mathcal{D}(B)]$$

where $\to_s$ denotes the stable function space. In this section, we look at corresponding results for the regenerative storage operator.

For a coherent space A, define its *active domain* $\mathcal{A}(A)$ to be the subdomain of $\mathcal{D}(\dagger A)$ consisting of only prefix-closed elements. As mentioned in 2.14, such elements are equivalently viewed as trees. We have the following order isomorphisms:

$$\begin{array}{rcl}
\mathcal{A}(\dagger A \multimap B) & \cong & [\mathcal{A}(A) \to_l \mathcal{A}(B)] \\
\mathcal{D}(\dagger A \multimap B) & \cong & [\mathcal{A}(A) \to_r \mathcal{A}(B)]
\end{array}$$

where $\to_l$ denotes the linear function space (cf. 2.1) and $\to_r$ denotes a subspace of linear functions consisting of so-called *regular* functions. While the elements of $\mathcal{D}(\dagger A \multimap B)$ are linear maps, the elements of $\mathcal{A}(\dagger A \multimap B)$ are not. (We call them *active* maps.) It is surprising that they should correspond to a linear function space! From a practical point of view, active maps correspond to procedures with side effects. In spite of such side effects, the procedures satisfy β equivalence in Algol [Rey81] and other related languages [SRI91, ORH93, PW93]. The first isomorphism above provides some insight into why this is so.

The utility of these results is that they allow us to view linear and active maps as functions. Unfortunately, it is not yet clear whether good representation results exist in the category of coherent spaces (atomic domains). We move to the larger class of dI-domains to obtain the representations.

3.1 Notation We use the term "trace" to refer to a finite sequence over some set and "trace set" to refer to a prefix-closed set of traces. The meta-variables $s, t, \ldots$ stand for traces, $X, Y, \ldots$ for sets of traces and $S, T, \ldots$ for trace sets.

The prefix relation on traces is denoted by "$\leq$". Define

$$\begin{array}{rcl} \downarrow t & = & \{ s : s \leq t \} \\ \downarrow X & = & \{ s : \exists t \in X,\ s \leq t \} \\ \max(T) & = & \{ t \in T : \neg \exists t' \in T,\ t < t' \} \\ T/t & = & \{ t' : t \cdot t' \in T \} \\ T \cdot T' & = & T \cup \{ t \cdot t' : t \in \max(T) \wedge t' \in T' \} \end{array}$$

Think of trace sets as trees, as mentioned in 2.14. Then $\max(T)$ is the set of maximal paths of the tree T. T/t is the subtree arrived at by following the path t. $T \cdot T'$ is the tree obtained by grafting a copy of T' at every leaf node of T. If there are no leaf nodes in T, then $T \cdot T' = T$.

Define the relations:

$$\begin{array}{rcl} t \sim t' & \Leftrightarrow & t \not\leq t' \wedge t' \not\leq t \\ T \sim T' & \Leftrightarrow & \forall t \in \max(T),\ \forall t' \in \max(T'),\ t \sim t' \end{array}$$

When $T \sim T'$ we say that T and T' "diverge".

3.2 Definition Define $\mathcal{A}(A) = \{ T : T \in \mathcal{D}(\dagger A),\ T = \downarrow T \}$, ordered by inclusion. $\mathcal{A}(A)$ satisfies the following properties:

- down-closure: if $T \in \mathcal{A}(A)$ and $T' \subseteq T$ is a trace set then $T' \in \mathcal{A}(A)$.
- coherence: if $X \subseteq \mathcal{A}(A)$ and $\forall T_1, T_2 \in X$, $T_1 \cup T_2 \in \mathcal{A}(A)$ then $\bigcup X \in \mathcal{A}(A)$.
- suffix completeness: if $T \in \mathcal{A}(A)$ and $t \in T$ then $T/t \in \mathcal{A}(A)$.
- extension completeness: if $T_1, T_2 \in \mathcal{A}(A)$ such that $T_1 \sim T_2$ and $T_1 \cup T_2 \in \mathcal{A}(A)$ then, for all $T_1', T_2' \in \mathcal{A}(A)$ such that $T_1 \cdot T_1'$, $T_2 \cdot T_2' \in \mathcal{A}(A)$, $(T_1 \cdot T_1') \cup (T_2 \cdot T_2') \in \mathcal{A}(A)$.
- freely generated: $\langle\rangle \in \mathcal{A}(A)$ and $T, T' \in \mathcal{A}(A) \Longrightarrow T \cdot T' \in \mathcal{A}(A)$.

In fact, these properties completely characterize *free active domains*, *i.e.*, if D is any domain of trace sets satisfying the above properties, then there is a coherent space A such that $D = \mathcal{A}(A)$. A domain of trace sets that only satisfies the first four properties is called an *active domain*. Such a domain may not contain all the trace sets of the corresponding $\mathcal{A}(A)$.

Thinking of trace sets as trees, down-closure says chopping some branches in a tree of $\mathcal{A}(A)$ gives another tree in $\mathcal{A}(A)$. Coherence says a set of trees can be merged if its members can be merged pairwise. Suffix completeness says the subtree at a path t is also a tree of $\mathcal{A}(A)$. Extension completeness, by far the most complex condition, says if $T_1 \cup T_2$ is a tree with divergent subtrees T_1 and T_2, then extending T_1 with some T_1' and extending T_2 with some T_2' still gives a tree in $\mathcal{A}(A)$. However, such extension is permissible for only certain valid extensions of T_1 and T_2 respectively. If all extensions are valid, we obtain free active domains.

3.3 An active domain is a Scott domain. Further, it is

- prime algebraic, *i.e.*, every element is the lub of the set of complete primes below it,[5] and
- finitary, *i.e.*, every compact element is approximated by a finite number of elements.

As shown in [Win80], finitary prime-algebraic domains are the same as Berry's dI-domains [Ber78]. Active domains are, in addition, coherent.

The complete primes of $\mathcal{A}(A)$ are trace sets of the form $\downarrow t$ for some trace t. Equivalently, they are trees with no branching. Prime algebraicity amounts to saying that a tree is the set of its paths. Finatariness means that a tree with a maximal path has a finite number of paths.

3.4 Definition We consider two kinds of functions between active domains. Linear functions between active domains, denoted $D \to_l E$, are stable functions satisfying (2). The linear function space with Berry order is denoted $[D \to_l E]$. In addition, we consider *regular* functions, denoted $D \to_r E$. These are linear functions such that, whenever $s \in S$ is a minimal trace such that $t \in f(\downarrow s)$,

$$t \cdot t' \in f(S) \iff t' \in f(S/s) \tag{3}$$

3.5 Examples

(i) Consider a function $f : \mathcal{A}(int) \to \mathcal{A}(int)$ such that

$$f(T) \quad = \quad \{\, \langle i_1, i_1 + i_2, \ldots, \Sigma_{k=1}^{n} i_k \rangle : n \in \omega,\ \langle i_1, \ldots, i_n \rangle \in T \,\}$$

The function is evidently stable and linear. However, it is not regular. For instance, $\langle 1 \rangle \in f(\downarrow\langle 1 \rangle)$ and $\langle 1, 3 \rangle \in f(\downarrow\langle 1, 2 \rangle)$, but $\langle 3 \rangle \notin f(\downarrow\langle 2 \rangle)$. Linear functions between active domains are, in general, history-sensitive (have "side-effects"). To implement f in a programming language, we would need an internal storage cell that remembers the current sum and gets modified each time f is "called". For this reason, we call linear functions between active domains "active functions".

(ii) In contrast, regular functions involve no internal memory. A simple example is $inc : \mathcal{A}(int) \to \mathcal{A}(int)$ given by

$$inc(T) \quad = \quad \{\, \langle i_1 + 1, \ldots, i_n + 1 \rangle : \langle i_1, \ldots, i_n \rangle \in T \,\}$$

For every "demand" on its output object, this function demands an integer from its input object and returns the incremented integer. The implementation of *inc* does not involve any memory.

(iii) Another example of a regular function is $evens : \mathcal{A}(int) \to \mathcal{A}(int)$ given by

$$evens(T) \quad = \quad \{\, \text{the sequence of even integers in } s : s \in T \,\}$$

This function (possibly) demands several integers from its input object to produce an integer of the output object. However, it is still regular. Since regular functions do not involve internal memory, we also call them "passive functions".

The representation results that follow demonstrate the active/passive nature of these functions.

[5] A *complete prime* is an element x such that, for all sets S of elements, $x \sqsubseteq \bigsqcup S \Longrightarrow \exists y \in S,\ x \sqsubseteq y$. In a finitary domain, complete primes are the elements with unique predecessors [Win80, Win87].

3.6 Lemma *Let $f : \mathcal{A}(A) \to_l \mathcal{A}(B)$ be a linear function, $S \in \mathcal{A}(A)$ and $t \in f(S)$. Then,*
 (i) *it is possible to find finite $S_0 \subseteq S$ such that $t \in f(S_0)$, and*
 (ii) *if S_0 is chosen minimal among the solutions to* (i)*, then $S_0 = \downarrow s$ for some unique trace $s \in S$.*

Proof (i) follows from continuity of f. For (ii), we obtain uniqueness of S_0 from the stability of f. Linearity of f gives that S_0 is a complete prime. □

This result, and the one that follows, are similar to Girard's [GLT89, Sec. 8.5 and 12.3]. See also [Zha92, Zha93].

3.7 Lemma *Given $f : \mathcal{A}(A) \to_l \mathcal{A}(B)$, define $\mu f \subseteq |A|^* \times |B|^*$ by*

$$\mu f = \{ (s, t) : \downarrow s \textit{ is a minimal trace set such that } t \in f(\downarrow s) \}$$

Then
 (i) $\mu f \in \mathcal{D}(\dagger A \multimap \dagger B)$,
 (ii) $(\langle\rangle, \langle\rangle) \in \mu f$, *and*
 (iii) *if $(s, t) \in \mu f$ and $t' \leq t$ then $(s', t') \in \mu f$ for some $s' \leq s$.*

Proof The verification of (i) is straightforward. (ii) follows from the fact that $\langle\rangle \in f(S)$ for all nonempty S. For (iii), note that $t' \in f(\downarrow s)$ and, so, there must be a shortest $s' \in \downarrow s$ such that $t' \in f(\downarrow s')$. □

The properties (ii) and (iii) mean, in particular, that if $(s, \langle \beta_1, \ldots, \beta_n \rangle) \in \mu f$ then there is a decomposition $s = s_1 \cdot \ldots \cdot s_n$ such that, for all $i \leq n$, $(s_1 \cdot \ldots \cdot s_i, \langle \beta_1, \ldots, \beta_i \rangle) \in \mu f$.

Note that trace set domains are needed for this result. A similar property is not available for linear functions $f : \mathcal{D}(\dagger A) \to_l \mathcal{D}(\dagger B)$.

3.8 Lemma *For $f : \mathcal{A}(A) \to_l \mathcal{A}(B)$, define $\psi f \subseteq (|A|^* \times |B|)^*$ by*

$$\begin{aligned} \psi f \;=\; \{ & \langle (s_1, \beta_1), \ldots, (s_n, \beta_n) \rangle : \\ & n \in \omega,\ \forall i = 1, \ldots, n,\ (s_1 \cdot \ldots \cdot s_i,\ \langle \beta_1, \ldots, \beta_i \rangle) \in \mu f \} \end{aligned}$$

Then $\psi f \in \mathcal{A}(\dagger A \multimap B)$.

Proof ψf is clearly prefix closed. We show that it is coherent in $\dagger(\dagger A \multimap B)$. Suppose

$$\langle (s_1, \beta_1), \ldots, (s_n, \beta_n) \rangle \sim \langle (s'_1, \beta'_1), \ldots, (s'_m, \beta'_m) \rangle \in \psi f \tag{4}$$

Let $k \leq \min(n, m)$ be an index such that $(s_i, \beta_i) = (s'_i, \beta'_i)$ for all $i < k$. By definition,

$$(s_1 \cdot \ldots \cdot s_k, \langle \beta_1 \ldots, \beta_k \rangle),\ (s'_1 \cdot \ldots \cdot s'_k, \langle \beta'_1, \ldots, \beta'_k \rangle) \in \mu f$$

Since μf is coherent in $\dagger A \multimap B$, $s_1 \cdot \ldots \cdot s_k \supset s'_1 \cdot \ldots \cdot s'_k$ implies $\langle \beta_1, \ldots, \beta_k \rangle \supset \langle \beta'_1, \ldots, \beta'_k \rangle$. So, $s_k \supset s'_k \Longrightarrow \beta_k \supset \beta'_k$ and $s_k \frown s'_k \Longrightarrow \beta_k \frown \beta'_k$, *i.e.*, $(s_k, \beta_k) \supset (s'_k, \beta'_k)$ [mod $\dagger A \multimap B$]. The pair of sequences in (4) is thus coherent in $\dagger(\dagger A \multimap B)$. □

3.9 Lemma *If $F \in \mathcal{A}(\dagger A \multimap B)$, there exists a linear function $\phi F : \mathcal{A}(A) \to_l \mathcal{A}(B)$ given by*

$$\phi F(S) = \{\, \langle \beta_1, \ldots, \beta_n \rangle : \exists s_1 \cdot \ldots \cdot s_n \in S,\ \langle (s_1, \beta_1), \ldots, (s_n, \beta_n) \rangle \in F \,\}$$

Proof We first verify that $\phi F(S) \in \mathcal{A}(B)$. Suppose $t = \langle \beta_1, \ldots, \beta_n \rangle$ and $t' = \langle \beta'_1, \ldots, \beta'_m \rangle$ are in $\phi F(S)$. There exist $s_1 \cdot \ldots \cdot s_n$ and $s'_1 \cdot \ldots \cdot s'_m$ in S such that

$$\langle (s_1, \beta_1), \ldots, (s_n, \beta_n) \rangle,\ \langle (s'_1, \beta'_1), \ldots, (s'_m, \beta'_m) \rangle \in F$$

Let $k \leq \min(n, m)$ be an index such that $(s_i, \beta_i) = (s'_i, \beta'_i)$ for all $i < k$. Then, $(s_k, \beta_k) \frown (s'_k, \beta'_k)$ [mod $\dagger A \multimap B$]. Also, because $s_i = s'_i$ for all $i < k$, $s_k \frown s'_k$ [mod $\dagger A$]. From these facts, we conclude (i) $\beta_k \frown \beta'_k$ [mod B], and (ii) $\beta_k = \beta'_k$ implies $s_k = s'_k$.

Suppose $l \leq \min(n, m)$ is an index such that $\beta_i = \beta'_i$ for all $i < l$. From the above, $s_i = s'_i$ for all $i < l$ and, hence, $\beta_l \frown \beta'_l$ [mod B]. This shows that $t \frown t'$ [mod $\dagger B$].

Suppose $t = t'$. Then, $s_1 \cdot \ldots \cdot s_n = s'_1 \cdot \ldots \cdot s'_m$. So, whenever $t \in \phi F(S)$, there exists a shortest $s \in S$ such that $t \in \phi F(\downarrow s)$. This shows that ϕF is stable and linear. □

3.10 Theorem *There is an order isomorphism $\mathcal{A}(\dagger A \multimap B) \cong [\mathcal{A}(A) \to_l \mathcal{A}(B)]$.*

Proof It may be verified that the maps ψ and ϕ defined above are mutually inverse. To check that these maps are monotone, notice the equivalences

$$f \sqsubseteq_B g \iff \mu f \subseteq \mu g \iff \psi f \subseteq \psi g$$

where $\sqsubseteq_B$ stands for the Berry order. The first equivalence is standard. (Cf. [GLT89, Sec. 8.5.3] and [Zha93].) The second equivalence can be verified easily. We show the implication right to left. Suppose $\psi f \subseteq \psi g$ and let $(s, \langle \beta_1, \ldots, \beta_n \rangle) \in \mu f$. By Lemma 3.7, there is a decomposition $s = s_1 \cdot \ldots \cdot s_n$ such that, for all $i = 1, \ldots, n$, $(s_1 \cdot \ldots \cdot s_i, \langle \beta_1, \ldots, \beta_i \rangle) \in \mu f$. This means $\langle (s_1, \beta_1), \ldots, (s_n, \beta_n) \rangle \in \psi f \subseteq \psi g$, and, by inverting the argument, we obtain $(s, \langle \beta_1, \ldots, \beta_n \rangle) \in \mu g$. □

3.11 Example For the function f mentioned in 3.5, ψf includes all traces of the form:

$$\langle\, (\langle i_1 \rangle, i_1),\ (\langle i_2 \rangle, i_1 + i_2),\ \ldots,\ (\langle i_n \rangle, \Sigma_{k=1}^{n} i_k) \,\rangle$$

for $n \in \omega$ and $i_1, \ldots, i_n \in |int|$.

3.12 Lemma *If $f : \mathcal{A}(A) \to_r \mathcal{A}(B)$ is a regular function, there exists $F \in \mathcal{D}(\dagger A \multimap B)$ such that $\psi f = F^*$.*

Proof The condition (3) means that, if $(s, t) \in \mu f$ then $(s \cdot s', t \cdot t') \in \mu f \iff (s', t') \in \mu f$. Define $F = \{\, (s, \beta) : \langle (s, \beta) \rangle \in \psi f \,\}$. Equivalently, $F = \{\, (s, \beta) : (s, \langle \beta \rangle) \in \mu f \,\}$. Now,

$$\begin{aligned} &\langle (s_1, \beta_1), \ldots, (s_n, \beta_n) \rangle \in \psi f \\ &\qquad \iff \text{for } i = 1, \ldots, n,\ (s_1 \cdot \ldots \cdot s_i, \langle \beta_1, \ldots, \beta_i \rangle) \in \mu f \\ &\qquad \iff \text{for } i = 1, \ldots, n,\ (s_i, \langle \beta_i \rangle) \in \mu f \\ &\qquad \iff \text{for } i = 1, \ldots, n,\ (s_i, \beta_i) \in F \end{aligned}$$

which shows that $\psi f = F^*$. □

3.13 Theorem *There is an order isomorphism* $\mathcal{D}(\dagger A \multimap B) \;\cong\; [\mathcal{A}(A) \rightarrow_r \mathcal{A}(B)]$.

Proof Given $f : \mathcal{A}(A) \rightarrow_r \mathcal{A}(B)$, let $\pi f \in \mathcal{D}(\dagger A \multimap B)$ be as in Lemma 3.12:

$$\pi f = \{\, (s, \beta) : \langle (s, \beta) \rangle \in \psi f \,\}$$

We can invert the mapping by associating, with each $F \in \mathcal{D}(\dagger A \multimap B)$, the function

$$f(S) = \{ \langle \beta_1, \ldots, \beta_n \rangle : \exists s_1 \cdot \ldots \cdot s_n \in S, \text{ for } i = 1, \ldots, n, \, (s_i, \beta_i) \in F \,\}$$

These mappings are monotone because, for regular f and g, $\pi f \subseteq \pi g \iff \psi f \subseteq \psi g$ and the latter is equivalent to $f \sqsubseteq_B g$ as in 3.10. □

3.14 Example

(i) For the function *inc* mentioned in 3.5, ψinc has traces of the form:

$$\langle\, (\langle i_1 \rangle, i_1 + 1), \ldots (\langle i_n \rangle, i_n + 1) \,\rangle$$

Note that $\psi f = \{\, (\langle i \rangle, i + 1) : i \in |int| \,\}^*$.

(ii) For the function *evens*, $\pi evens$ has pairs of the form $(s \cdot \langle i \rangle, i)$ where s is a sequence of odd integers and i is an even integer.

4 Interference-controlled Algol

We would like to give a semantic account of programming languages and design proof systems using the ideas presented in Section 2. We consider programming languages first so as to provide concrete computational intuitions. A proof system called "linear logic model of state" is presented in Section 5.

Algol 60 [Nau60] is one of the earliest and most influential programming languages. Reynolds [Rey81] clarified the essential design of Algol 60 as a typed lambda calculus with primitive types for state-manipulation. We refer to Reynolds's presentation as *Idealized Algol* or, simply, *Algol*. An important criticism of Algol and other Algol-like languages is that they permit uncontrolled interference between active (state-manipulating) objects which makes reasoning about programs complex. From our point of view, semantic analysis of these languages is also made complicated by this feature. Reynolds [Rey78] proposed a system for syntactically controlling interference using the principle: "distinct identifiers do not interfere".[6] We refer to this system as *interference-controlled Algol*. O'Hearn [O'H91] studied the connections between interference control methods used by Reynolds and the resource control implicit in linear logic. Our semantics builds on O'Hearn's work. Reynolds also proposed in improved interference control system in [Rey89] using a sophisticated subtype discipline. While we believe that the semantics of this system also falls within our framework, we relegate its study to future work.

[6] Other languages that use some form of interference control include Concurrent Pascal [Bri73], Euclid [Pop77], Turing [HMRC87], Occam [PM87] and FX [GL86].

$$\frac{}{\Gamma \vdash 0 : \mathbf{exp}} \qquad \frac{\Gamma \vdash e : \mathbf{exp}}{\Gamma \vdash \mathsf{succ}\ e : \mathbf{exp}} \qquad \frac{\Gamma \vdash e_1 : \mathbf{exp} \quad \Gamma \vdash e_2 : \mathbf{exp}}{\Gamma \vdash e_1 + e_2 : \mathbf{exp}}$$

$$\frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{comm}} \qquad \frac{}{\Gamma \vdash \mathbf{diverge} : \mathbf{comm}}$$

$$\frac{\Gamma \vdash c_1 : \mathbf{comm} \quad \Gamma \vdash c_2 : \mathbf{comm}}{\Gamma \vdash (c_1; c_2) : \mathbf{comm}} \qquad \frac{\Gamma \vdash c_1 : \mathbf{comm} \quad \Delta \vdash c_2 : \mathbf{comm}}{\Gamma, \Delta \vdash (c_1 \parallel c_2) : \mathbf{comm}}$$

$$\frac{\Gamma, x : \mathbf{var} \vdash c : \mathbf{comm}}{\Gamma \vdash \mathbf{new}\ x.\, c : \mathbf{comm}} \qquad \frac{\Gamma \vdash v : \mathbf{var} \quad \Gamma \vdash e : \mathbf{exp}}{\Gamma \vdash v := e : \mathbf{comm}} \qquad \frac{\Gamma \vdash v : \mathbf{var}}{\Gamma \vdash v : \mathbf{exp}}$$

Figure 2: Sample primitive phrases

Primitive types

4.1 Definition Let δ range over primitive data types, such as **int** and **bool**. Then the primitive *types* of Algol (sometimes called "phrase types") are as follows:

$$\theta \quad ::= \quad \delta\ \mathbf{exp} \mid \mathbf{comm} \mid \delta\ \mathbf{var}$$

Intuitively,

- δ **exp** stands for "expressions" (passive state-observers) that yield values of type δ,
- **comm** stands for "commands" that modify state, and
- δ **var** stands for "variables" (storage cells) that store values of type δ.[7]

To simplify matters, we consider a single data type **int**, and abbreviate **int exp** and **int var** to **exp** and **var** respectively.

Sample program phrases of these types are shown in Fig. 2. The phrase $c_1; c_2$ denotes sequential composition and $c_1 \parallel c_2$ denotes parallel composition. Note that, in $c_1 \parallel c_2$, c_1 and c_2 are built from separate variable contexts Γ and Δ. This ensures that the two parallel commands do not interfere. (We are implicitly using Reynolds's interference control principle that distinct identifiers do not interfere.)

The operational semantics of this fragment of Algol is standard. See, for example, [Gun92]. To execute a command $x_1 : \mathbf{var}, \ldots, x_n : \mathbf{var} \vdash c : \mathbf{comm}$ one uses a state $\sigma \in \omega^n$. Execution is then defined as a relation of the form $(c, \sigma) \Downarrow \sigma'$ which means executing c in state σ yields the final state σ'. For non-interfering parallel composition (which is not standard), one uses the rule:

$$\frac{(c_1, \sigma_1) \Downarrow \sigma_1' \quad (c_2, \sigma_2) \Downarrow \sigma_2'}{(c_1 \parallel c_2, (\sigma_1, \sigma_2)) \Downarrow (\sigma_1', \sigma_2')}$$

For evaluating expressions $x_1 : \mathbf{var}, \ldots, x_n : \mathbf{var} \vdash e : \mathbf{exp}$, one similarly uses an evaluation relation $(e, \sigma) \Downarrow n$ (for $n \in \omega$).

[7]Unfortunately, this terminology conflicts with the mathematical usage of the term "variable". In this section, we use Algol's term "identifier" for mathematical variables, and use "variable" for storage cells.

The identity and structural rules for this fragment of Algol are shown in Fig. 3. Note that contraction is conspicuously absent. Had we allowed contraction, we would be able to have two identifiers denoting the same variable and this would violate the principle that distinct identifiers do not interfere. See [O'H91, O'H] for further discussion of this aspect.

4.2 To interpret the primitive types of Algol, associate a coherent space θ° with each type θ as follows:

$$\begin{array}{rcl} \mathbf{exp}^\circ &=& \mathit{int} \\ \mathbf{comm}^\circ &=& \mathbf{1} \\ \mathbf{var}^\circ &=& \mathit{int} \mathbin{\&} \mathit{int}^\perp \end{array}$$

(Recall that $\mathit{int}^\perp \cong \mathit{int} \multimap \mathbf{1}$.) We write the tokens of $\mathbf{var}^\circ$ as $\mathrm{get}\,.\,i$ and $\mathrm{put}\,.\,j$ corresponding to the int and $\mathit{int}^\perp$ components respectively.

We interpret a phrase with typing

$$x_1 : \theta_1, \ldots, x_n : \theta_n \vdash p : \theta$$

as a linear map of type

$$\dagger\theta_1^\circ \otimes \ldots \otimes \dagger\theta_n^\circ \multimap \theta^\circ$$

Let θ^* denote the coherent space $\dagger\theta^\circ$. For a type assignment Γ such as the one above, let Γ^* denote the coherent space $\dagger\theta_1^\circ \otimes \ldots \otimes \dagger\theta_n^\circ$. There are two important maps that occur often:

$$\begin{array}{lcl} \mathbf{ss} : \Gamma^* \to \Gamma^* \rhd \Gamma^* &=& \bigotimes \dagger\Gamma^\circ \xrightarrow{\bigotimes \mathbf{seq}} \bigotimes(\dagger\Gamma^\circ \rhd \dagger\Gamma^\circ) \longrightarrow (\bigotimes \dagger\Gamma^\circ) \rhd (\bigotimes \dagger\Gamma^\circ) \\ \mathbf{SS} : \Gamma^* \to \dagger\Gamma^* &=& \bigotimes \dagger\Gamma^\circ \xrightarrow{\bigotimes \mathbf{Seq}} \bigotimes \dagger\dagger\Gamma^\circ \xrightarrow{\mathbf{sprom}} \dagger(\bigotimes \dagger\Gamma^\circ) \end{array}$$

where the second map in the first definition is an appropriate combination of weak distributivity injections mentioned in 2.9. If $f : \Gamma^* \to \theta^\circ$, we write **skleisli** f for the map $\mathbf{SS};\, \dagger f : \Gamma^* \to \theta^*$.

The phrases of Fig. 2 can now be interpreted as follows:

$$\begin{array}{rcl}
[\![\Gamma \vdash 0 : \mathbf{exp}]\!] &=& \Gamma^* \xrightarrow{\bigotimes \mathbf{done}} \bigotimes \mathbf{1} \;\cong\; \mathbf{1} \xrightarrow{[\![0]\!]} \mathit{int} \\
[\![\Gamma, \Delta \vdash e_1 + e_2 : \mathbf{exp}]\!] &=& \Gamma^* \xrightarrow{\mathbf{ss}} \Gamma^* \rhd \Gamma^* \xrightarrow{[\![\Gamma \vdash e_1 : \mathbf{exp}]\!] \rhd [\![\Gamma \vdash e_2 : \mathbf{exp}]\!]} \mathit{int} \rhd \mathit{int} \xrightarrow{+} \mathit{int} \\
[\![\Gamma \vdash \mathbf{skip} : \mathbf{comm}]\!] &=& \Gamma^* \xrightarrow{\bigotimes \mathbf{done}} \bigotimes \mathbf{1} \;\cong\; \mathbf{1} \\
[\![\Gamma \vdash \mathbf{diverge} : \mathbf{comm}]\!] &=& \Gamma^* \xrightarrow{\perp} \mathbf{1} \\
[\![\Gamma \vdash c_1 ; c_2 : \mathbf{comm}]\!] &=& \Gamma^* \xrightarrow{\mathbf{ss}} \Gamma^* \rhd \Gamma^* \xrightarrow{[\![\Gamma \vdash c_1 : \mathbf{comm}]\!] \rhd [\![\Gamma \vdash c_2 : \mathbf{comm}]\!]} \mathbf{1} \rhd \mathbf{1} \;\cong\; \mathbf{1} \\
[\![\Gamma, \Delta \vdash c_1 \,\|\, c_2 : \mathbf{comm}]\!] &=& \Gamma^* \otimes \Delta^* \xrightarrow{[\![\Gamma \vdash c_1 : \mathbf{comm}]\!] \otimes [\![\Gamma \vdash c_2 : \mathbf{comm}]\!]} \mathbf{1} \otimes \mathbf{1} \;\cong\; \mathbf{1} \\
[\![\Gamma \vdash \mathbf{new}\; x . c : \mathbf{comm}]\!] &=& \Gamma^* \;\cong\; \Gamma^* \otimes \mathbf{1} \xrightarrow{\mathbf{id} \otimes \mathbf{cell}_0} \Gamma^* \otimes \dagger(\mathit{int} \mathbin{\&} \mathit{int}^\perp) \xrightarrow{[\![\Gamma, x : \mathbf{var} \vdash c : \mathbf{comm}]\!]} \mathbf{1} \\
[\![\Gamma \vdash v := e : \mathbf{comm}]\!] &=& \Gamma^* \xrightarrow{\mathbf{ss}} \Gamma^* \rhd \Gamma^* \xrightarrow{[\![\Gamma \vdash e : \mathbf{exp}]\!] \rhd [\![\Gamma \vdash v : \mathbf{var}]\!]} \mathit{int} \rhd (\mathit{int} \mathbin{\&} \mathit{int}^\perp) \xrightarrow{\mathrm{assign}} \mathbf{1} \\
[\![\Gamma \vdash v : \mathbf{exp}]\!] &=& \Gamma^* \xrightarrow{[\![\Gamma \vdash v : \mathbf{var}]\!]} \mathit{int} \mathbin{\&} \mathit{int}^\perp \xrightarrow{\pi_0} \mathit{int}
\end{array}$$

$$\frac{\Gamma, x:\theta_1, y:\theta_2, \Delta \vdash p:\theta}{\Gamma, y:\theta_2, x:\theta_1, \Delta \vdash p:\theta}\ \textsf{Exchange} \qquad \frac{\Gamma \vdash p:\theta'}{\Gamma, x:\theta \vdash p:\theta'}\ \textsf{Weakening}$$

$$\frac{}{x:\theta \vdash x:\theta}\ \textsf{Id} \qquad \frac{\Gamma \vdash p:\theta \quad \Delta, x:\theta \vdash q:\theta'}{\Gamma, \Delta \vdash q[p/x]:\theta'}\ \textsf{Cut}$$

Figure 3: Identity and structural rules for contraction-free Algol

The map $\perp$ above is the empty linear map, $\mathbf{cell}_0$ picks out the cell trace set defined in 2.15, and **assign** is $(\mathbf{id} \triangleright \pi_1); \mathbf{sapply}$. Though we give the interpretation with coherent spaces in mind, it is clear that it applies to any cpo-enriched LLMS-category with a discrete object *int*. (Cf. 2.11 and 2.18.)

It is instructive to compose the above arrows to obtain direct linear maps. We use the following notation. A token of $[\![\Gamma \vdash p : \theta]\!]$ is written as $\mathbf{s} \mapsto \alpha$ where $\mathbf{s}$ is a vector of traces in $|\Gamma^*|$ (of the same length as that of Γ) and α a token of θ°. We show a few important cases:

$$\begin{array}{rcl} [\![\Gamma \vdash \mathbf{new}\ x.\, c : \mathbf{comm}]\!] & = & \{ (\mathbf{s} \mapsto *) : (\mathbf{s}, t \mapsto *) \in [\![\Gamma, x : \mathbf{var} \vdash c : \mathbf{comm}]\!],\ t \in \mathit{cell}_0 \} \\ [\![\Gamma \vdash v := e : \mathbf{comm}]\!] & = & \{ (\mathbf{s}_1 \cdot \mathbf{s}_2 \mapsto *) : \\ & & \qquad (\mathbf{s}_1 \mapsto i) \in [\![\Gamma \vdash e : \mathbf{exp}]\!], \\ & & \qquad (\mathbf{s}_2 \mapsto \mathrm{put}\,.\,i) \in [\![\Gamma \vdash v : \mathbf{var}]\!] \} \\ [\![\Gamma \vdash v : \mathbf{exp}]\!] & = & \{ (\mathbf{s} \mapsto i) : (\mathbf{s} \mapsto \mathrm{get}\,.\,i) \in [\![\Gamma \vdash v : \mathbf{var}]\!] \} \end{array}$$

The notation $\mathbf{s}_1 \cdot \mathbf{s}_2$ denotes component-wise concatenation of $\mathbf{s}_1$ and $\mathbf{s}_2$ (which are vectors of sequences).

4.3 The identity and structural rules of Fig. 3 are interpreted as follows:

$$\begin{array}{ll} \textsf{Exchange} & \Gamma^* \otimes \theta_2^* \otimes \theta_1^* \otimes \Delta^* \xrightarrow{\mathbf{id} \otimes \mathbf{exch} \otimes \mathbf{id}} \Gamma^* \otimes \theta_1^* \otimes \theta_2^* \otimes \Delta^* \xrightarrow{[\![\Gamma, x:\theta_1, y:\theta_2, \Delta \vdash p:\theta]\!]} \theta^\circ \\ \textsf{Weakening} & \Gamma^* \otimes \theta^* \xrightarrow{\mathbf{id} \otimes \mathbf{done}} \Gamma^* \otimes \mathbf{1} \xrightarrow{\mathbf{delr}} \Gamma^* \xrightarrow{[\![\Gamma \vdash p:\theta']\!]} \theta'^\circ \\ \textsf{Id} & \theta^* \xrightarrow{\mathbf{sread}} \theta^\circ \\ \textsf{Cut} & \Gamma^* \otimes \Delta^* \xrightarrow{(\mathbf{skleisli}\ [\![\Gamma \vdash p:\theta]\!]) \otimes \mathbf{id}} \theta^* \otimes \Delta^* \xrightarrow{\mathbf{exch}} \Delta^* \otimes \theta^* \xrightarrow{[\![\Delta, x:\theta \vdash q:\theta']\!]} \theta'^\circ \end{array}$$

Notice that these interpretations are very similar to those of intuitionistic logic [GLT89], with the only difference being that ! is replaced by †.

4.4 Examples

(i) The command $x : \mathbf{var} \vdash x := x + 1 : \mathbf{comm}$ denotes the linear map:

$$\{ (\langle \mathrm{get}\,.\,i,\ \mathrm{put}\,.\,i+1 \rangle \mapsto *) : i \in |\mathit{int}| \}$$

As shown in Sec. 3, such a linear map can also be viewed as a regular function $f : \mathcal{A}(\mathbf{var}^\circ) \to_r \mathcal{A}(\mathbf{1})$. The function maps variable trace sets to command trace sets such that, whenever the input contains a trace of the form:

$$\langle \mathrm{get} \,.\, i_1,\ \mathrm{put} \,.\, i_1 + 1,\ \ldots,\ \mathrm{get} \,.\, i_n,\ \mathrm{put} \,.\, i_n + 1 \rangle$$

the output contains $\langle * \rangle^n$. In particular, the cell trace set contains all traces of the form

$$\langle \mathrm{get}.i,\ \mathrm{put} \,.\, i + 1,\ \mathrm{get} \,.\, i + 1,\ \mathrm{put} \,.\, i + 2,\ \ldots,\ \mathrm{get} \,.\, i + (n-1),\ \mathrm{put} \,.\, i + n \rangle$$

So, given an n-fold approximation of the cell trace set, the command gives a sequence of n $*$'s. The best way to read this is to say that, if the command $x := x+1$ is executed n times then an integer cell with an initial value i passes through some intermediate states and ends in the final state $i + n$.

(ii) The command $x : \mathbf{var},\ y : \mathbf{var} \vdash (x := 1 \parallel y := 2) : \mathbf{comm}$ is interpreted as the linear map

$$\{ \langle \mathrm{put} \,.\, 1 \rangle,\ \langle \mathrm{put} \,.\, 2 \rangle \mapsto * \}$$

(iii) The command $v : \mathbf{var} \vdash \mathbf{new}\ x.\, (x := x+1; v := x) : \mathbf{comm}$ receives the interpretation

$$\{ \langle \mathrm{put} \,.\, 1 \rangle \mapsto * \}$$

Notice that, by making x a local variable, we suppress all its intermediate states from the interpretation.

(iv) The command $\vdash \mathbf{new}\ x.\, x := x + 1 : \mathbf{comm}$ gets the trivial interpretation

$$\{ \ \mapsto * \}$$

The creation of a local variable (in a block structure discipline) has no effect on the observable behavior. In fact, every closed command phrase is equivalent to one of **skip** and **diverge**.

Next, we look at some cases that illustrate the limitations of the above interpretation:

(v) The command $x : \mathbf{exp},\ v : \mathbf{var},\ w : \mathbf{var} \vdash (v := x;\ w := x) : \mathbf{comm}$ gets the linear map

$$\{ \, (\langle i, j \rangle,\ \langle \mathrm{put} \,.\, i \rangle,\ \langle \mathrm{put} \,.\, j \rangle \mapsto *) : i, j \in |int| \, \}$$

Since no two distinct identifiers interfere, the assignments to v and w do not affect the value of the expression x. So, both the uses of the expression must obtain the same integer i. Using $\dagger int$ to model inputs of type **exp** does not model this aspect.

(vi) The command $u : \mathbf{var},\ v : \mathbf{var},\ w : \mathbf{var} \vdash (v := u;\ w := u) : \mathbf{comm}$ gets the interpretation:

$$\{ \, (\langle \mathrm{get} \,.\, i,\ \mathrm{get} \,.\, j \rangle,\ \langle \mathrm{put} \,.\, i \rangle,\ \langle \mathrm{put} \,.\, j \rangle \mapsto *) : i, j \in |int| \, \}$$

The same problem reappears. The two uses of u are presumed to give possibly different integers i and j.

Handling these limitations requires a treatment of "passive types" which is beyond the scope of this paper. But, see 4.11 for some remarks.

4.5 Theorem (Adequacy) *If* $\vdash c : \mathbf{comm}$ *is a command,* $(\ \mapsto *) \in [\![\vdash c : \mathbf{comm}]\!]$ *iff* $(c, \sigma_0) \Downarrow \sigma_0$*.* ($\sigma_0$ *is the empty state.*)

To prove this result, we need some definitions. Let s be a trace in $\mathbf{var}^*|$. Define the relation $\xrightarrow{s} \subseteq \omega \times \omega$ inductively by

- $i \xrightarrow{\langle\rangle} i$ for all $i \in \omega$.
- $i \xrightarrow{\langle \mathrm{get}.j \rangle \cdot s} i'$ iff $i = j$ and $i \xrightarrow{s} i'$.
- $i \xrightarrow{\langle \mathrm{put}.j \rangle \cdot s} i'$ iff $j \xrightarrow{s} i'$.

Think of $i \xrightarrow{s} j$ as stating that a trace s takes a cell with state i to a state j. Note that this holds only if $s \in$ *cell* and that there is at most one j of this form for a given initial state i. Similarly, if $\mathbf{s} \in \mathit{cell}^n$, define $\xrightarrow{\mathbf{s}} \subseteq \omega^n \times \omega^n$ as the evident extension of $\xrightarrow{s}$.

4.6 Lemma *Let* Γ *be a type context of the form* $x_1 : \mathbf{var}, \ldots, x_n : \mathbf{var}$*.*

(i) *If* $\Gamma \vdash e : \mathbf{exp}$ *and* $(e, \sigma) \Downarrow i$ *then there exists* $\mathbf{s} \in \mathit{cell}^n$ *such that* $\sigma \xrightarrow{\mathbf{s}} \sigma$ *and* $(\mathbf{s} \mapsto i) \in [\![\Gamma \vdash e : \mathbf{exp}]\!]$*.*

(ii) *If* $\Gamma \vdash c : \mathbf{comm}$ *and* $(c, \sigma) \Downarrow \sigma'$ *then there exists* $\mathbf{s} \in \mathit{cell}^n$ *such that* $\sigma \xrightarrow{\mathbf{s}} \sigma'$ *and* $(\mathbf{s} \mapsto *) \in [\![\Gamma \vdash c : \mathbf{comm}]\!]$*.*

The proof is by induction on the definition of "$\Downarrow$".

4.7 Definition Call a context Γ a *variable context* if it is of the form $x_1 : \mathbf{var}, \ldots, x_n : \mathbf{var}$. The *computable* phrases of primitive Algol are inductively defined as follows (with Γ a variable context):

(i) An expression $\Gamma \vdash e : \mathbf{exp}$ is computable if, whenever $(\mathbf{s} \mapsto i) \in [\![\Gamma \vdash e : \mathbf{exp}]\!]$, for all states σ, σ' such that $\sigma \xrightarrow{\mathbf{s}} \sigma'$, $\sigma = \sigma'$ and $(e, \sigma) \Downarrow i$.

(ii) A command $\Gamma \vdash c : \mathbf{comm}$ is computable if whenever $(\mathbf{s} \mapsto *) \in [\![\Gamma \vdash c : \mathbf{comm}]\!]$, for all states σ, σ' such that $\sigma \xrightarrow{\mathbf{s}} \sigma'$, $(c, \sigma) \Downarrow \sigma'$.

(iii) A variable $\Gamma \vdash x : \mathbf{var}$ is computable.

(iv) A phrase $\Gamma, x_1 : \theta_1, \ldots x_n : \theta_n \vdash p : \theta$ is computable if, for all phrases $\Gamma_i \vdash p_i : \theta_i$ such that Γ_i is a variable context, $\Gamma, \Gamma_1, \ldots, \Gamma_n \vdash p[p_1/x_1, \ldots, p_n/x_n] : \theta$ is computable.

4.8 Lemma *All phrases* $\Gamma \vdash p : \theta$ *are computable.*

The proof is by induction on the type derivation of $\Gamma \vdash p : \theta$. Theorem 4.5 follows from the two lemmas.

Higher types

4.9 The type system is extended to higher-type values by adding function types $\theta \to \theta'$. This type denotes procedures which do not interfere with their arguments. The phrases are given by the type rules:

$$\frac{\Gamma, x : \theta \vdash p : \theta'}{\Gamma \vdash \lambda x.p : \theta \to \theta'} \qquad \frac{\Gamma \vdash p : \theta \quad \Delta, x : \theta' \vdash q : \theta''}{\Gamma, \Delta, f : \theta \to \theta' \vdash q[fp/x] : \theta''}$$

The operational semantics of function application is beta-reduction, as usual. The coherent semantics is given by the interpretation: $(\theta \to \theta')^\circ = \dagger\theta^\circ \multimap \theta'^\circ$. The phrases are then interpreted as follows:

$$\begin{array}{rcl} [\![\Gamma \vdash \lambda x.p : \theta \to \theta']\!] & = & \{\, (\mathbf{s} \mapsto (t, \beta)) : (\mathbf{s}, \mathbf{t} \mapsto \beta) \in [\![\Gamma, x : \theta \vdash p : \theta']\!] \,\} \\ [\![\Gamma, \Delta, f : \theta \to \theta' \vdash q[fp/x] : \theta'']\!] & = & \{\, ((\mathbf{s_1} \cdot \ldots \cdot \mathbf{s_n}), \mathbf{t}, \langle (u_1, \beta_1), \ldots, (u_n, \beta_n) \rangle \mapsto \gamma) : \\ & & \quad (\mathbf{s_1} \mapsto u_1), \ldots, (\mathbf{s_n} \mapsto u_n) \in \textbf{skleisli}\ [\![\Gamma \vdash p : \theta]\!], \\ & & \quad (\mathbf{t}, \langle \beta_1, \ldots, \beta_n \rangle \mapsto \gamma) \in [\![\Delta, x : \theta' \vdash q : \theta'']\!] \,\} \end{array}$$

A special case of the second definition is often useful:

$$[\![\Gamma, f : \theta \to \theta' \vdash fp : \theta'']\!] \quad = \quad \{\, (\mathbf{s}, \langle (u, \beta) \rangle \mapsto \beta) : (\mathbf{s} \mapsto u) \in \textbf{skleisli}\ [\![\Gamma \vdash p : \theta]\!] \,\}$$

Similarly, product types can be added in the usual fashion. Their interpretation is $(\theta_0 \times \theta_1)^\circ = \theta_0^\circ \,\&\, \theta_1^\circ$.

4.10 Examples The following convention is useful in writing Algol programs. Let $\neg$ be a type constructor defined by $\neg\theta = \theta \to \mathbf{comm}$. It denotes "$\theta$-consumers". Then the type $\neg\neg\theta$ denotes θ-consumer-consumers whose information content is essentially the same as that of θ, except that a $\neg\neg\theta$-typed computation can perform some state-manipulation before producing a θ-typed value. For each $s \in |\theta^*|$, there is a token $s^A = \langle s \mapsto * \rangle \mapsto *$ in $|(\neg\neg\theta)^\circ|$.

We consider the example objects mentioned in 2.15.

(i) A counter object is of type $\mathit{counter} = \mathbf{exp} \times \mathbf{comm}$. To create a counter, we use a phrase of type $\neg\neg\mathit{counter}$:

$$\lambda k.\, \mathbf{new}\ x.\, k\ (x,\, x := x + 1)$$

Its meaning includes s^A for every s in the counter trace set described in 2.15.

(ii) A stepper object is meant to give a different integer for each use. However, the primitive type **exp** for integers is passive (does not allow state manipulation). We obtain a state-manipulating type by double negation: $\mathit{stepper} = \neg\neg\mathbf{exp}$. The following is a stepper that steps through the integers starting from some integer:

$$x : \mathbf{var} \vdash \lambda k.\, (x := x + 1;\ k\ x) : \mathit{stepper}$$

Such a stepper can be used multiple times sequentially. For example, if $\mathrm{print} : \mathbf{exp} \to \mathbf{comm}$ is a printing procedure, the command

$$s : \mathit{stepper} \vdash s\ \mathrm{print};\ s\ \mathrm{print} : \mathbf{comm}$$

prints two successive integers.

To create a stepper, use a phrase of type $\neg\neg\mathit{stepper}$:

$$\lambda k'.\, \mathbf{new}\ x.\, k'\ (\lambda k.\, x := x + 1;\ k\ x)$$

Its meaning consists of tokens of the form

$$\langle \langle 1, \ldots, 1 \rangle^A, \ldots, \langle i, \ldots, i \rangle^A \rangle^A$$

(iii) A passive function from steppers to steppers which increments the input integers (cf. 3.5):

$$\text{inc}\ s \quad = \quad \lambda k.\, s\, \lambda i.\, k\,(i+1)$$

Its meaning consists of tokens of the form:

$$\langle\langle i_1, \ldots, i_1\rangle^A, \ldots, \langle i_n, \ldots, i_n\rangle^A\rangle^A \mapsto \langle\langle i_1+1, \ldots, i_1+1\rangle^A, \ldots, \langle i_n+1, \ldots, i_n+1\rangle^A\rangle^A$$

(iv) An active function from steppers to steppers that accumulates the sum of the integers in a variable called *sum*:

$$\lambda s.\, \lambda k.\, s\, \lambda i.\, (sum := sum + i; k\ sum)$$

To create such an accumulator, use a phrase of type $\neg\neg(stepper \to stepper)$:

$$\lambda k.\, \mathbf{new}\ sum.\ k\ (\lambda s.\, \lambda k.\, s\, \lambda i.\, (sum := sum + i; k\ sum))$$

4.11 Passivity Reynolds [Rey78] recognized that many phrases of Algol are *passive*, *i.e.*, they do not affect the state. Such phrases have the property that they do not interfere with each other (including themselves). So, passive phrases can be reused in independent contexts. This allows us to write, for example, phrases like $v := x \parallel w := x$ and $(\lambda y.\, x+y)\ x$. The identifier x of type **exp** is a passive phrase and, so, parallel access to it is permissible.

To permit such reuse, Reynolds classifies phrase types into *passive* and *active types* as follows:

types	θ	::=	$\phi \mid \alpha$
passive types	ϕ	::=	**exp**
active types	α	::=	**var** | **comm**

Passive-typed inputs can be duplicated using a contraction rule:

$$\frac{\Gamma, x : \phi, y : \phi \vdash p : \theta}{\Gamma, z : \phi \vdash p[z/x, z/y] : \theta}\ \textsf{Contraction}$$

Higher type phrases are similarly classified into passive and active phrases: $\theta \to_P \theta'$ is the subspace of $\theta \to \theta'$ that denotes passive functions. This type is especially important because it allows a fixed point operator:

$$\text{fix} : (\theta \to_P \theta) \to_P \theta$$

A simple idea for interpreting passive phrases is to use the interpretation $\phi^* = !\phi^\circ$ instead of $\dagger\phi^\circ$. The idea has indeed been explored in an earlier abstract of the present paper [Red93]. However, more structure is needed to push this interpretation through. For example, the variable dereferencing operation of Algol requires a map of the form $\dagger\mathbf{var}^\circ \multimap !\mathbf{exp}^\circ$ which is not available in general. The fact that $!A$ is a subobject of $\dagger A$ gives a rich subtyping structure (cf. [O'H91]) and a careful analysis of this structure is necessary to properly model passivity (somewhat along the lines of Girard's correlation spaces [Gir91]). We hope to show these details in a future paper.

4.12 Previous work Finding a fully abstract model of Algol is an open problem in the study of programming languages. The traditional models are based on functions over global states [Sto77] which fail to be abstract in various ways. Reynolds and Oles [Rey81, Ole85], on the one hand, and Meyer and Sieber [MS88], on the other, focused on one impediment to full abstraction: the issue of *locality* for variables created using **new** . Apparently satisfactory solutions to this problem have recently been obtained, albeit in a rather technical setting of functor categories [OT93].

A somewhat orthogonal problem in modelling Algol is the issue of *history-sensitivity* (also called *single-threadedness*). This is the notion that a storage object exhibits a behavior dependent on its past history of operations. An example that violates history-sensitivity is the so-called "snap back" combinator which resets the state of a storage object upon completion of an operation (which might temporarily modify the state of the object). The author is aware of no previous work on modelling history-sensitivity of Algol, even though much work on concurrency addresses this problem for first-order languages [Hoa85, Mil89].

Notice that the present work addresses both the locality and history-sensitivity issues. It fails to be fully abstract for other reasons: passivity is not modelled, intermediate states are represented in the semantics, and it also faces the usual problems with respect to stable functions. However, we believe that important insights regarding locality and history-sensitivity have been obtained through this study. While it is possible to address the remaining issues by quotienting with appropriate equivalences, their detailed study must await a future paper.

5 Linear logic model of state

We consider an intuitionistic linear logic[8] with the following operators: constants $\mathbf{1}$ and $\top$; binary connectives $\otimes$, $\rhd$, $\multimap$, and $\&$; modalities ! and $\dagger$.

5.1 The left context of a sequent has the following syntax:

$$\Gamma \quad ::= \quad \epsilon \mid A \mid \Gamma_0, \Gamma_1 \mid \Gamma_0; \Gamma_1$$

with the interpretation that ϵ (empty context) denotes $\mathbf{1}$, the context Γ_0, Γ_1 denotes $\Gamma_0 \otimes \Gamma_1$ and the context $\Gamma_0; \Gamma_1$ denotes $\Gamma_0 \rhd \Gamma_1$.

A context Γ is called an *independent* context if it has no ";" connective.

By the properties mentioned in 2.10, we can associate with each context Γ,

- $|\Gamma|$, the set of formula occurrences in Γ,and
- $\leq_\Gamma$, a partial order on $|\Gamma|$ denoting sequencing constraints.

The pair $(|\Gamma|, \leq_\Gamma)$ is often called a "pomset" (partially ordered multiset). We define the pomset representation of a context inductively as follows:

$$\begin{array}{rcllcl}
|\epsilon| & = & \{\} & & & \\
|A| & = & \{A\} & \leq_A & = & \{(A, A)\} \\
|\Gamma_0, \Gamma_1| & = & |\Gamma_0| + |\Gamma_1| & \leq_{\Gamma_0,\Gamma_1} & = & \leq_{\Gamma_0} \cup \leq_{\Gamma_1} \\
|\Gamma_0; \Gamma_1| & = & |\Gamma_0| + |\Gamma_1| & \leq_{\Gamma_0;\Gamma_1} & = & \leq_{\Gamma_0} \cup \leq_{\Gamma_1} \cup |\Gamma_0| \times |\Gamma_1|
\end{array}$$

[8] "Intuitionistic" means that the sequents are of the form $\Gamma \vdash A$ or $\Gamma \vdash$ (though we have no use for the latter). "Linear" means that there are no structural rules of weakening and contraction applicable to all formulas.

Notice that the pomset representation induces an equivalence on type contexts. The equivalence is that generated by associativity, commutativity and unit laws for "," and associativity and unit laws for ";".

The notation $\Gamma[\,]$ denotes a context with a hole; the hole can be plugged with another context as in $\Gamma[\Delta]$ to obtain a context. Plugging a whole with ϵ essentially "dissolves" the hole by the unit laws.

5.2 The proof rules of LLMS are as follows.

Structural rule

$$\frac{\Gamma' \vdash A}{\Gamma \vdash A}\,\mathsf{Ser} \quad \text{if } |\Gamma| = |\Gamma'| \text{ and } \leq_\Gamma \subseteq \leq_{\Gamma'}$$

Identity rules

$$\frac{}{A \vdash A}\,\mathsf{Id} \qquad \frac{\Gamma \vdash A \quad \Delta[A] \vdash B}{\Delta[\Gamma] \vdash B}\,\mathsf{Cut}$$

Multiplicative rules:

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\,\otimes\mathcal{R} \qquad \frac{\Gamma[A, B] \vdash C}{\Gamma[A \otimes B] \vdash C}\,\otimes\mathcal{L} \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma; \Delta \vdash A \rhd B}\,\rhd\mathcal{R} \qquad \frac{\Gamma[A; B] \vdash C}{\Gamma[A \rhd B] \vdash C}\,\rhd\mathcal{L}$$

$$\frac{}{\vdash \mathbf{1}}\,\mathbf{1}\mathcal{R} \qquad \frac{\Gamma[\epsilon] \vdash C}{\Gamma[\mathbf{1}] \vdash C}\,\mathbf{1}\mathcal{L} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\,\multimap\mathcal{R} \qquad \frac{\Gamma \vdash A \quad \Delta[B] \vdash C}{\Delta[\Gamma, A \multimap B] \vdash C}\,\multimap\mathcal{L}$$

Additive rules:

$$\frac{}{\Gamma \vdash \top}\,\top\mathcal{R} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\,\&\mathcal{R} \qquad \frac{\Gamma[A] \vdash C}{\Gamma[A \,\&\, B] \vdash C}\,1\,\&\mathcal{L} \qquad \frac{\Gamma[B] \vdash C}{\Gamma[A \,\&\, B] \vdash C}\,2\,\&\mathcal{L}$$

Modalities:

$$\frac{\Gamma[\epsilon] \vdash C}{\Gamma[\dagger A] \vdash C}\,\dagger\mathsf{Weak} \qquad \frac{\Gamma[A] \vdash C}{\Gamma[\dagger A] \vdash C}\,\dagger\mathsf{Der} \qquad \frac{\Gamma[\dagger A; \dagger A] \vdash C}{\Gamma[\dagger A] \vdash C}\,\dagger\mathsf{Thread}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \dagger A}\,\dagger\mathcal{R} \quad \text{(if } \Gamma \text{ is an independent context with only ! or } \dagger \text{ formulas)}$$

$$\frac{\Gamma[\dagger A] \vdash C}{\Gamma[!A] \vdash C}\,!\mathsf{Ser} \qquad \frac{\Gamma[!A, !A] \vdash C}{\Gamma[!A] \vdash C}\,!\mathsf{Contr}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash !A}\,!\mathcal{R} \quad \text{(if } \Gamma \text{ is an independent context with only ! formulas)}$$

Note that !Ser allows !Weak and !Der as derived rules.

These proof rules can be given a term assignment in the standard fashion. See [Abr93].

5.3 Theorem *The cut rule is redundant in the LLMS sequent calculus.*

The proof proceeds in the standard fashion. Define the degree of a formula by

$$\begin{array}{rcl} \partial(A) & = & 1 \quad \text{(for } A \text{ atomic)} \\ \partial(A \odot B) & = & \max(\partial(A), \partial(B)) + 1 \\ \partial(\Box A) & = & \partial(A) + 1 \end{array}$$

and the degree of a proof by the maximum degree of its cut formulas.

5.4 Lemma *If A is a formula of degree d and $\Gamma \vdash A$ and $\Delta[A] \vdash B$ have proofs π and π' of degree smaller than d then $\Delta[\Gamma] \vdash B$ has a proof of degree smaller than d.*

The proof is by induction on the sum of the heights of the two given proofs. If π and π' end in Id or right and left rules for A (the structural rules of the modalities are considered left rules), then the symmetries of the rules allow reduction in the height of one proof. For example,

$$\dfrac{\dfrac{\Gamma \vdash A}{\Gamma \vdash !A}\,!\mathcal{R} \qquad \dfrac{\Delta[\dagger A] \vdash B}{\Delta[!A] \vdash B}\,!\mathsf{Ser}}{\Delta[\Gamma] \vdash B}\,\mathsf{Cut} \quad \Longrightarrow \quad \dfrac{\dfrac{\Gamma \vdash A}{\Gamma \vdash \dagger A}\,\dagger\mathcal{R} \qquad \Delta[\dagger A] \vdash B}{\Delta[\Gamma] \vdash B}\,\mathsf{Cut}$$

By inductive hypothesis, we conclude that $\Delta[\Gamma] \vdash B$ has a proof of degree smaller than d.

If π and π' end in left/right rules for some other formula, then we consider some appropriate subproofs of π and π' for inductive hypothesis. □

It can then be shown that every proof of degree $d > 0$ is reducible to a proof of a smaller degree. Hence, the theorem follows.

5.5 Theorem *The LLMS proof system has a semantics in* **COHL***, in fact, in any LLMS-category. Further, this semantics is preserved by cut elimination.*

The interpretation of a context is as mentioned in 5.1. A sequent $\Gamma \vdash A$ is interpreted as a linear map $\Gamma \multimap A$.

The interpretation of most rules follow from the properties of **COHL** (and other LLMS-categories) mentioned in Sec. 2. The only rules that need some care are $\dagger\mathcal{R}$ and $!\mathcal{R}$. Consider $\dagger\mathcal{R}$. Since Γ is an independent context, it is of the form $!C_1, \ldots, !C_n, \dagger D_1, \ldots, \dagger D_m$. The fact that ! and † are monoidal comonads (cf. 2.18) gives the following map:

$$\begin{array}{rcll} \Gamma & = & (\bigotimes !C_i) \otimes (\bigotimes \dagger D_j) & \xrightarrow{(\bigotimes \mathbf{Dup})\otimes(\bigotimes \mathbf{Seq})} (\bigotimes !!C_i) \otimes (\bigotimes \dagger\dagger D_j) \\ & & & \xrightarrow{(\bigotimes \mathbf{Ser})\otimes \mathbf{id}} (\bigotimes \dagger !C_i) \otimes (\bigotimes \dagger\dagger D_j) \\ & & & \xrightarrow{\mathbf{sprom}} \dagger((\bigotimes !C_i) \otimes (\bigotimes \dagger D_j)) \quad = \quad \dagger\Gamma \\ & & & \xrightarrow{\dagger[\![\Gamma \vdash A]\!]} \dagger A \end{array}$$

The interpretation of $!\mathcal{R}$ is similar.

It is a routine exercise to check that cut-elimination preserves the semantics.

5.6 Since the correspondence between LLMS-categories and the LLMS proof system is rather close, the interpretation of Algol given in Sec. 4 can be easily adapted to a translation to LLMS. This shows that LLMS is a rich framework for modelling state-manipulations.

One might wonder if we can use LLMS (with an appropriate term assignment) as a programming language. At this stage, such an exercise would seem unwise. The type system represented by LLMS is quite complex, with two kinds of connectives in type contexts. In contrast, interference-controlled Algol seems to hide much of this complexity by using a good set of combinators. While it is an open question whether LLMS can give rise to a programming framework that is superior to Algol, we believe the best use of LLMS is as

a conceptual model for thinking about programs in Algol and other languages supporting state-manipulation.

6 Conclusion

We have described here a logical model for state-manipulation using the framework of linear logic. The model is characterized abstractly in terms of a proof system as well as a categorical definition and, also, concretely in terms of coherent spaces. It is demonstrated that the model is able to handle nontrivial programming languages incorporating state-manipulation.

Many issues need further resolution. The most immediate one is modelling passivity. There seem to be two possibilities. One is to enrich coherent spaces with an "independence relation" to obtain, say, "independence spaces" and then treat $\dagger A$ as the free partially co-commutative comonoid generated by the independence space A. See [CF69, Per86] for the algebraic concepts, which have been used for modelling Petri nets [Maz89]. $!A$ is then a special case of $\dagger A$ when A is fully independent (so that $\dagger A$ becomes co-commutative). In fact, independence spaces have two choices for "with": an independent product $\&$ and a dependent product $\&_D$. The former has the rather surprising property: $\dagger(A \& B) \cong \dagger A \otimes \dagger B$. Another possibility is to use "correlation spaces" as in [Gir91]. An active correlation space is what we get by $\dagger$ in the world of coherent spaces. So, active correlation spaces are comonoids. Passive correlation spaces (the same as Girard's positive correlation spaces) can then be treated as the special case of co-commutative comonoids.

Game semantics is another exciting avenue. Note that "before" has quite simple interpretation in Abramsky-Jagadeesan games [AJ92, AJM93]. $A \triangleright B$ is the game that allows one-way switching for both the Player and the Opponent. This can in fact be extended to an LLMS-category of games which, not surprisingly, involve history-sensitive strategies. The issue here, again, is modelling passivity.

Much work remains to be done in developing a "theory" of state-manipulation using this model. The additional equivalences, which are not presently captured, must be studied. The trace-based model also need to be related to the traditional state-based models. Finally, reasoning principles must be developed as in, for example, [Hoa85, MT92].

Acknowledgements I owe much debt to Peter O'Hearn for numerous discussions and encouragement. His work on relating linear logic and interference control was the starting point for this work. Thanks are also due to Samson Abramsky, Radha Jagadeesan, Bob Tennent and Sam Kamin for contributing to this work in various ways through their discussions.

A A brief review of classical features

... to be filled in.

Essentially, define $A^{\perp} \cong A \multimap 1$ and $A \invamp B = A^{\perp} \multimap B$.

References

[Abr93] S. Abramsky. Computational interpretations of linear logic. *Theoretical Comp. Science*, 111(1-2):3–57, 1993.

[AJ92] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. Manuscript, Imperial College, 1992. (available by ftp from theory.doc.ic.ac.uk, directory /theory/papers/Abramsky.).

[AJM93] S. Abramsky, R. Jagadeesan, and P. Malacaria. Games and full abstraction for PCF: Preliminary announcement. Manuscript, Imperial College, July 1993. (available by ftp from theory.doc.ic.ac.uk, directory /theory/papers/Abramsky.).

[Ber78] G. Berry. Stable models of typed λ-calculi. In *Fifth Intern. Colloq. Automata, Languages. and Programming*, volume 62 of *Lect. Notes in Comp. Science*, pages 72–88. Springer-Verlag, 1978.

[Bri73] P. Brinch Hansen. *Operating Systems Principles.* Prentice Hall, 1973.

[CF69] P. Cartier and D. Foata. *Problèmes Combinatoires de Commutation et Réarrangements*, volume 85 of *Lect. Notes in Math.* Springer-Verlag, 1969.

[Gir87a] J.-Y. Girard. Linear logic. *Theoretical Comp. Science*, 50:1–102, 1987.

[Gir87b] J.-Y. Girard. Towards a geometry of interaction. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, pages 69–108, Boulder, Colorado, June 1987. American Mathematical Society. (Contemporary Mathematics, Vol. 92).

[Gir91] J.-Y. Girard. A new constructive logic: Classical logic. Prepublication 24, Equipe de Logique, University of Paris VII, May 1991.

[GL86] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *ACM Symp. on LISP and Functional Programming*, pages 28–38, 1986.

[GLT89] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Univ. Press, Cambridge, 1989.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.

[HMRC87] R. C. Holt, P. A. Matthews, J. A. Rosselet, and J. R. Cordy. *The Turing Programming Language.* Prentice Hall, 1987.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.

[Jac92] B. Jacobs. Semantics of weakening and contraction. Manuscript, Cambridge University, Nov 1992.

[Koc72] A. Kock. Strong functors and monoidal monads. *Archiv. der Mathematik*, XXIII:113–120, 1972.

[Laf88] Y. Lafont. The linear abstract machine. *Theoretical Comp. Science*, 59:157–180, 1988.

[Mac71] S. MacLane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[Maz89] A. Mazurkiewicz. Basic notions of trace theory. In J. S. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency*, volume 354 of *Lect. Notes*

in Comp. Science, pages 285–363. Springer-Verlag, 1989.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MS88] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *Fifteenth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 191–203. ACM, 1988.

[MT92] I. A. Mason and C. L. Talcott. References, local variables and operational reasoning. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 186–197, Santa Cruz, California, 22–25 June 1992. IEEE Computer Society Press.

[Nau60] P. Naur, *et. al.* Report on the algorithmic language ALGOL 60. *Comm. ACM*, 3(5):299–314, May 1960.

[O'H] P. W. O'Hearn. A model for syntactic control of interference. *Math. Structures in Comp. Science.* (to appear).

[O'H91] P. W. O'Hearn. Linear logic and interference control. In D. H. Pitt, editor, *Category Theory and Computer Science*, volume 350 of *Lect. Notes in Comp. Science*, pages 74–93. Springer-Verlag, 1991.

[Ole85] F. J. Oles. Type algebras, functor categories and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 543–573. Cambridge Univ. Press, Cambridge, U. K., 1985.

[ORH93] M. Odersky, D. Rabin, and P. Hudak. Call by name, assignment and the lambda calculus. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.* ACM, 1993.

[OT93] P. W. O'Hearn and R. D. Tennent. Relational parametricity and local variables. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 171–184. ACM, 1993.

[Per86] D. Perrin. Words over a partially commutative alphabet. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI Series*, pages 329–340. Springer-Verlag, 1986.

[PM87] D. Pountain and D. May. *A Tutorial Introduction to Occam Programming.* McGraw-Hill, 1987.

[Pop77] G. J. Popek *et. al.* Notes on the design of EUCLID. *SIGPLAN Notices*, 12(3):11–18, 1977.

[PW93] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.* ACM, 1993.

[Red93] U. S. Reddy. A linear logic model of state (extended abstract). Technical report, University of Glasgow, Jan 1993. (also available by ftp from theory.doc.ic.ac.uk, directory /theory/papers/Reddy.).

[Ret93a] C. Retore. Pomset logic: A truly concurrent linear calculus. Draft, Ecole des Mines de Paris, Sophia Antipolis, France, 1993.

[Ret93b] C. Retoré. *Réseaux et Séquents Ordonnés Mathématiques.* PhD thesis, Université Paris 7, Février 1993.

[Rey78] J. C. Reynolds. Syntactic control of interference. In *ACM Symp. on Princ. of Program. Lang.*, pages 39–46. ACM, 1978.

[Rey81] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[Rey89] J. C. Reynolds. Syntactic control of interference, Part II. In *Intern. Colloq. Automata, Languages. and Programming*, volume 372 of *Lect. Notes in Comp. Science*, pages 704–722. Springer-Verlag, 1989.

[See89] R. A. G. Seely. Linear logic, *-autonomous categories and cofree coalgebras. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 371–382. American Mathematical Society, 1989.

[SRI91] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In R. J. M. Hughes, editor, *Conf. on Functional Program. Lang. and Comput. Arch.*, volume 523 of *Lect. Notes in Comp. Science*, pages 192–214. Springer-Verlag, 1991.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[Wad90] P. Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. North-Holland, Amsterdam, 1990. (Proc. IFIP TC 2 Working Conf., Sea of Galilee, Israel).

[Wad91] P. Wadler. Is there a use for linear logic? In *Proc. Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255–273. ACM, 1991. (SIGPLAN Notices, Sep. 1991).

[Win80] G. Winskel. *Events in Computation*. PhD thesis, Univ. of Edinburgh, 1980.

[Win87] G. Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *Lect. Notes in Comp. Science*, pages 325–392. Springer-Verlag, 1987.

[Zha92] G.-Q. Zhang. DI-domains as prime information systems. *Information and Computation*, 100(2):151–177, 1992.

[Zha93] G.-Q. Zhang. Some monoidal closed categories of stable domains and event structures. *Math. Structures in Comp. Science*, 3:259–276, 1993.